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

Restaurant Automation Codename Adam

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

Date Submitted: March 10, 2019

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

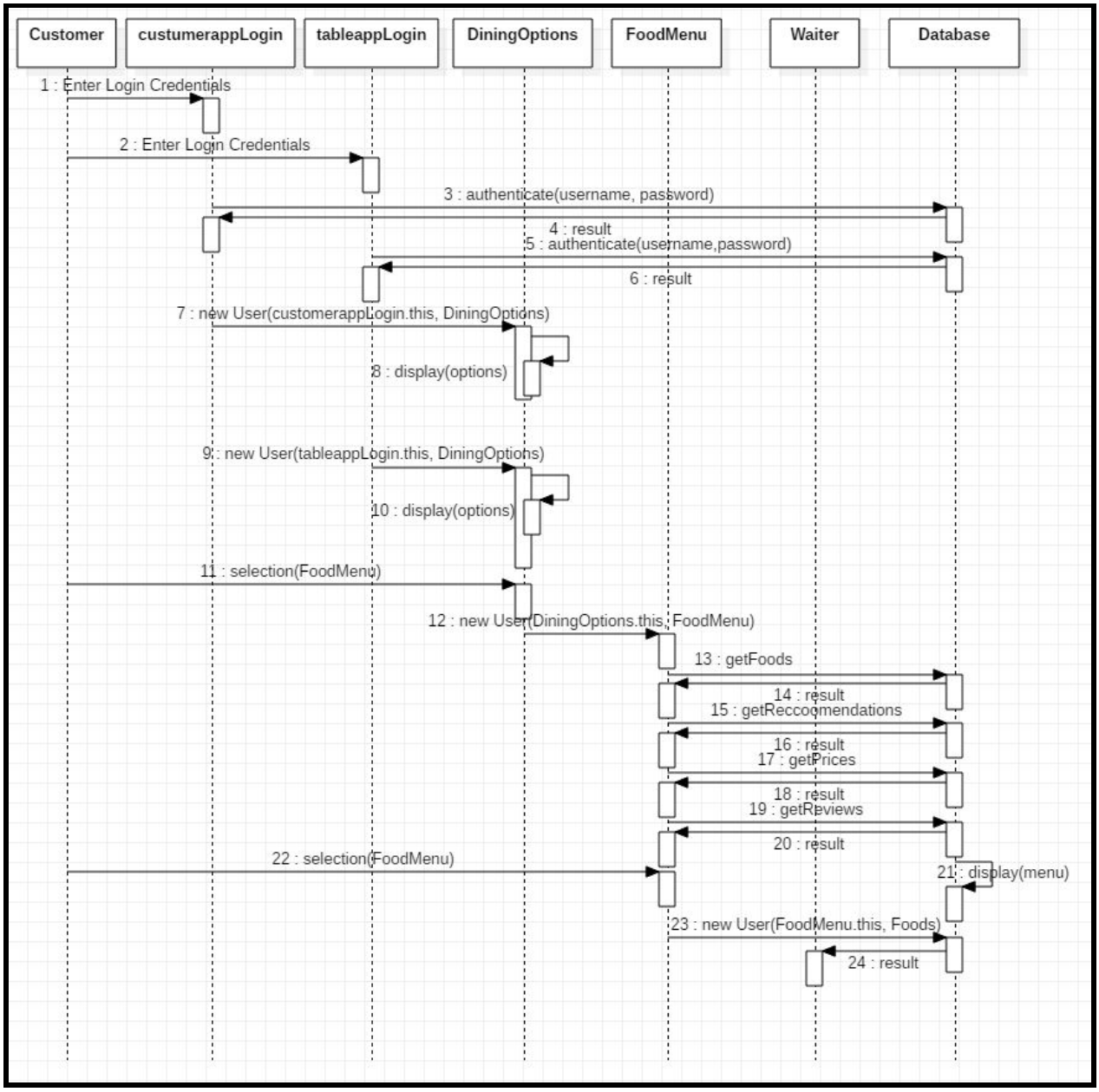
Table of Contents

Contributions Breakdown

Everyone contributed equally to this part of Report #2.

