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

Restaurant Automation Codename Adam

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

Date Submitted: March 17, 2019

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

Table of Contents

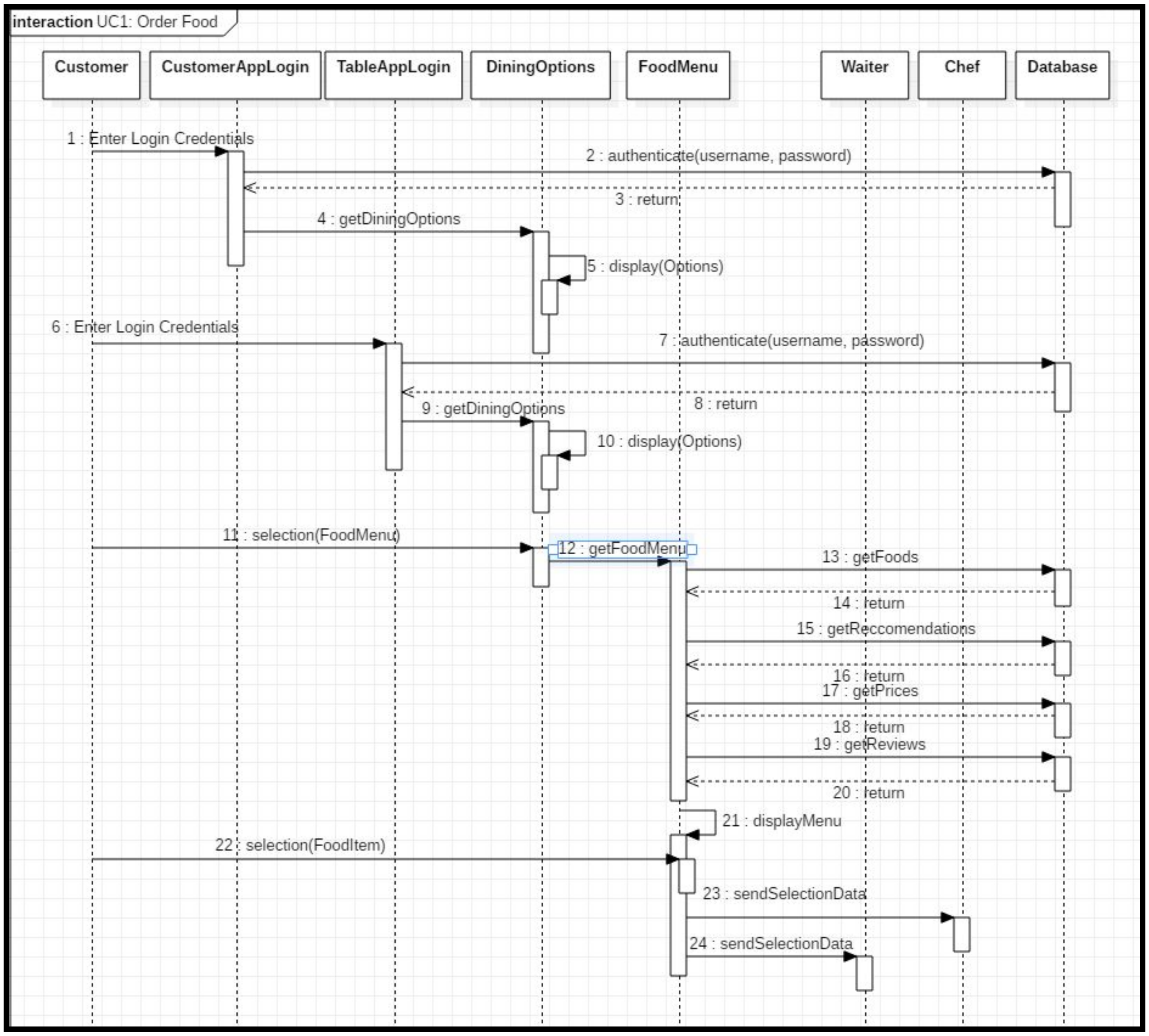
Contributions Breakdown

Everyone except for Hongpeng Zhang (who contributed 0%) contributed equally towards this report.

Section 1: Interactions Diagrams

1.1: UC-1 - Order Food

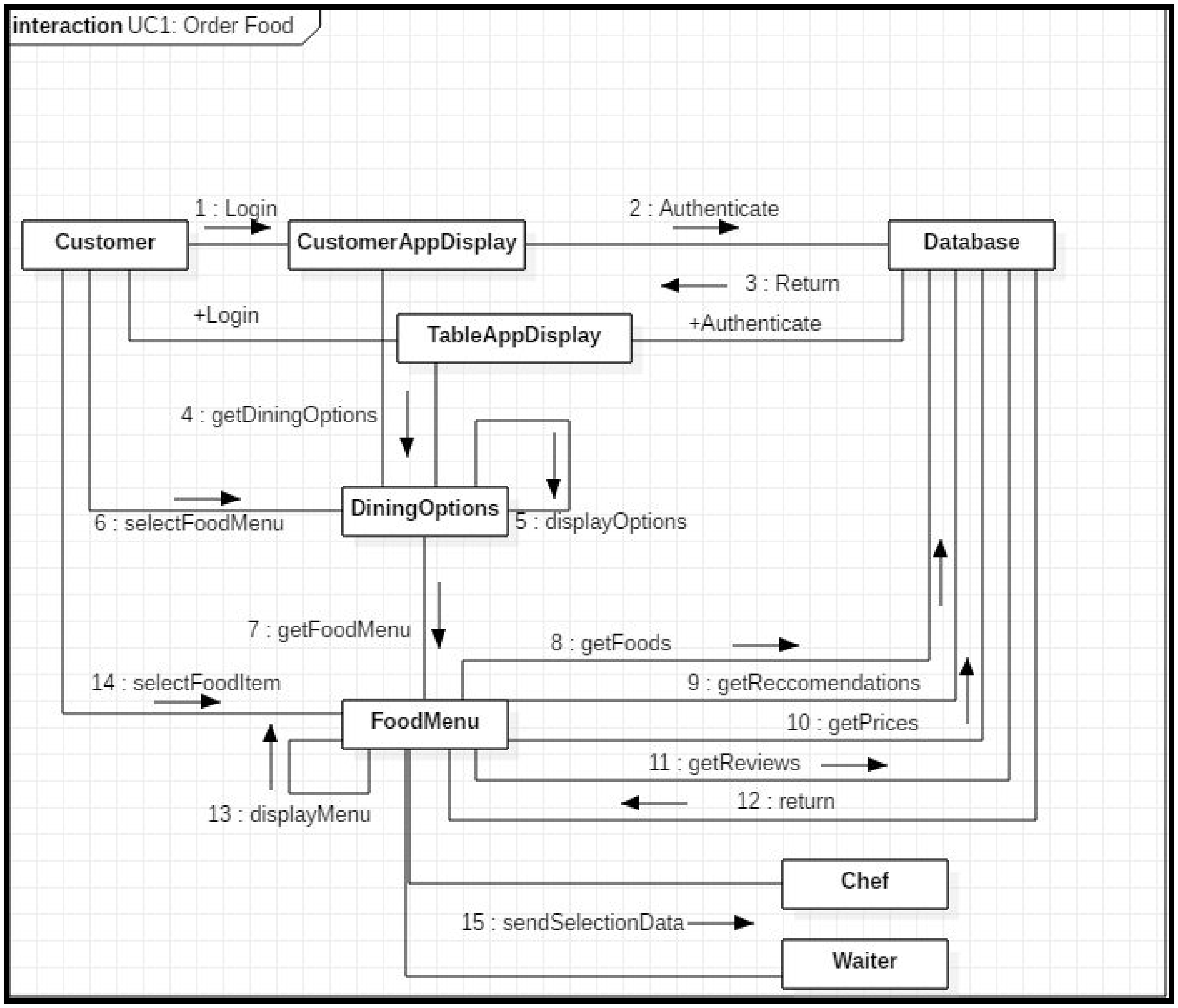
Sequence Diagram



<u>Design Principles Used:</u> 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.

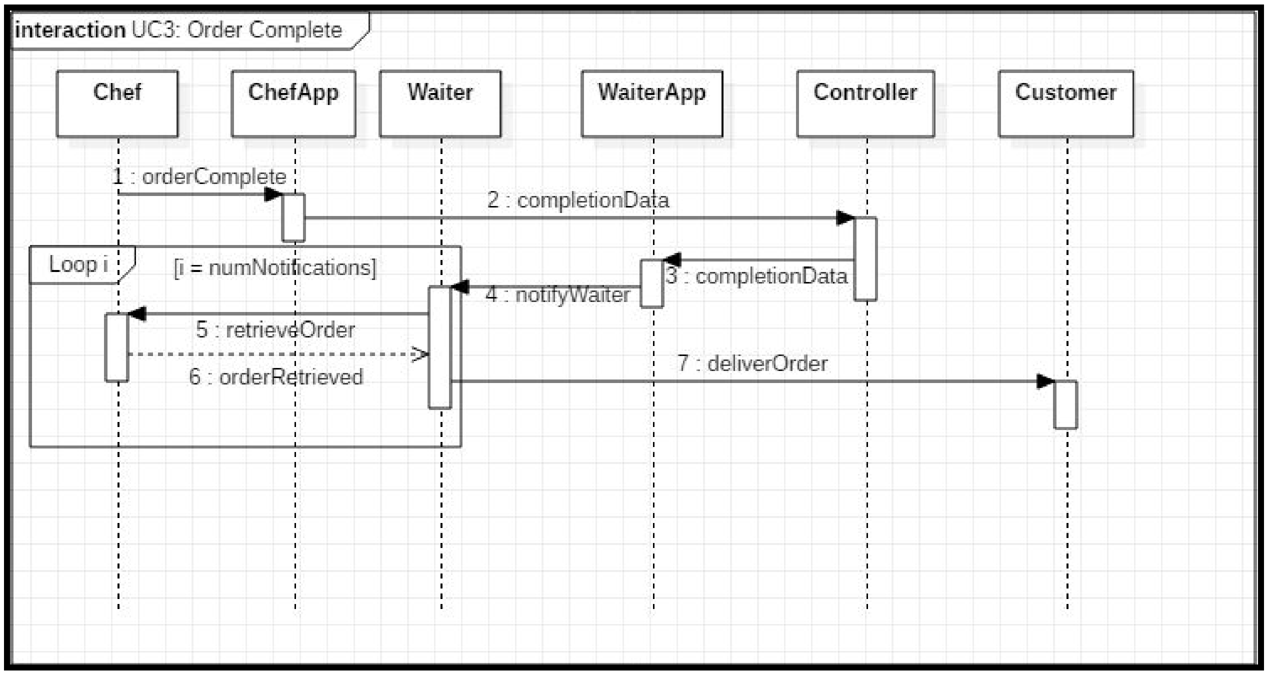
Open/close principle(OCP) is also used as the food the user orders is open for extension for adding more items after the order is placed but not for modifying by cancelling items in the existing order. Thus you can use sendSelectionData() to update the order. Creator is used as a principle design as objects are created. For example selection(food item) uses the displayMenu and getRecommendtions(), getPrice() ,getReviews() uses the menu options in the food menu.