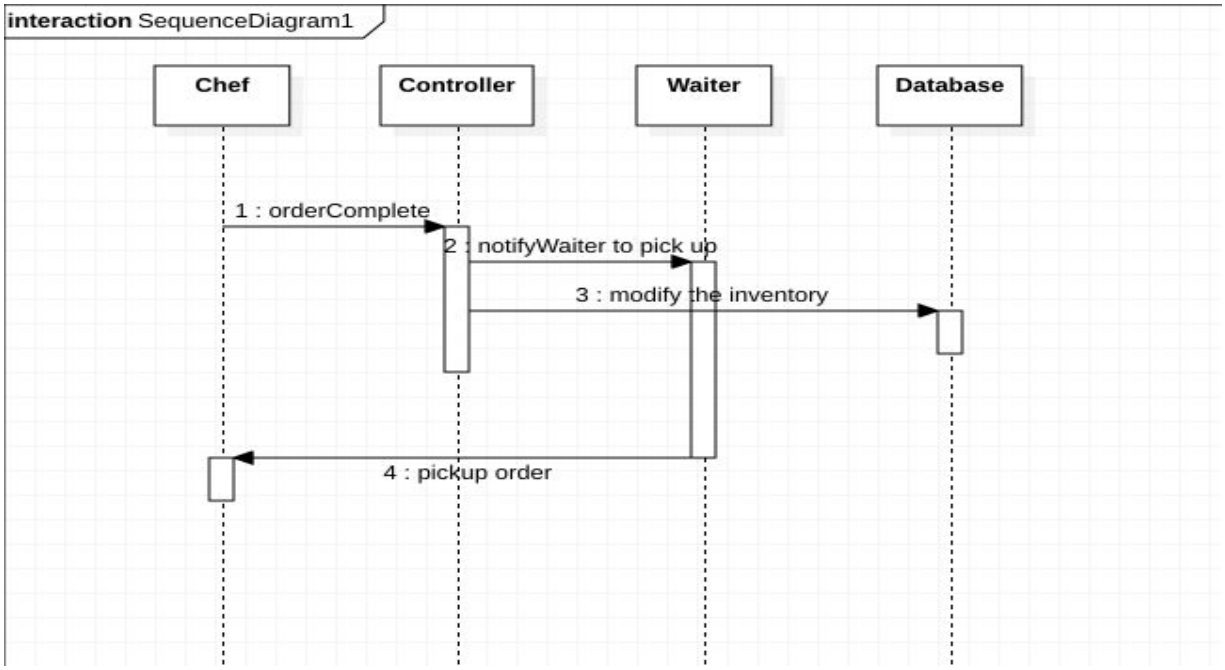
Section 1: Interactions Diagrams

1.1: UC-1 - Order Food



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.

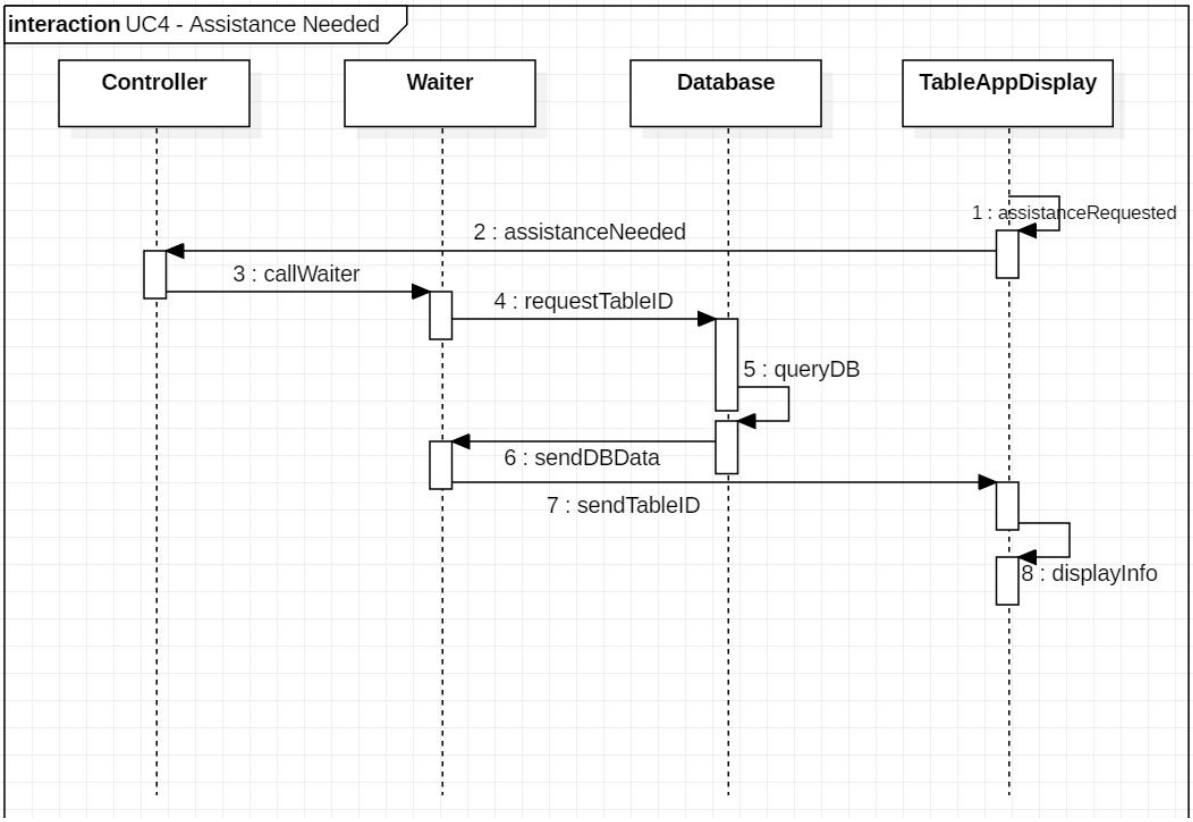
1.2: UC-3 - Order Complete



Design Principles Used:

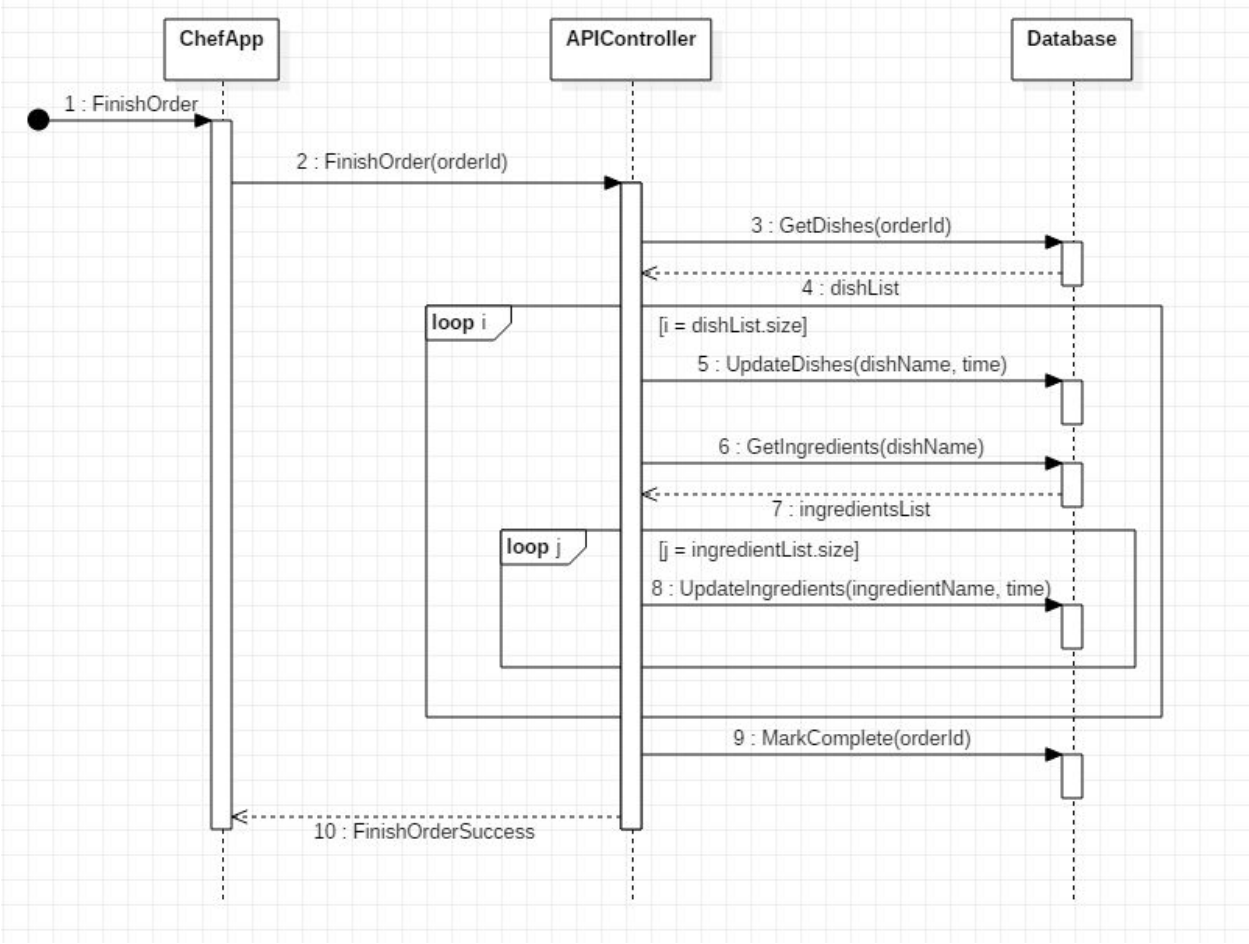
In line with the Dependency Inversion Principle, the Chef and the Waiter Apps do not communicate directly with the database, and instead have their requests routed through the Controller class. This also follows Interface Segregation Principle, because we only use four interfaces here. Therefore, minimum number of dependencies is achieved.