Communication Diagram



1.2: UC-3 - Order Complete

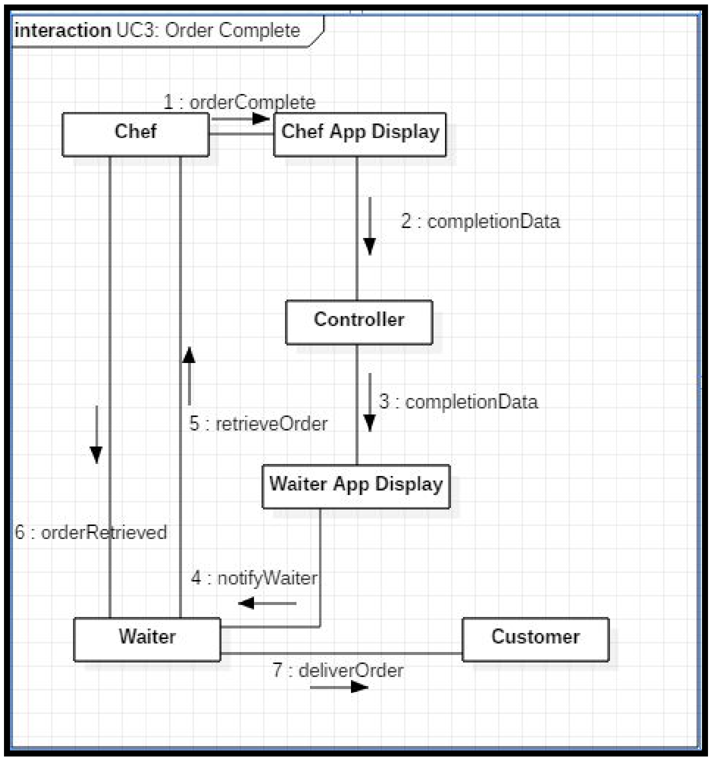
Sequence Diagram



Design Principles Used:

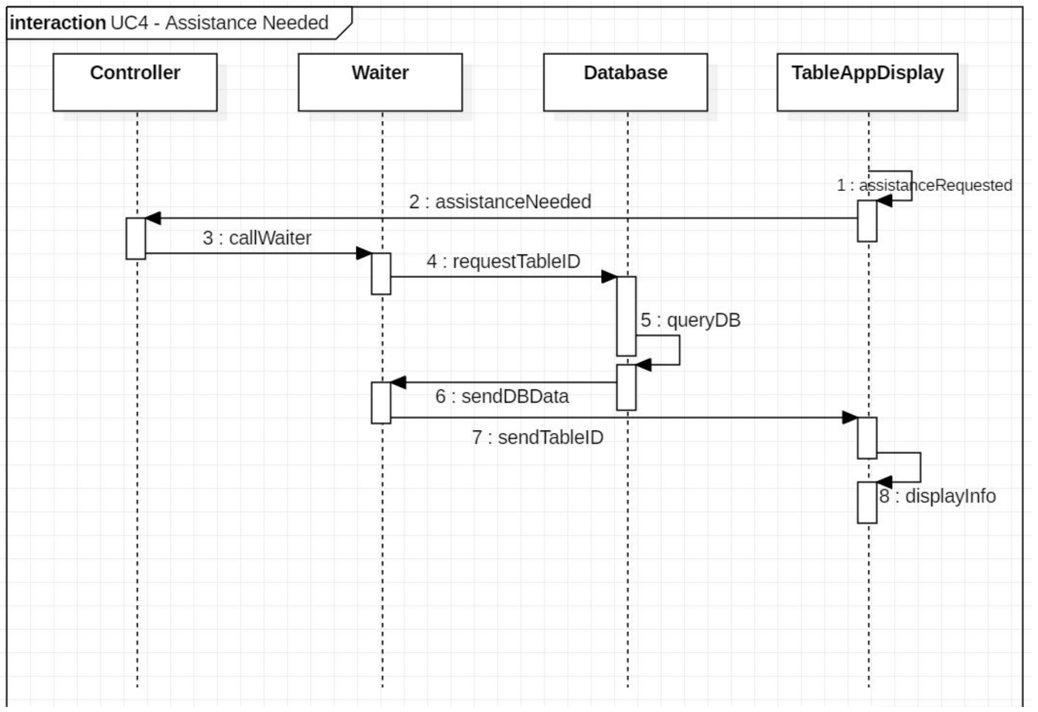
In line with the Dependency Inversion Principle, the Chef and the Waiter Apps do not communicate directly with the database, and instead have their requests routed through the Controller class. This also follows Interface Segregation Principle(ISP), because we only use four interfaces here. High cohesion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

Communication Diagram



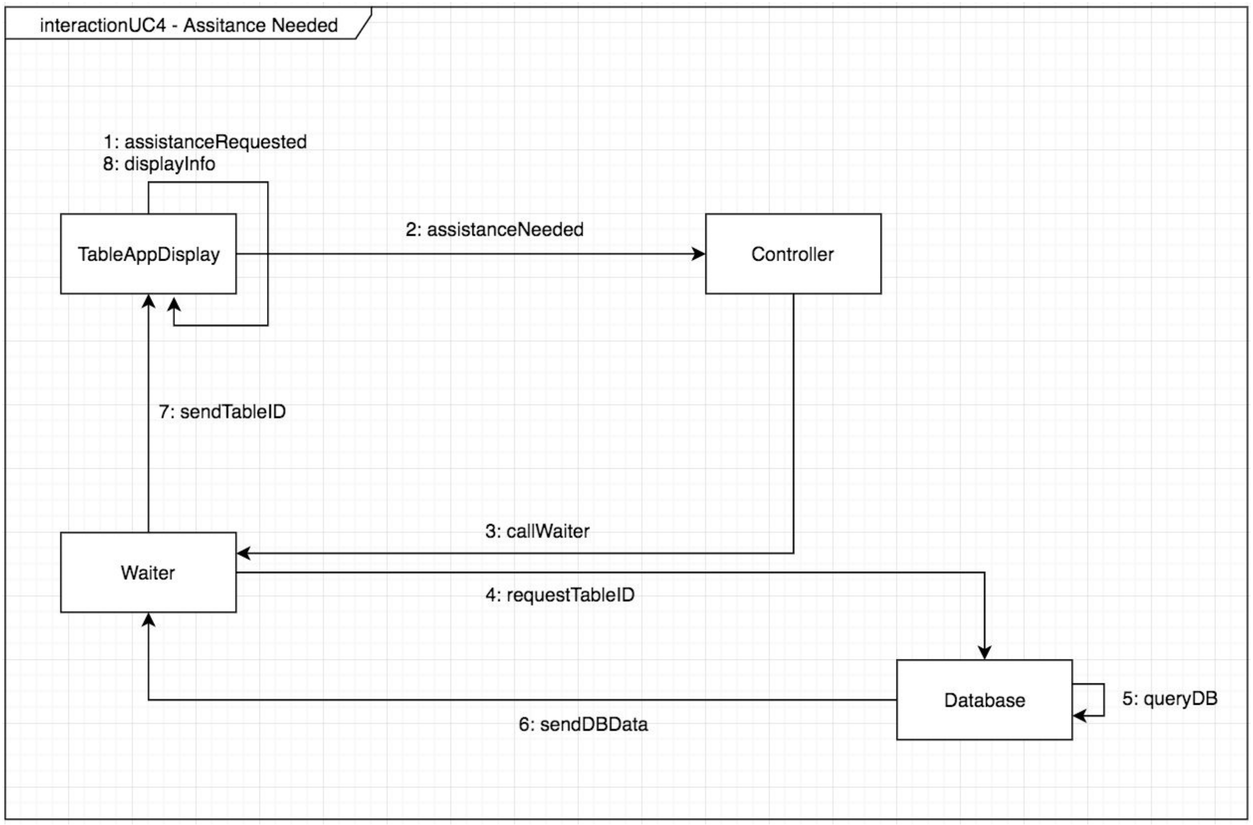
1.3: UC-4 - Assistance Needed

Sequence Diagram



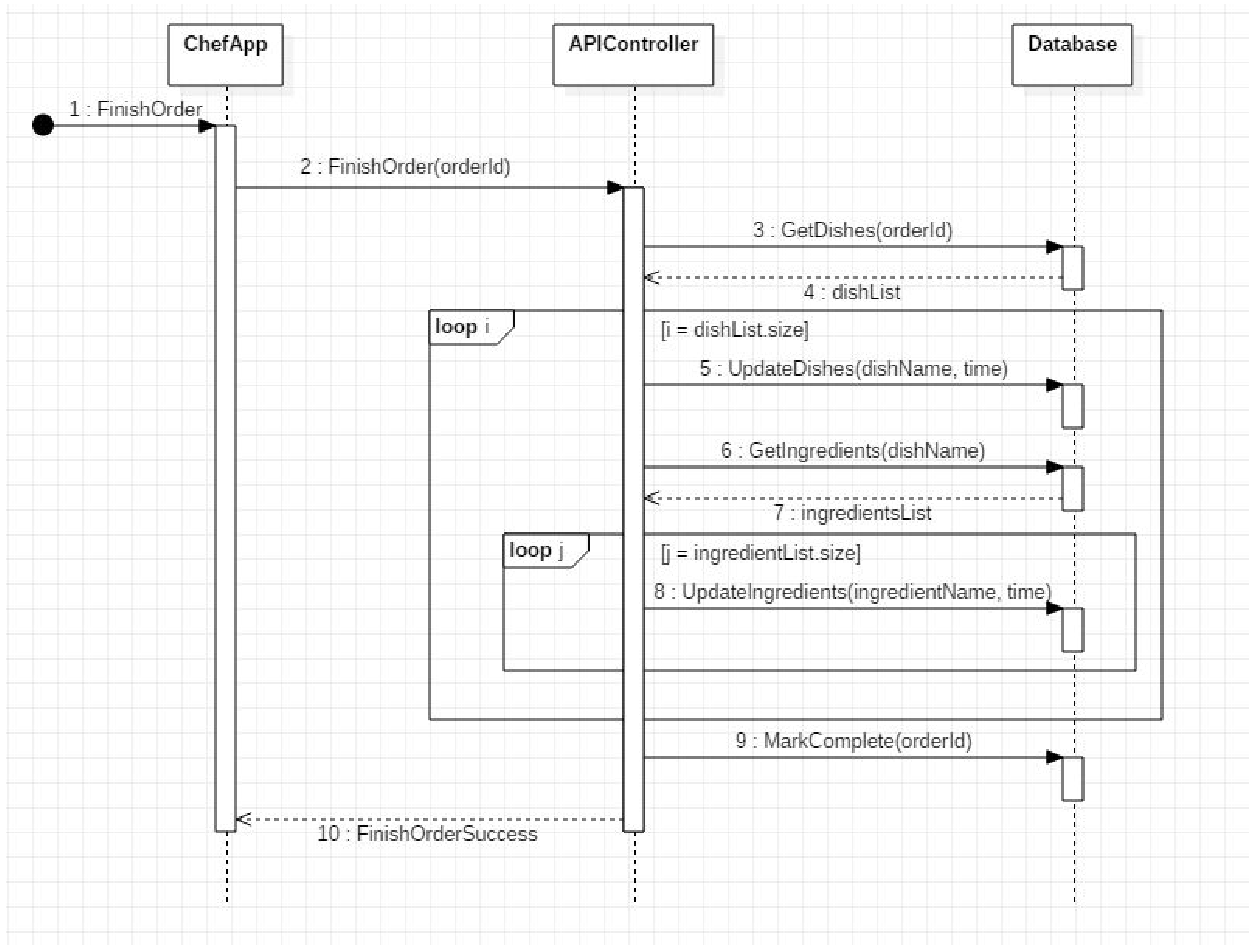
Design Principles Used: This follows the Dependency Inversion Principle (DIP) because the app has to go through the controller class in order get the correct information from the database. This also follows the Liskov Substitution Principle (LSP) because the user interfaces will only contain information 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.