1.3: UC-4 - Assistance Needed



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).

1.4: UC-6 - Finish Order



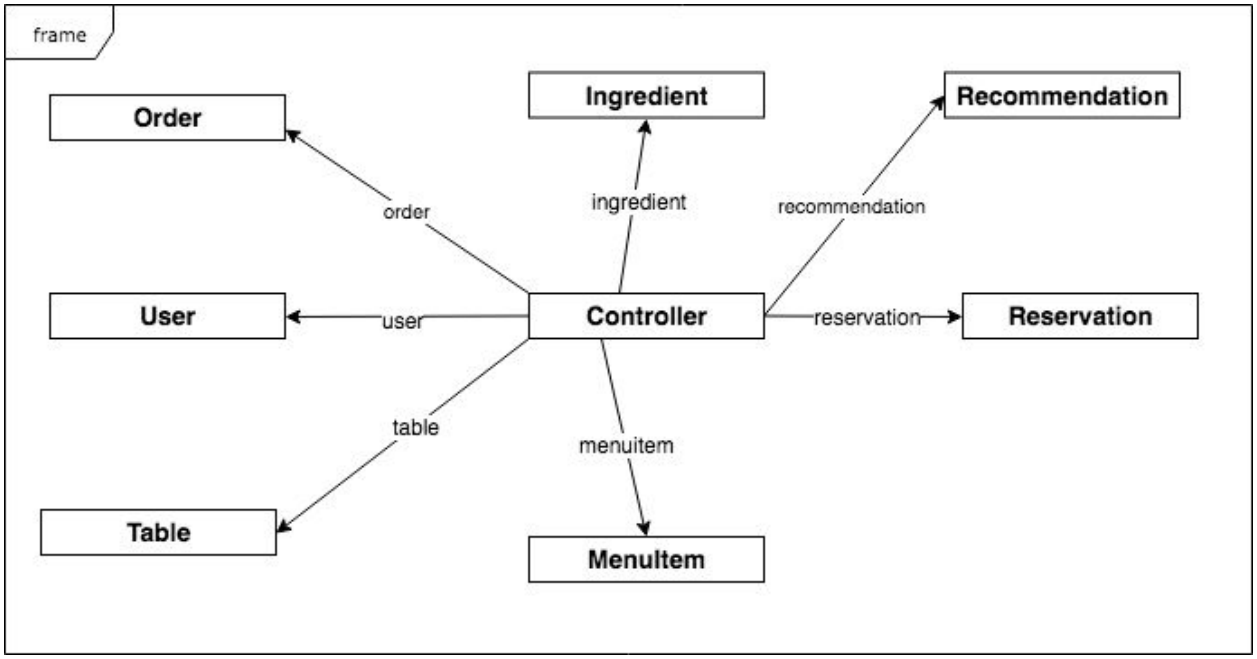
Design Principles Used:

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.