Communication Diagram



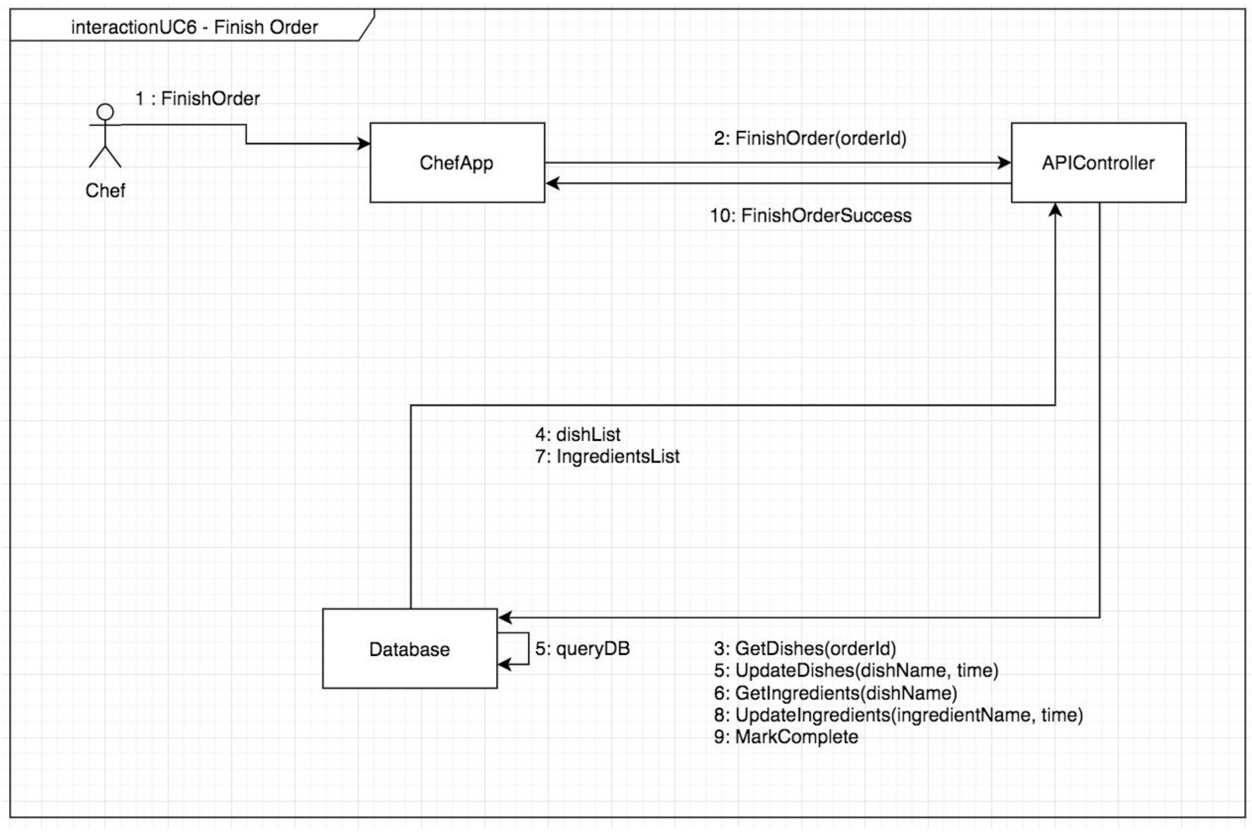
1.4: UC-6 - Finish Order

Sequence Diagram



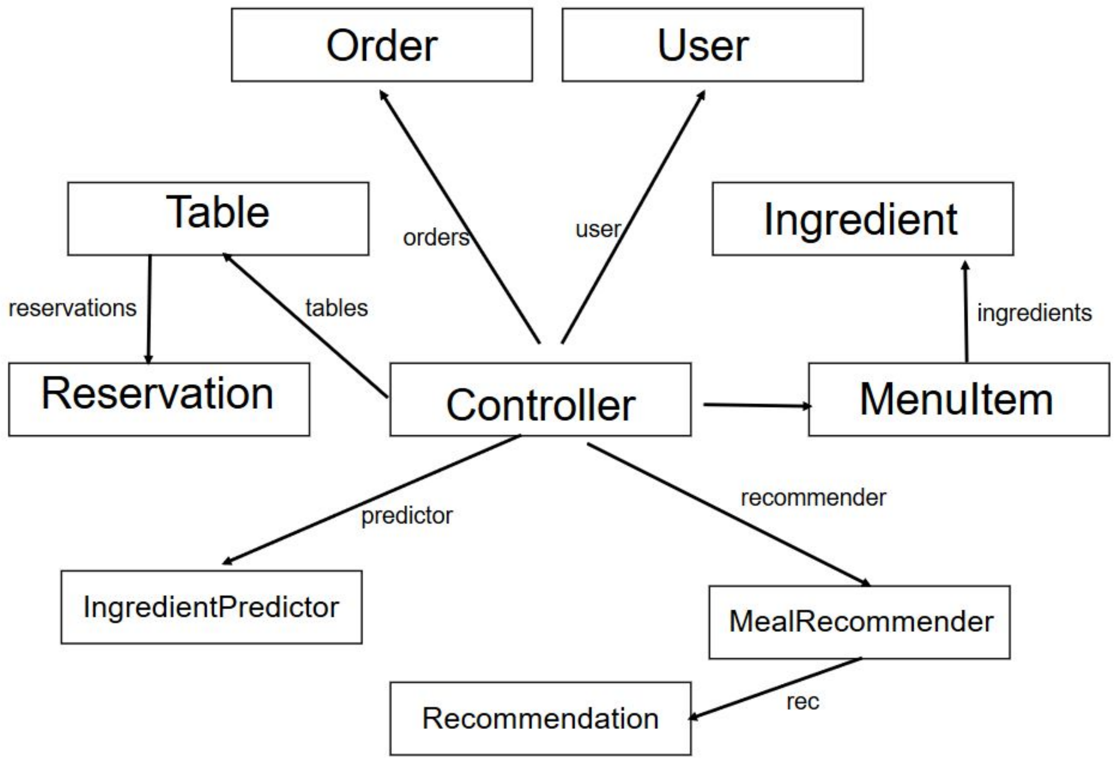
<u>Design Principles Used:</u>
Following the Dependency Inversion Principle, the Chef App sends its request to the API controller, which in turn handles the individual requests to the Database.
Following the Principle of Least Knowledge, or the Law of Demeter, the APIController has limited knowledge 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().

Communication Diagram

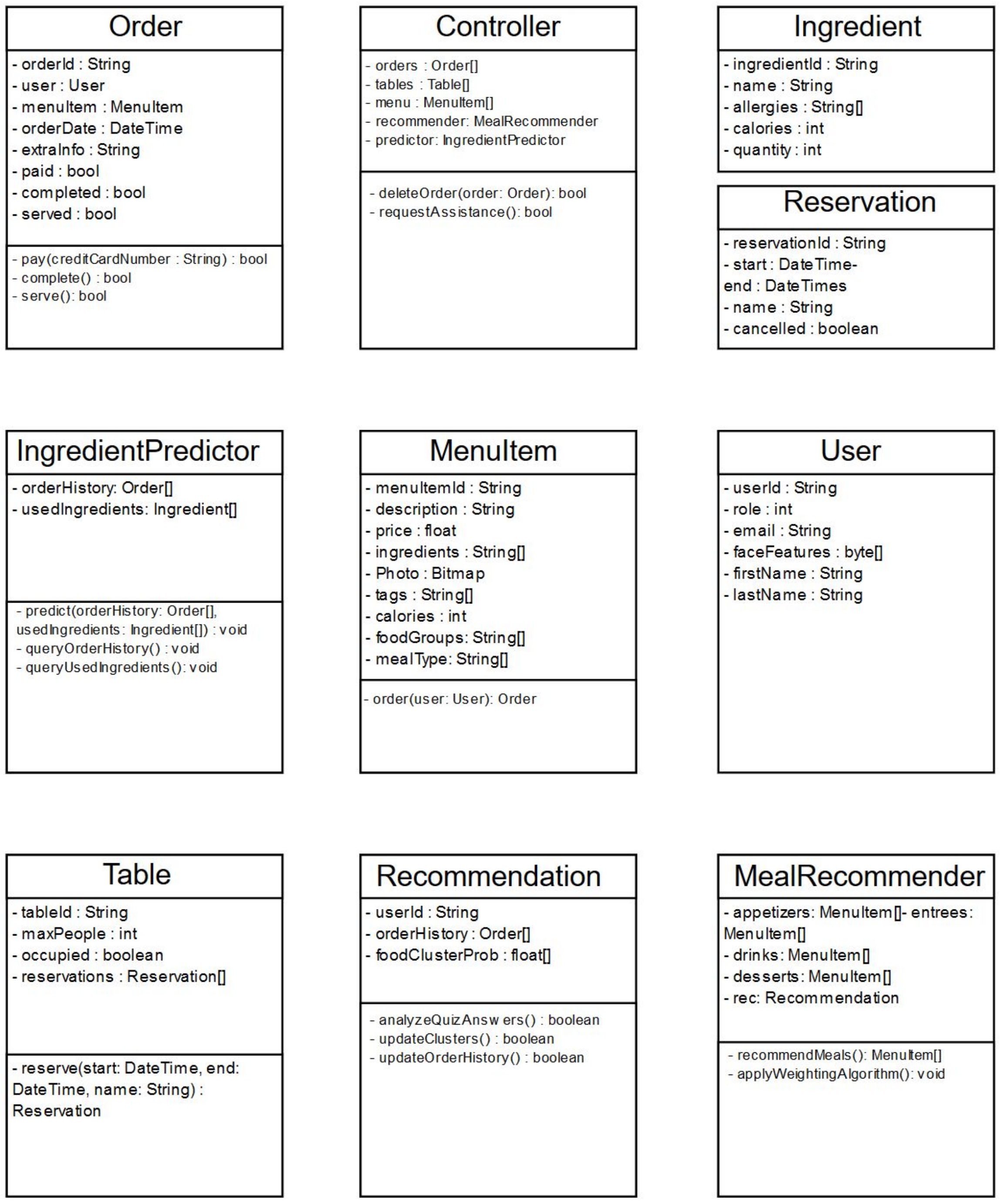


Section 2: Class Diagram and Interface Specification

2.1: Class Diagram



2.2: 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.

2.3: 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.

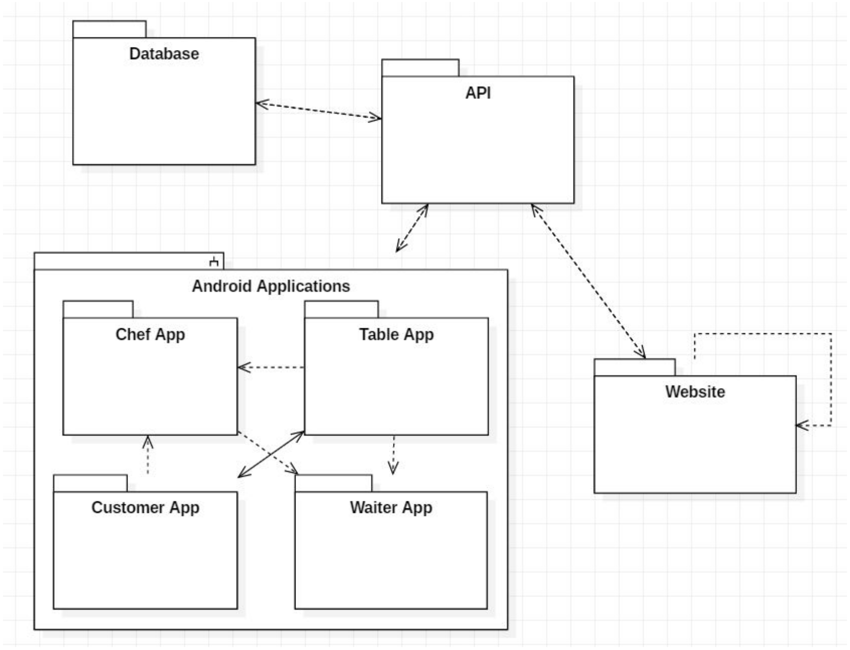
Section 3: System Architecture and System Design

3.1: 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.

3.2: 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.

3.3: 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.

3.4: 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.

3.5: 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.

3.6: 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.