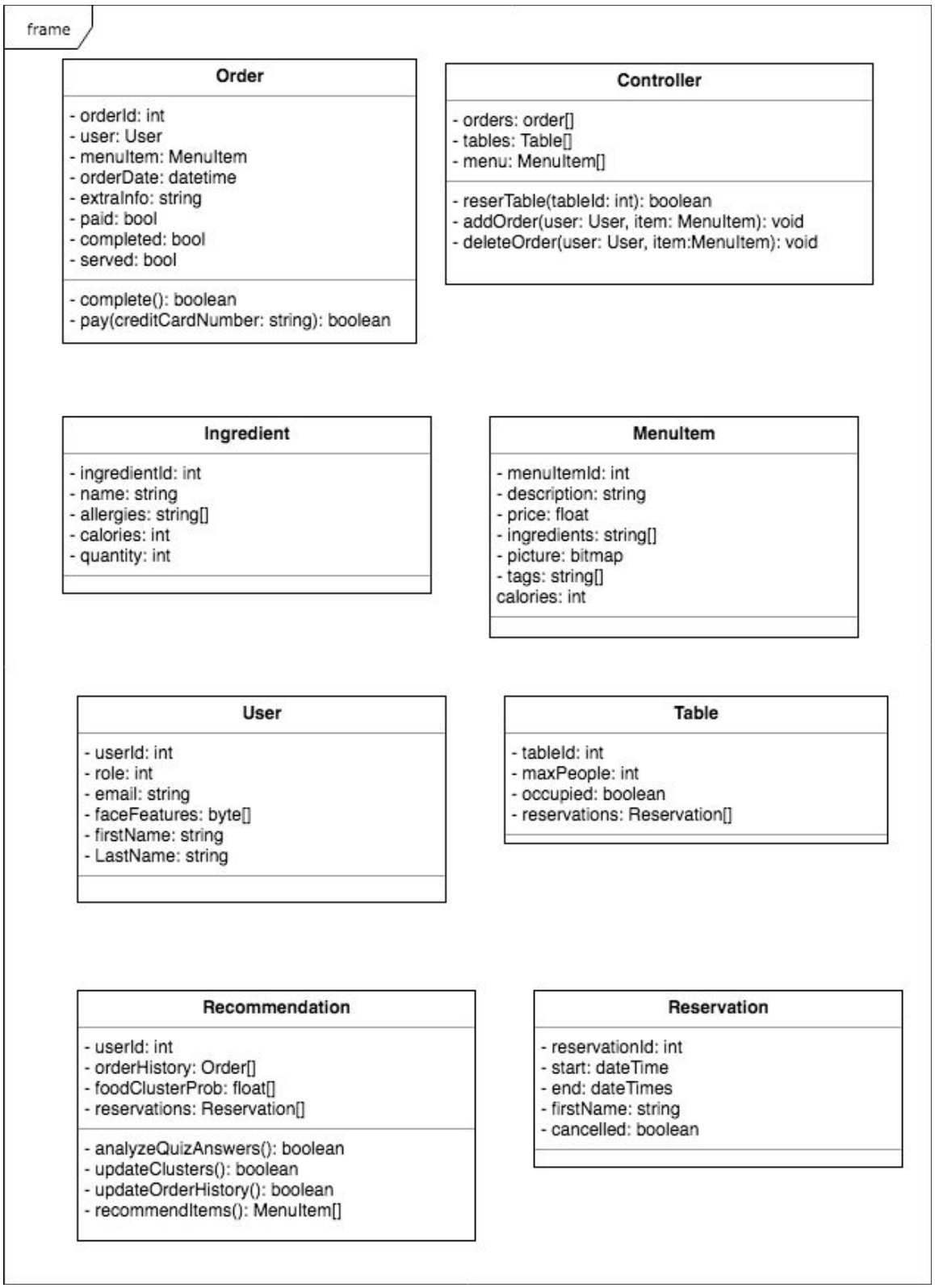
Following the Principle of Least Knowledge, or the Law of Demeter, the APIController has limited knowledge and must fetch all information about the orders, dishes, and ingredients from the database.

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:
Every action is made inside the controller. When a customer made an order, the controller will call addorder(). When a customer reserved a table, the controller will call reserveTable() and create a reservation object with its associated id and stored the customer info into it. When a customer read the menu, the controller will show recommendation by calling corresponding method. When an order is finished, the controller will call method to update corresponding ingredient object.

User: used to store user-related info: username, userId, role to determine which kind of user he or she is? (customer, server or cheif). If customer: whether s/he is served; his/her face feature captured by face recognition.

Ingredient: used to store ingredient-related info: its name, how many calories it has, whether it has allergies, how many quantities left.

Recommendation: by given information about a certain user's order history, this class is used to give suggestions when that user is reading the menu.

Table: used to determine whether the table is occupied and how many seats does the table have.

Reservation: everytime a customer made a reservation, a reservation object is created to hold the information of time of it, whether it is canceled or not.

MenuItem: used to contain all dished that the restaurant provide.

2.3: Traceability Matrix

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

*Recommendation and Reservation class titles abbreviated

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. Therefore, the Order and Controller classes had to be created.

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.

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.

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: Project Management

#	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

**Note that some developers are listed twice, we believe having one "expert" on two groups will help integrate the groups together. For example, Taras will integrate the Mobile app and the Database/Accounts together.

Weekly Meetings

We plan to meet at least once a week, either in person or online (through Google Hangouts) in order to keep everyone updated with each sub-team's progress.

Methods of Communication

We have a GroupMe and a Slack workspace environment for communicating with each other. The Slack workspace environment has channels for each subteam.

Shared Resources

Our group is using a Google Drive folder for keeping track of report work, since Google Drive facilitates collaboration. GitHub and Git will be used to keep track of progress of our software application.

Disaster Plan

We will enforce internal deadlines for project milestones as listed in the plan of work. If those deadlines are not met, then other group members will have to pitch in to ensure a milestone has been achieved.

Section 5: References

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