3.7: Hardware Requirements

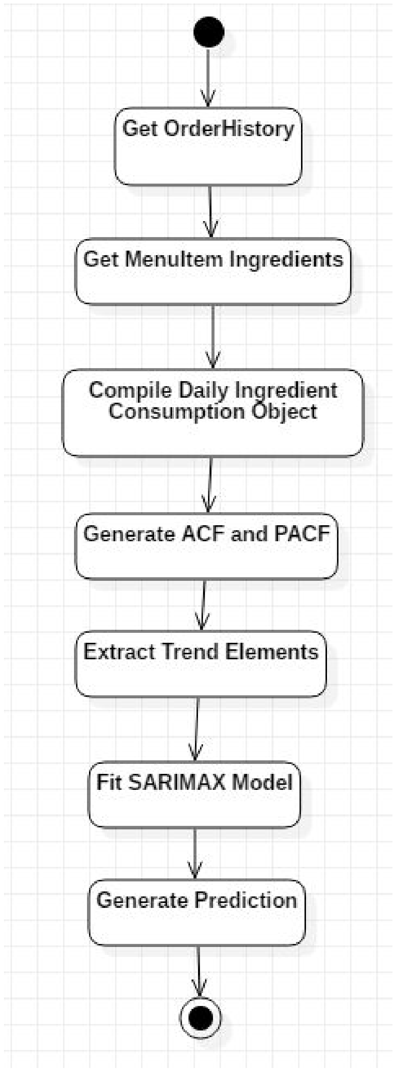
The apps will run on any mobile device running Android 5.0 or higher.
The website should work on any modern web browser (however we will only test it (and ensure it runs properly) on Microsoft Edge, Mozilla Firefox and Google Chrome).
The server should run on any hardware capable of running NodeJS, however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

Section 4: Algorithms and Data Structures

4.1: Algorithms

4.1.1: Ingredient Prediction System

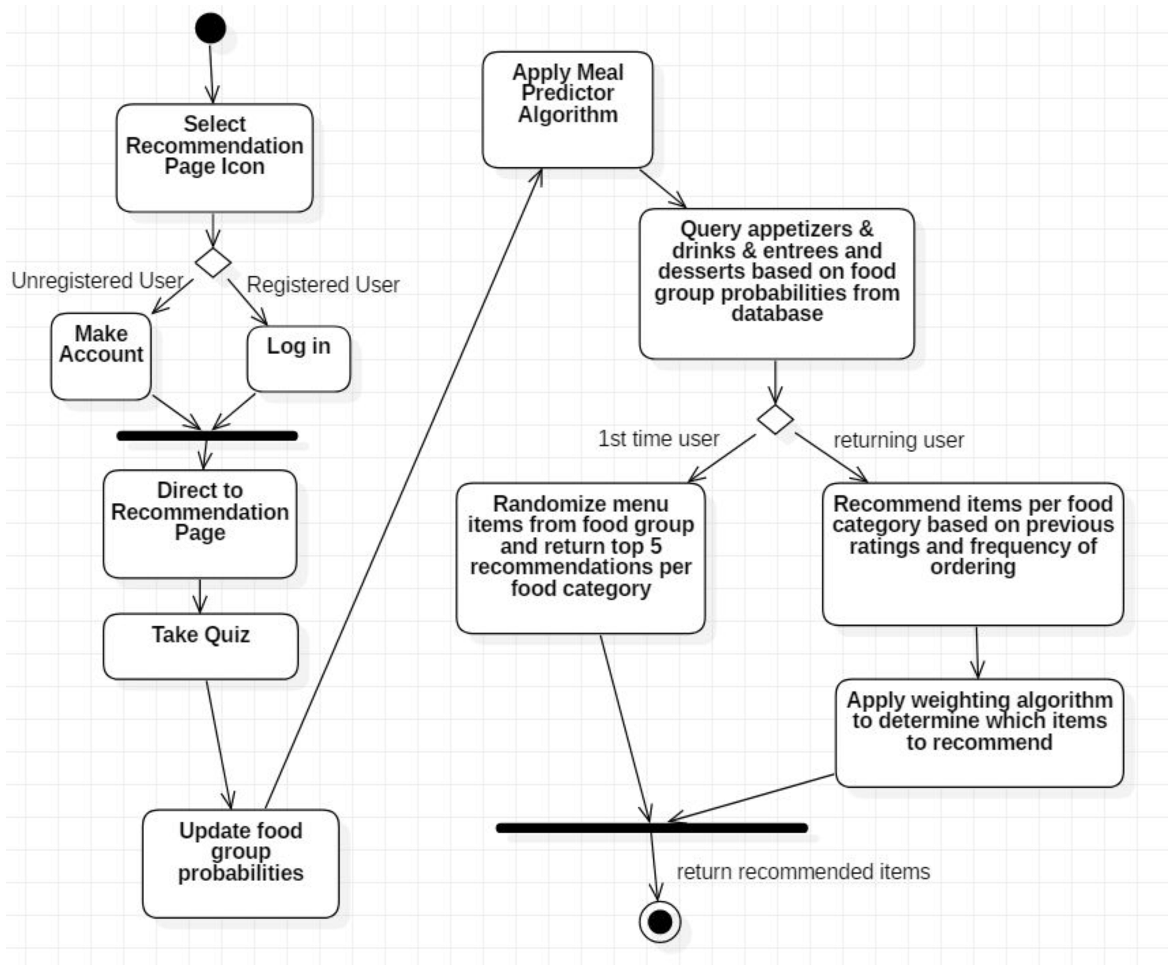
To perform ingredient prediction, the first step is to pull all of the relevant information about dish order history from the previous several weeks, and the component ingredients of the dishes from the database. This information will come in the form of a JSON object containing a list of all orders 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

4.1.2: Recommendation and Rating System

Demo 1 Goal: Content Based Filtering

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


Select Recommendation Page Icon
Unregistered User
Registered User
Make Account
Log in
Direct to Recommendation Page
Take Quiz
Update food group probabilities
Apply Meal Predictor Algorithm
Query appetizers & drinks & entrees and desserts based on food group probabilities from database
1st time user
returning user
Randomize menu items from food group and return top 5 recommendations per food category
Recommend items per food category based on previous ratings and frequency of ordering
Apply weighting algorithm to determine which items to recommend
return recommended items

How Quiz Answers are Analyzed:

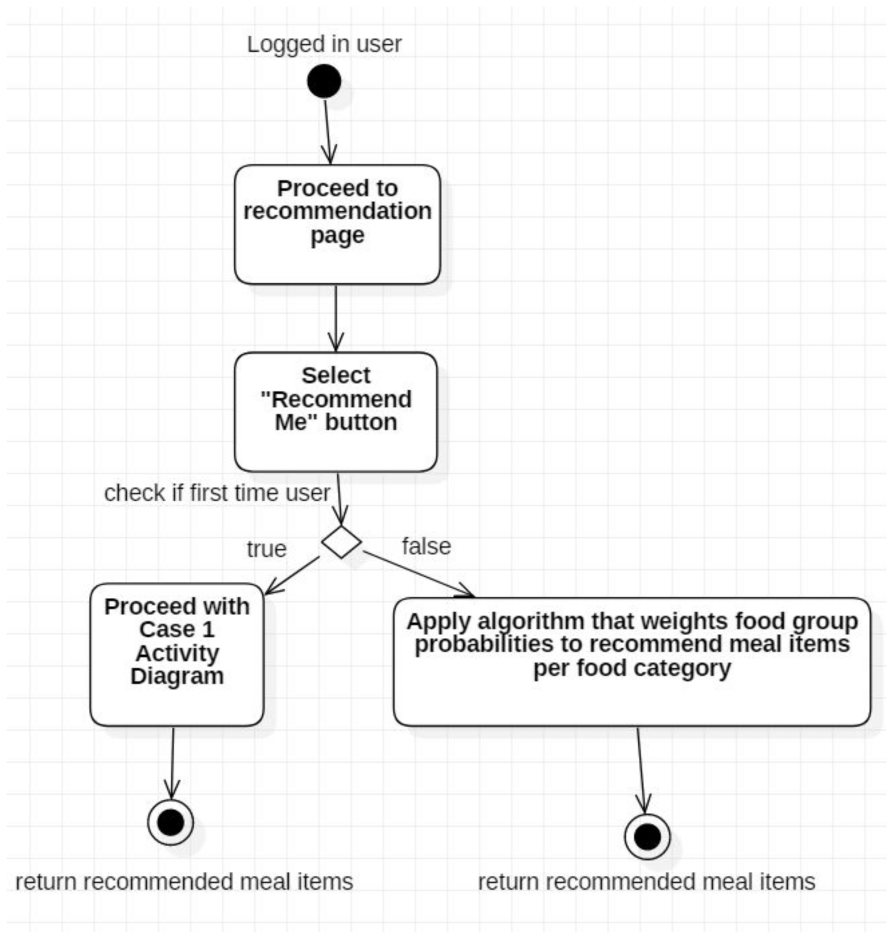
Menu items will be divided into a few supergroups (dairy, meats, vegetarian, drinks, carbohydrates etc.), which will be further subdivided into groups such as red meats, seafood, pasta, rice. The food group probabilities array will be initialized to have the same probability for each subgroup (each subgroup 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)>

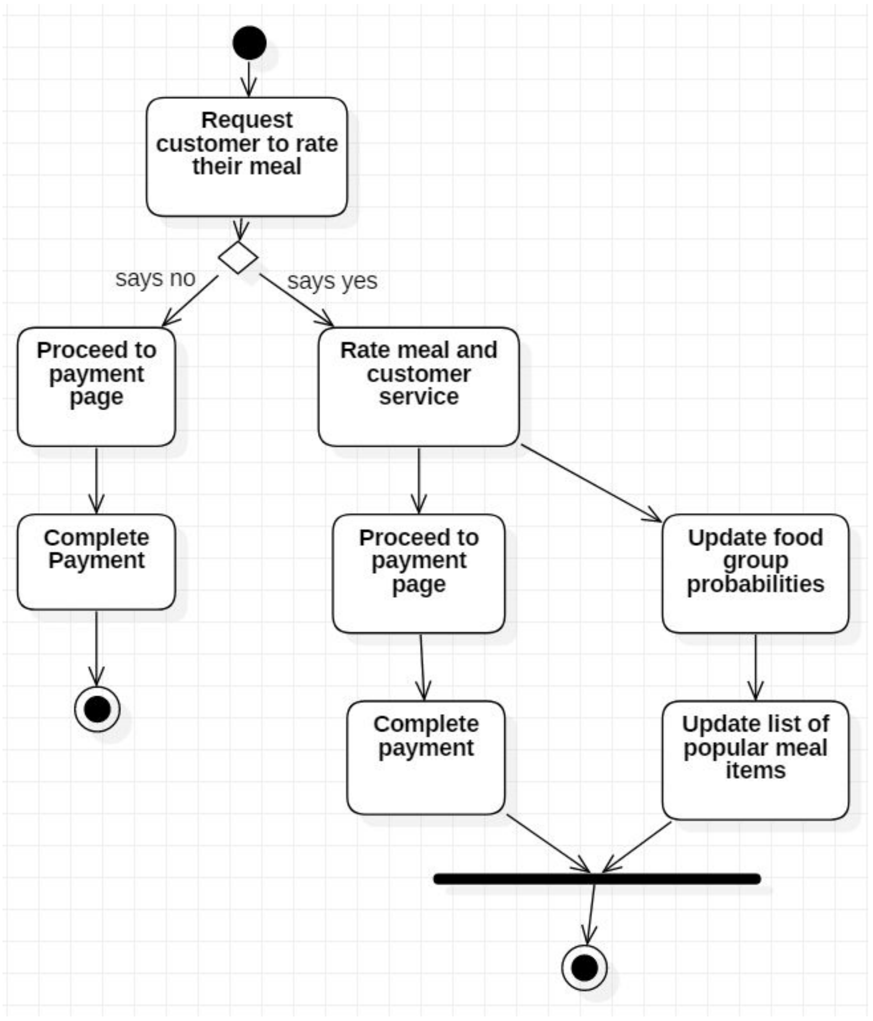
Description of Weighting Algorithm:

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

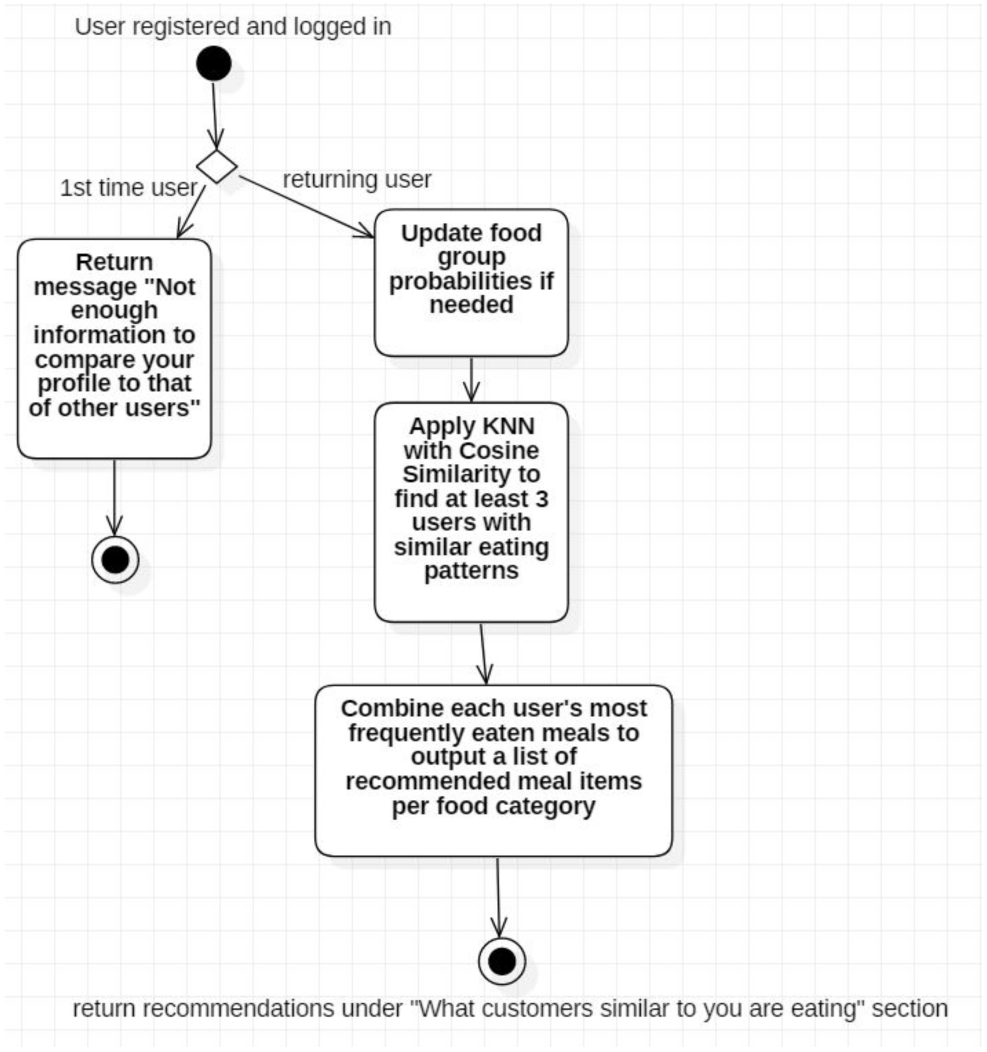

Logged in user
Proceed to recommendation page
Select "Recommend Me" button
check if first time user
true
false
Proceed with Case 1 Activity Diagram
Apply algorithm that weights food group probabilities to recommend meal items per food category
return recommended meal items
return recommended meal items

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


Request customer to rate their meal
says no
says yes
Proceed to payment page
Rate meal and customer service
Complete Payment
Proceed to payment page
Update food group probabilities
Complete payment
Update list of popular meal items

Demo 2 Goal: Collaborative Filtering

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


User registered and logged in
1st time user
returning user
Return message "Not enough information to compare your profile to that of other users"
Update food group probabilities if needed
Apply KNN with Cosine Similarity to find at least 3 users with similar eating patterns
Combine each user's most frequently eaten meals to output a list of recommended meal items per food category
return recommendations under "What customers similar to you are eating" section

Description of KNN/Cosine Similarity Algorithm

Once a user's food group probabilities array is updated, an unsupervised K-Nearest Neighbors (KNN) model trained on the food group probabilities of all customers will be used to search for users with eating patterns similar to that of the current user. This KNN model will be trained to use cosine similarity as a distance metric to determine which arrays are most similar to that of the current user's. 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.

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>

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

4.2: 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
};
```

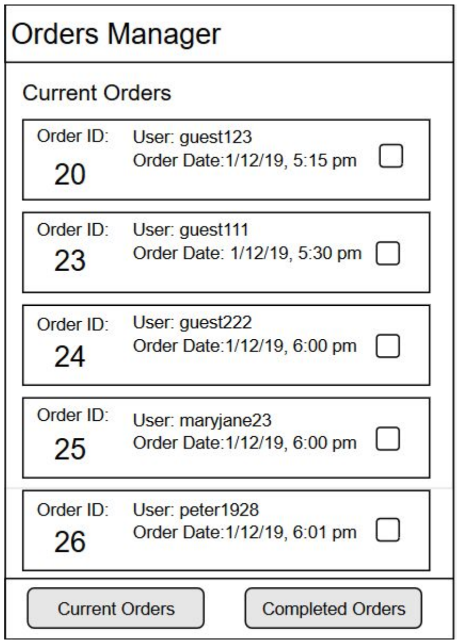
The above object may also be embedded in an ingredient object.

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 5: User Interface Design and Implementation

5.1: Chef App



Orders Manager

Current Orders

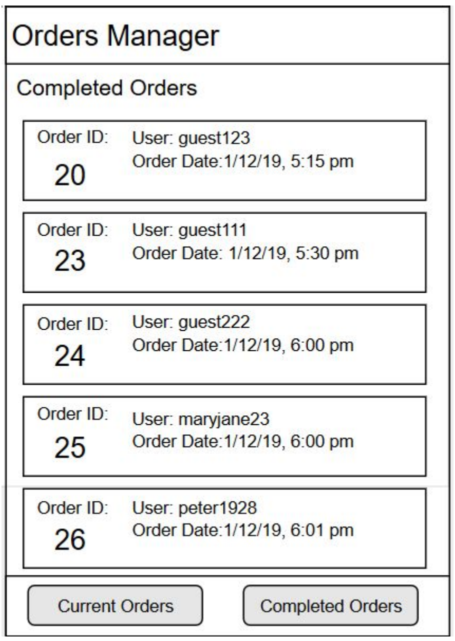
Order ID: 20 User: guest123 Order Date:1/12/19, 5:15 pm

Order ID: 23 User: guest111 Order Date: 1/12/19, 5:30 pm

Order ID: 24 User: guest222 Order Date:1/12/19, 6:00 pm

Order ID: 25 User: maryjane23 Order Date:1/12/19, 6:00 pm

Order ID: 26 User: peter1928 Order Date:1/12/19, 6:01 pm



Orders Manager

Completed Orders

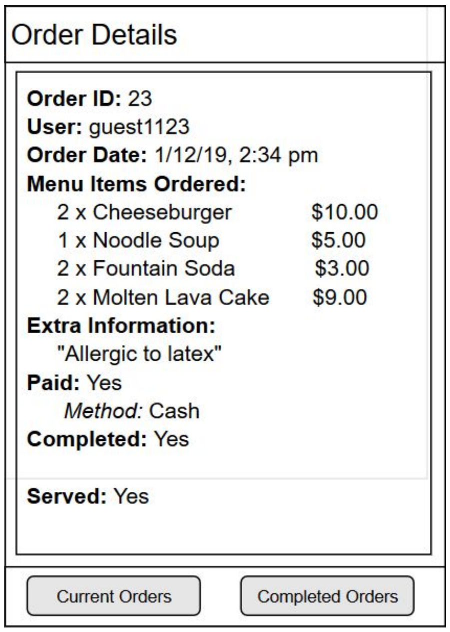
Order ID: 20 User: guest123 Order Date:1/12/19, 5:15 pm

Order ID: 23 User: guest111 Order Date: 1/12/19, 5:30 pm

Order ID: 24 User: guest222 Order Date:1/12/19, 6:00 pm

Order ID: 25 User: maryjane23 Order Date:1/12/19, 6:00 pm

Order ID: 26 User: peter1928 Order Date:1/12/19, 6:01 pm



Order Details

Order ID: 23
User: guest1123
Order Date: 1/12/19, 2:34 pm
Menu Items Ordered:

2 x Cheeseburger	$10.00
1 x Noodle Soup	$5.00
2 x Fountain Soda	$3.00
2 x Molten Lava Cake	$9.00

Extra Information:
"Allergic to latex"
Paid: Yes
Method: Cash
Completed: Yes

Served: Yes

The UI screens for the Chef App, as described in UC3, were modified so that the chef can see all orders that were completed or in progress by simply switching between two screens on the Chef App. The Orders Manager screens are almost completely implemented. To view more information about an order, the chef can simply click on an order in either the completed or current order queues. The order information screen contains more information than the report #1 mockup, as it can also include customer comments and order payment information.

5.2: Customer/Table Apps

Customer App Dashboard

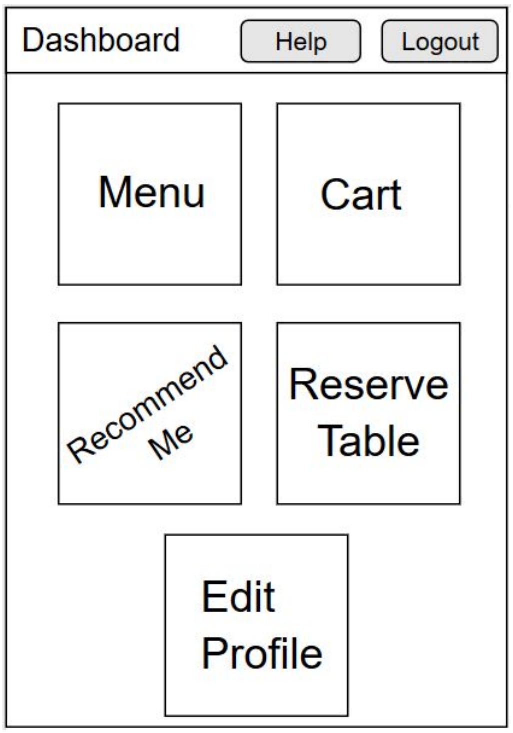
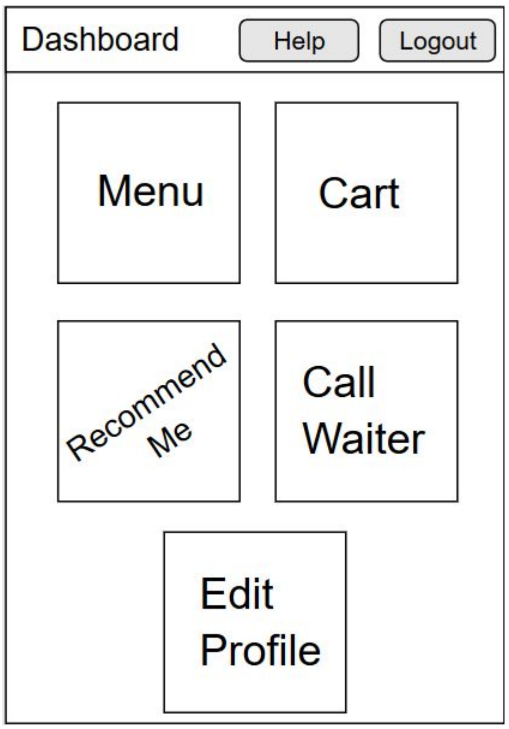
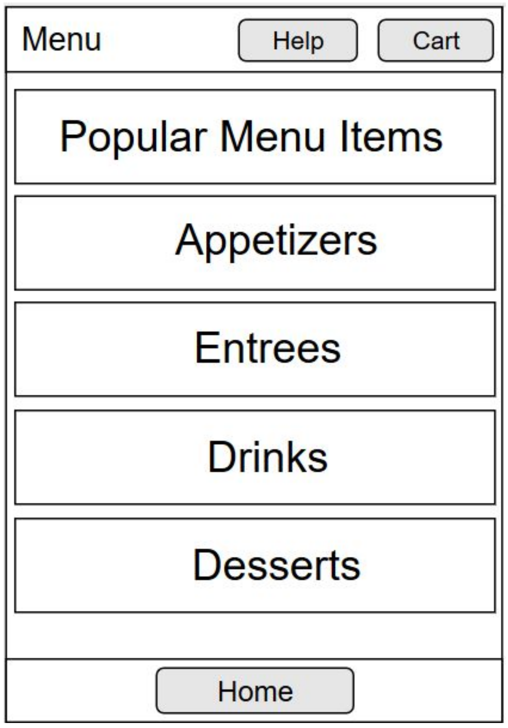
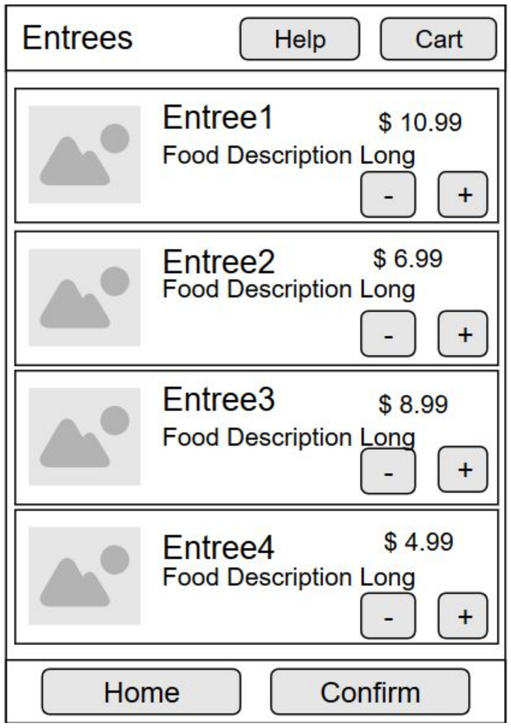


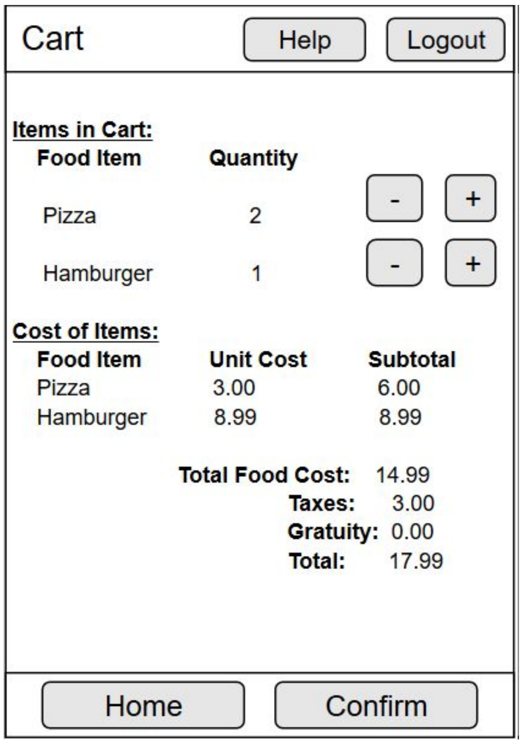
Table App Dashboard







The dashboard and menu designs are designed to reduce scrolling and swiping between pages. Users can easily select what appetizers/entrees they want and can add them directly to their cart from the menu pages.



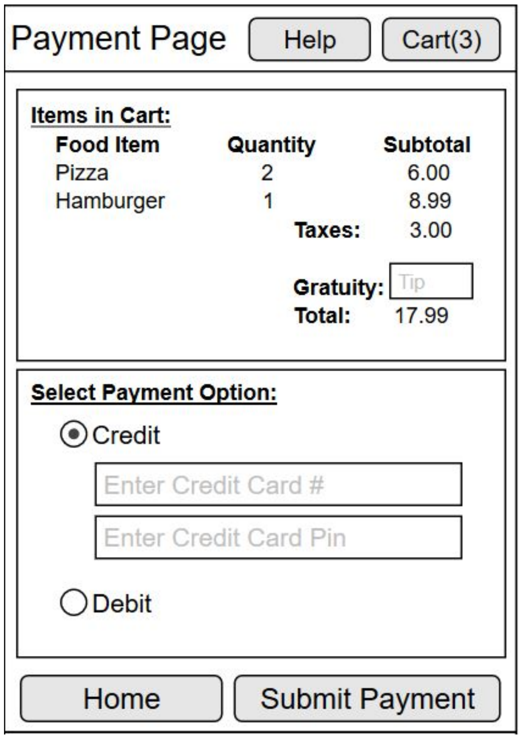
Items in Cart:

Food Item	Quantity
Pizza	2
Hamburger	1

Cost of Items:

Food Item	Unit Cost	Subtotal
Pizza	3.00	6.00
Hamburger	8.99	8.99

Total Food Cost:	14.99
Taxes:	3.00
Gratuity:	0.00
Total:	17.99

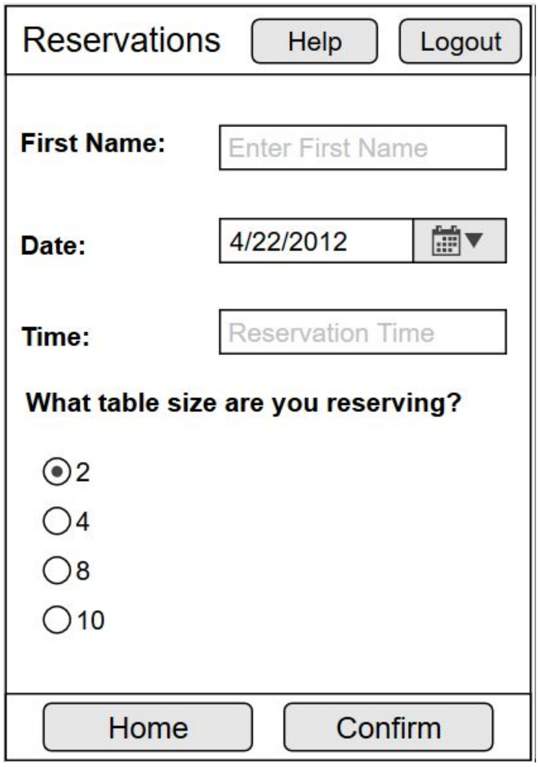


Items in Cart:

Food Item	Quantity	Subtotal
Pizza	2	6.00
Hamburger	1	8.99
	Taxes:	3.00
	Gratuity:	
	Total:	17.99

Select Payment Option:

[✓ Credit] Debit



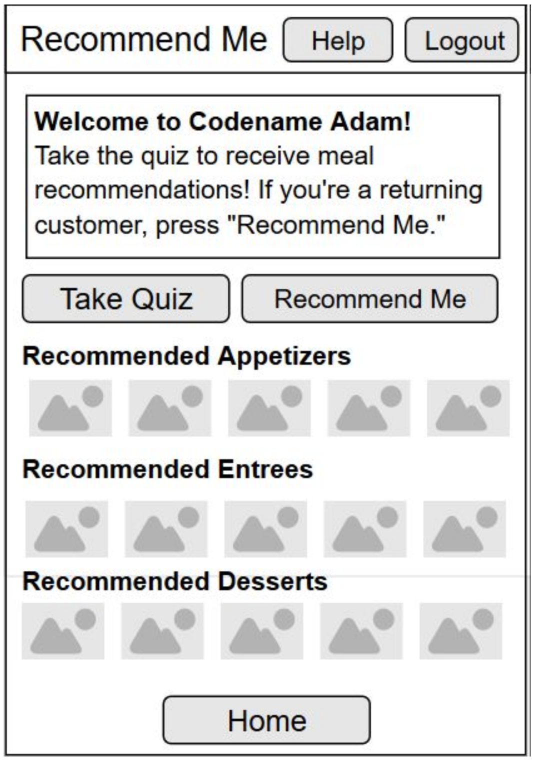
First Name:

Date: 4/22/2012

Time:

What table size are you reserving?

[✓ 2] 4 8 10



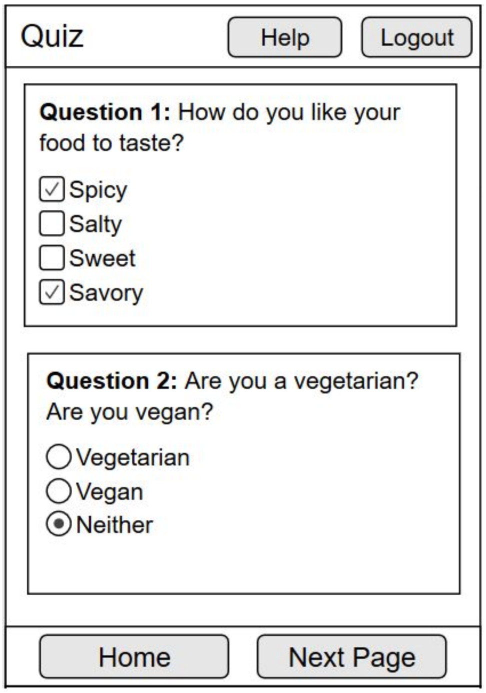
Welcome to Codename Adam!
Take the quiz to receive meal recommendations! If you're a returning customer, press "Recommend Me."

Recommended Appetizers

Recommended Entrees

Recommended Desserts

Users can edit their quantity of their meal items directly from the cart. Moreover, the reservation and recommendation pages are intuitive. The number of icons and words are minimized to improve user experience. Pressing on a recommended item will direct them to that item's menu description.



Question 1: How do you like your food to taste?

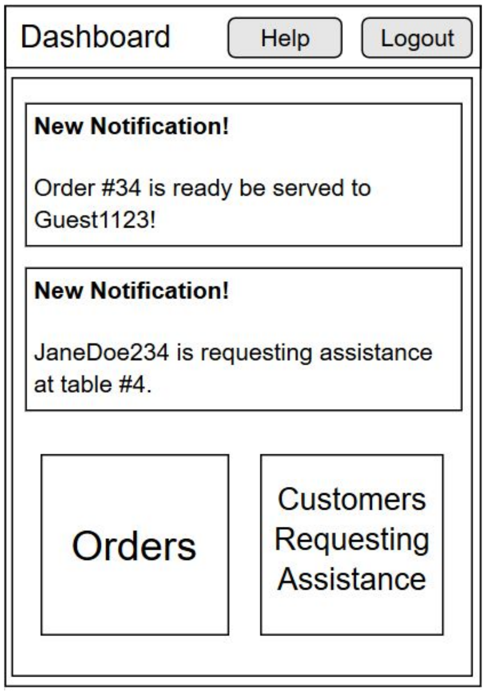
- [✓] Spicy
- [ ] Salty
- [ ] Sweet
- [✓] Savory

Question 2: Are you a vegetarian? Are you vegan?

- ○ Vegetarian
- ○ Vegan
- ● Neither

To make the quiz interface easy to use, the user will receive a few quiz questions per page and will keep on clicking "next page" until they finish the quiz. Afterwards, they will receive their recommendations on the initial recommendation screen.

5.3: Waiter App



Dashboard

Help | Logout

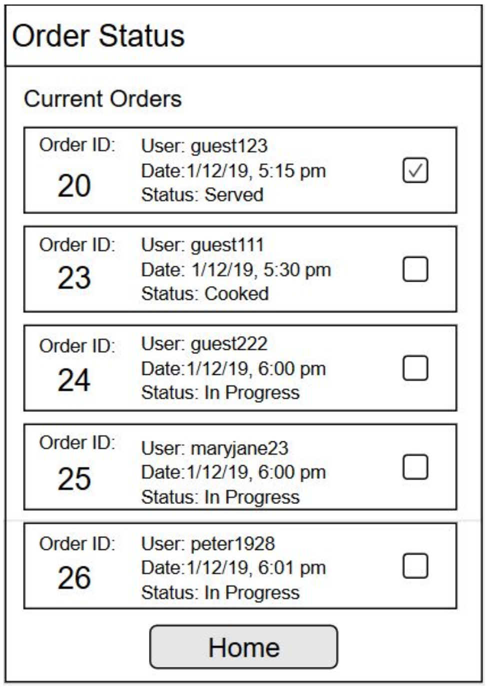
New Notification!

Order #34 is ready be served to Guest1123!

New Notification!

JaneDoe234 is requesting assistance at table #4.

Orders | Customers Requesting Assistance



Order Status

Current Orders

Order ID: 20 — User: guest123, Date:1/12/19, 5:15 pm, Status: Served ✓

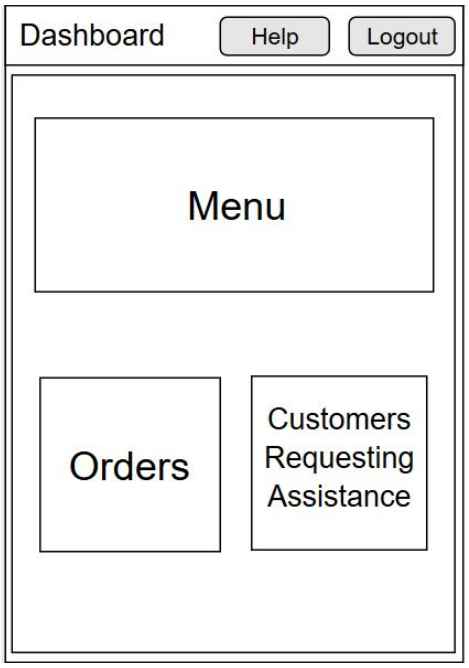
Order ID: 23 — User: guest111, Date: 1/12/19, 5:30 pm, Status: Cooked ☐

Order ID: 24 — User: guest222, Date:1/12/19, 6:00 pm, Status: In Progress ☐

Order ID: 25 — User: maryjane23, Date:1/12/19, 6:00 pm, Status: In Progress ☐

Order ID: 26 — User: peter1928, Date:1/12/19, 6:01 pm, Status: In Progress ☐

Home

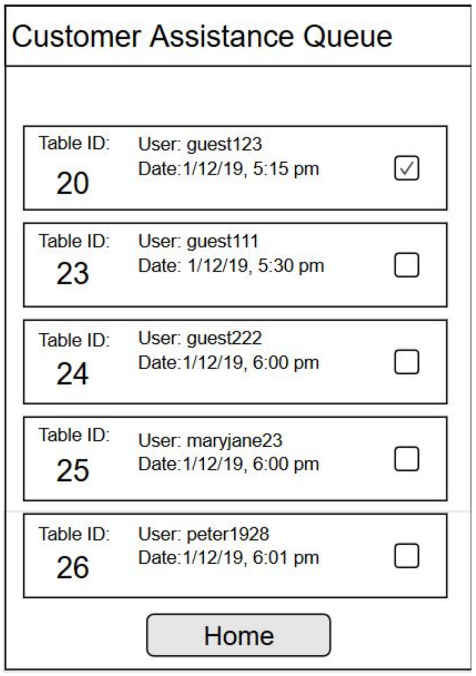


Dashboard

Help | Logout

Menu

Orders | Customers Requesting Assistance



Customer Assistance Queue

Table ID: 20 — User: guest123, Date:1/12/19, 5:15 pm ✓

Table ID: 23 — User: guest111, Date: 1/12/19, 5:30 pm ☐

Table ID: 24 — User: guest222, Date:1/12/19, 6:00 pm ☐

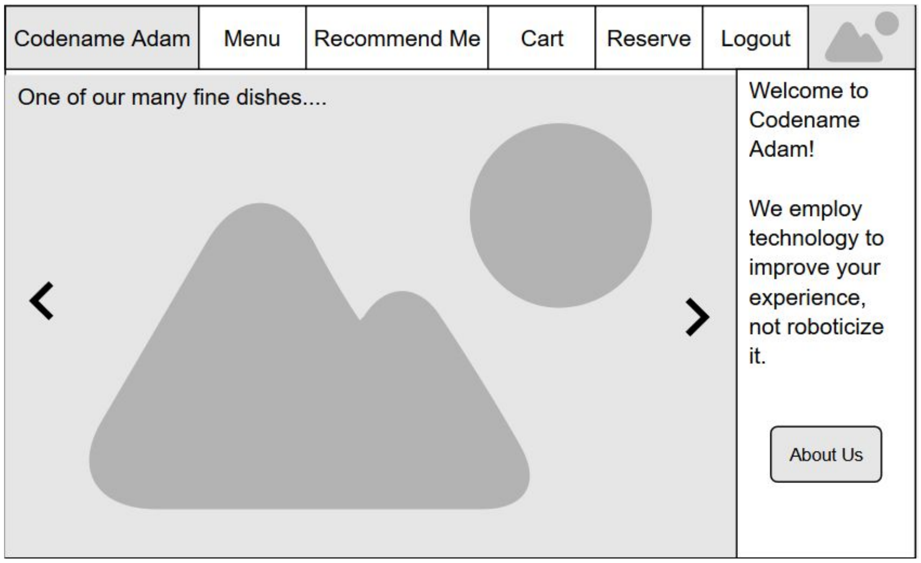
Table ID: 25 — User: maryjane23, Date:1/12/19, 6:00 pm ☐

Table ID: 26 — User: peter1928, Date:1/12/19, 6:01 pm ☐

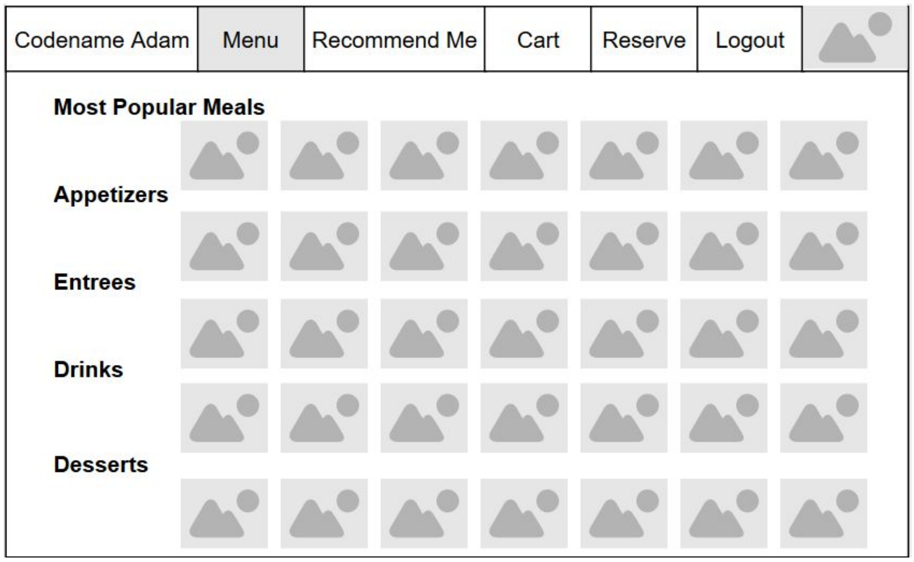
Home

The Waiter app will have 3 screens since waiters cannot afford to spend too much time looking at their phones. They will have access to the menu to answer customers' questions, the list of orders to keep track of what orders need to be served and have access to a list of customers that requested assistance. They can update order status and customer assistance request statuses by simply clicking on the appropriate checkboxes.

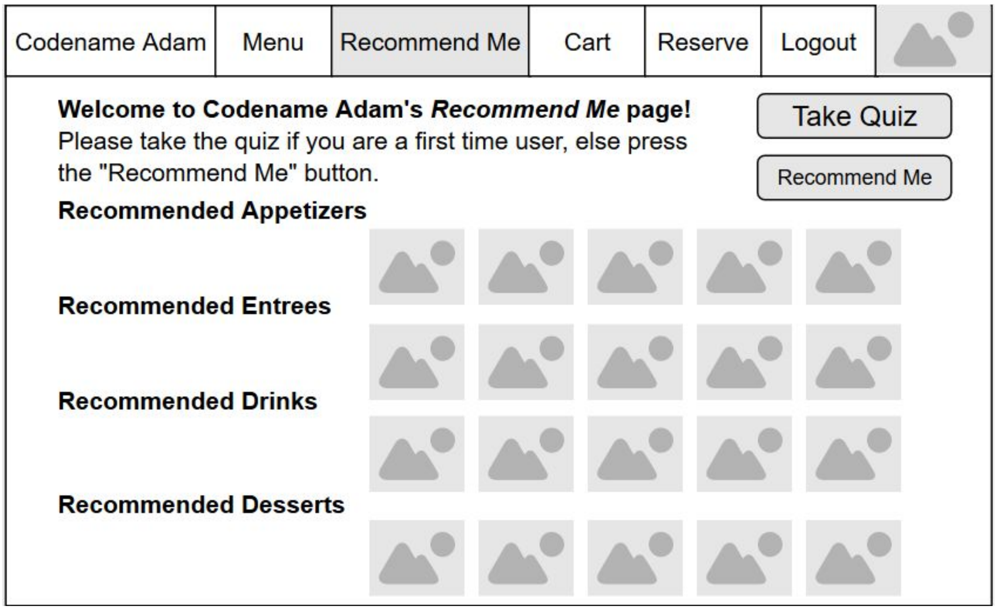
5.4: Restaurant Website


Codename Adam
Menu
Recommend Me
Cart
Reserve
Logout
One of our many fine dishes....
Welcome to Codename Adam!
We employ technology to improve your experience, not roboticize it.
About Us

The mockup for the restaurant website homepage is based on a static page that we have implemented already. The UI is designed to appeal to customers while being informative.



To make the menu page appealing, food will be represented with pictures. The user can simply scroll over each menu item and get more information through a pop-up. To get access to more menu items, horizontal scrolling functionality will be implemented.



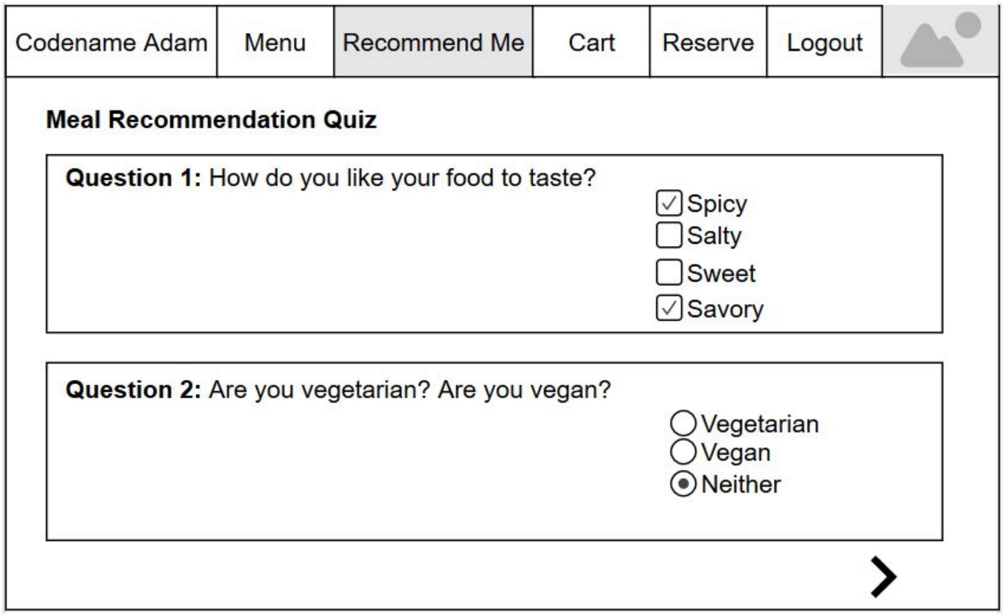
Welcome to Codename Adam's *Recommend Me* page!
Please take the quiz if you are a first time user, else press the "Recommend Me" button.

Recommended Appetizers

Recommended Entrees

Recommended Drinks

Recommended Desserts

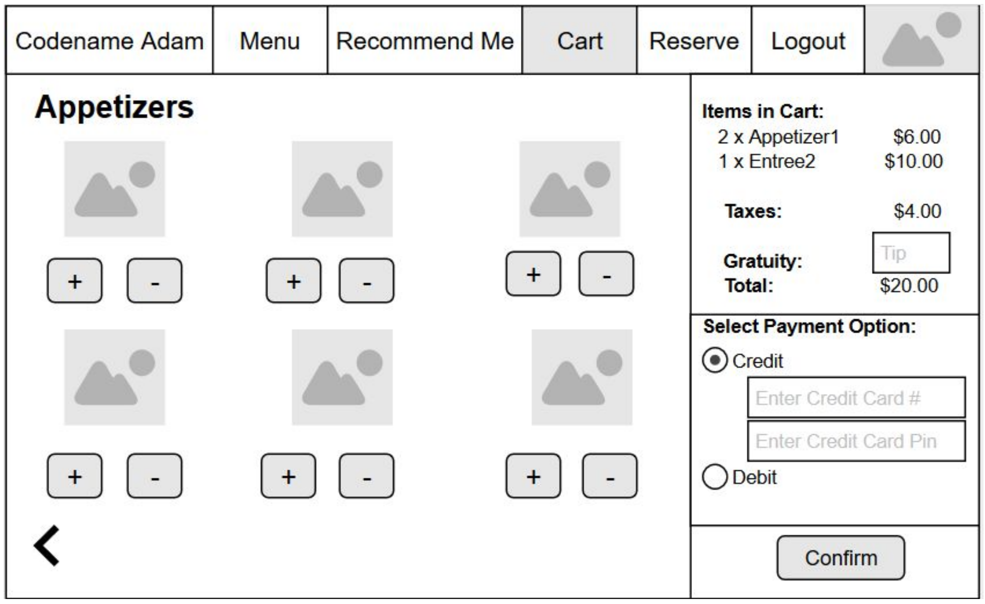


Meal Recommendation Quiz

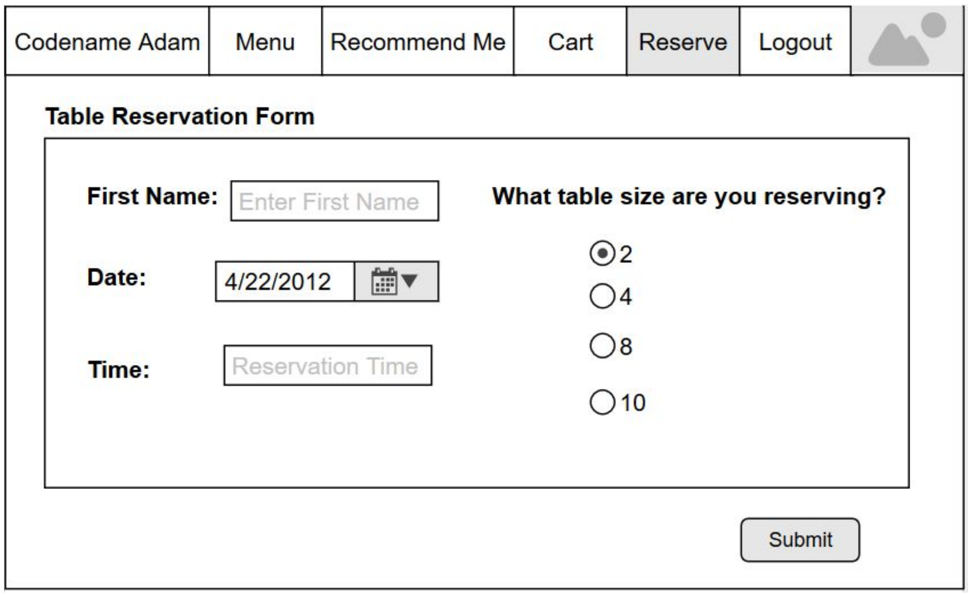
Question 1: How do you like your food to taste?
[✓ Spicy]
Salty
Sweet
[✓ Savory]

Question 2: Are you vegetarian? Are you vegan?
Vegetarian
Vegan
[✓ Neither]

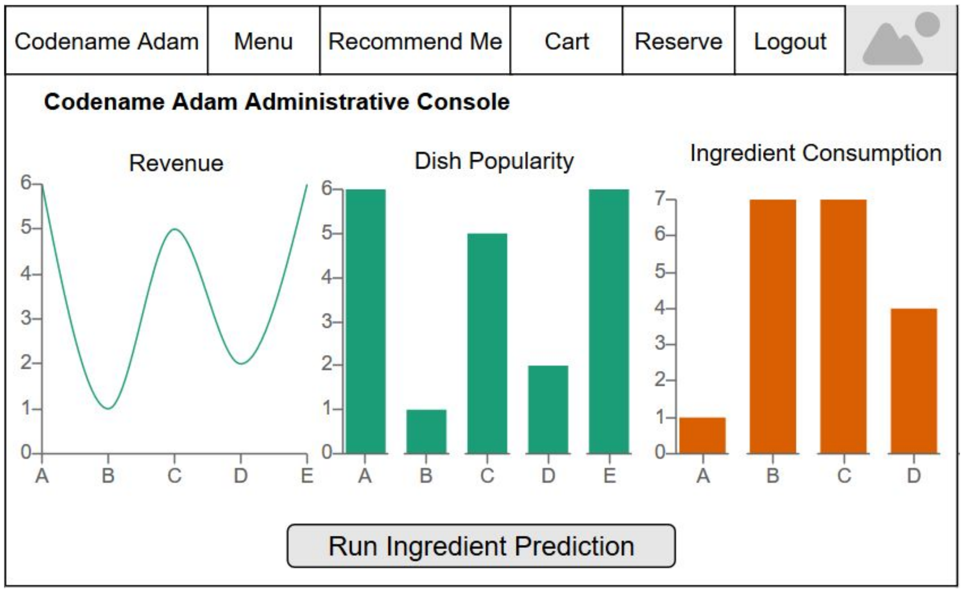
The recommendation page will follow a design similar to that of the menu page, in order to make it appealing to the user. Clicking on a recommended item will direct user to the cart page. Moreover, the Meal Recommendation Quiz screen designed is based on the one in the Table/Customer Apps and is designed to be intuitive.



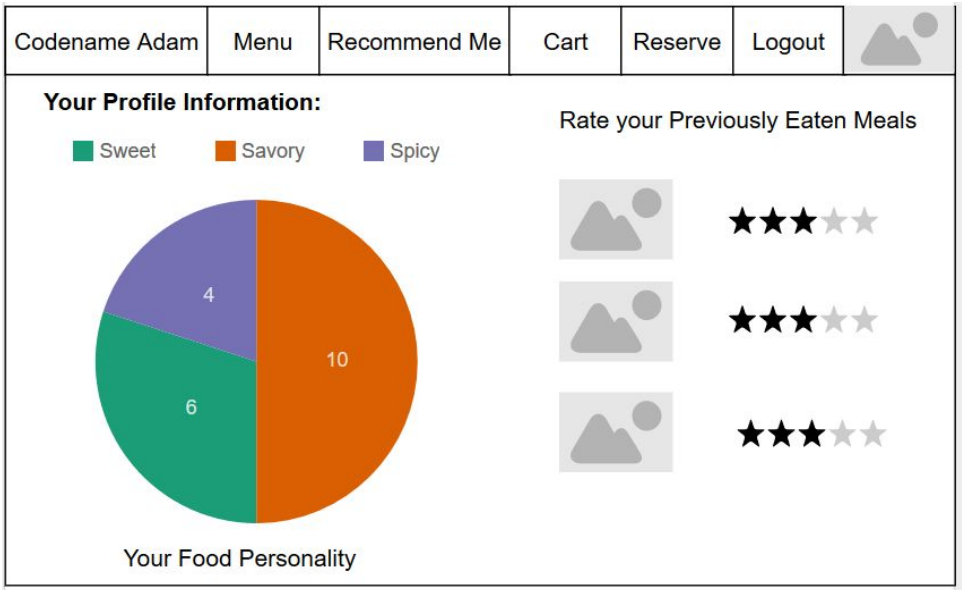
By clicking on the bolded categories on the menu page, users will be directed to the cart. This way they can directly add menu items to their carts. By pressing on the back button, they will be redirected to the menu screen. The cart functionality can also be accessed by pressing on the cart button on the navbar. The current structure makes it easy to edit cart.



The table reservation form is based on the form developed for the customer app. It contains few questions to improve user experience and a reservation ID is sent to the user (have to be logged in to use this functionality) once reservation is confirmed.



The administrative console (accessed when admin selects their user icon on the navbar), will contain all of the information a restaurant owner needs to know. It also allows them to make use of the ingredient prediction functionality. The admin can edit the menu with their privileges. An edit option will be available once they click 'menu.'



A regular customer can access their profile information by clicking on their user icon. They can view their food personality, as determined by their food group probabilities, and can also rate previously eaten meals. The rating functionality is easier to use on the website interface, so it will implemented for the website only.

Section 6: Design of Tests

6.1: Unit Tests

On the server side, since we are using NodeJS as our back-end, we will be 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 will have two 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.

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);
        }
}
```

6.2: 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.

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

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 7: Project Management

7.1: Merging Contributions from Individual Team Members

Coordinating meetings with 9 different people is already difficult, and the difficulty of compiling this report was exacerbated by the fact that some people had to leave early for their Spring Break vacations. We tried to address this difficulty by having meetings immediately after class to ensure that majority of our members can attend. During these meetings we discussed how we can keep track of progress towards the reports and our project. We decided to use Asana as a good way of keeping track of whatever tasks need to be done, which has helped us complete this report greatly

7.2: Project Coordination and Progress Report

We have set up some static website pages and some static pages for the Chef App. However, the bulk of our work has been done towards designing the API for our project. Considering how our project is an ecosystem of Android applications and a website, a well-defined API is essential to consistently transmitting information between different devices. We have defined our API with the help of NodeJS along with defining the overall structure of our project. The Android apps will parse information received from HTTP requests sent/received from the web application server. Moving forward, we can further define how the information can be presented in a dynamic manner on the websites and applications.

7.3: Plan of Work

Some milestones going forward include the following...

Milestone	**Deadline**
Finalize Website Pages (front-end)	March 18
Write a python script that creates fake data for our MongoDB database tables	March 19

Implement Content-Based Filtering for the Recommendation System	March 20
Finalize Recommendation Quiz	March 20
Chef App - “mark order as complete” functionality	March 24
Authenticate API endpoints	March 25
Waiter App - “request assistance” functionality	March 24
Implement Reservation Functionality for Website	April 22
All Android applications demonstrate all desired functionalities	April 22
Develop Ingredient Prediction System Backend	April 22
Implement Collaborative Filtering for the Recommendation System	April 22

7.4: Breakdown of Responsibilities

#	**Feature Subteams**	**Names**
1	Mobile Applications	● Taras Tysovskyi ● Lieyang Chen ● Hongpeng Zhang
2	Restaurant Website	● Arushi Tandon ● Yuwei Jin ● Seerat Aziz ● Alex Gu
3	Database and API Design	● Yuwei Jin ● Chris Lombardi ● Taras Tysovskyi
4	Recommendation and Review Systems	● Chris Gordon ● Seerat Aziz
5	Ingredient Prediction and Reservation Systems	● Alex Gu ● Chris Lombardi

Prior to Demo 1 (over Spring Break), Taras and Lieyang will be working on the Android applications and API. Seerat will be working on developing the website and the recommendation system.

Taras and Lieyang will work on developing the following classes for the Chef and Waiter applications:

- Order
- Controller
- MenuItem
- User
- Table

Seerat will work on developing the following classes for the website:

- Order
- Controller
- MenuItem
- User
- Recommendation
- MealRecommender

After Alex and Chris Lombardi come back from their Spring Break vacations, they will be working on the Ingredient Prediction System. We will catch up Chris Gordon, Yuwei Jin, Hongpeng Zhang and Arushi Tandon on all of the development that occurred during break.

Prior to the Demo 1, Seerat, Alex (who will be back the weekend before school resumes), Taras, and Lieyang will meet to integrate all of the code written over break. Seerat and Alex will perform integration testing for the website while Taras and Lieyang will perform integration testing for the applications.

Section 8: References

1. Professor Marsic's Lecture 10 Notes (Object Oriented Design II)
2. UML tool: https://staruml.io