Report #3:

ChefBoyRD

Restaurant Automation

GROUP #6:

Richard Ahn
Zachary Blanco
Benjamin Chen
Jeffrey Huang
Jarod Morin
Seo Bo Shim
Brandon Smith

GITHUB & WEBSITE:

https://github.com/ZacBlanco/ChefBoyRD
http://blanco.io/ChefBoyRD

SUBMISSION DATE:

May 2, 2017

Individual Contributions Breakdown

	A	B	C	D	E	F	G	H	I	J
1	**Responsibility Matrix**									
2		**Team Member Name**								
3	**Task**	**Possible Points**	**Richard**	**Zachary**	**Benjamin**	**Jeffrey**	**Jarod**	**Seo Bo**	**Brandon**	**Completed Points**
4	Summary of Changes	5	17%	17%	0%	17%	17%	17%	17%	5
5	Sec.1: Customer Statement of Requirements	6	17%	17%	0%	17%	17%	17%	17%	6
6	Sec.2: Glossary of Terms	4	17%	17%	0%	17%	17%	17%	17%	4
7	Sec.3: System Requirements (User Stories)	6	17%	17%	0%	17%	17%	17%	17%	6
8	Sec 4: Functional Requirement Specifications	30	17%	17%	0%	17%	17%	17%	17%	30
9	Sec.5: Effort Estimation	4	17%	17%	0%	17%	17%	17%	17%	4
10	Sec.6: Domain Analysis	25	12%	12%	30%	12%	12%	12%	12%	25
11	Sec.7: Interaction Diagrams	40	17%	17%	0%	17%	17%	17%	17%	40
12	Sec.8: Class Diagram and Interface Specification	20	17%	17%	0%	17%	17%	17%	17%	20
13	Sec.9: System Architecture and System Design	15	17%	17%	0%	17%	17%	17%	17%	15
14	Sec.10: Algorithms and Data Structures	4	17%	17%	0%	17%	17%	17%	17%	4
15	Sec.11: User Interface Design and Implementation	11	17%	17%	0%	17%	17%	17%	17%	11
16	Sec.12: Design of Tests	12	17%	17%	0%	17%	17%	17%	17%	12
17	Sec 13: History of Work, Current Status, Future Work	5	17%	17%	0%	17%	17%	17%	17%	5
18	Sec 14: References	0	17%	17%	0%	17%	17%	17%	17%	0
19	PROJECT MANAGEMENT	13	17%	17%	0%	17%	17%	17%	17%	13
20	Total Points	200	32.1	32.1	7.5	32.1	32.1	32.1	32.1	200

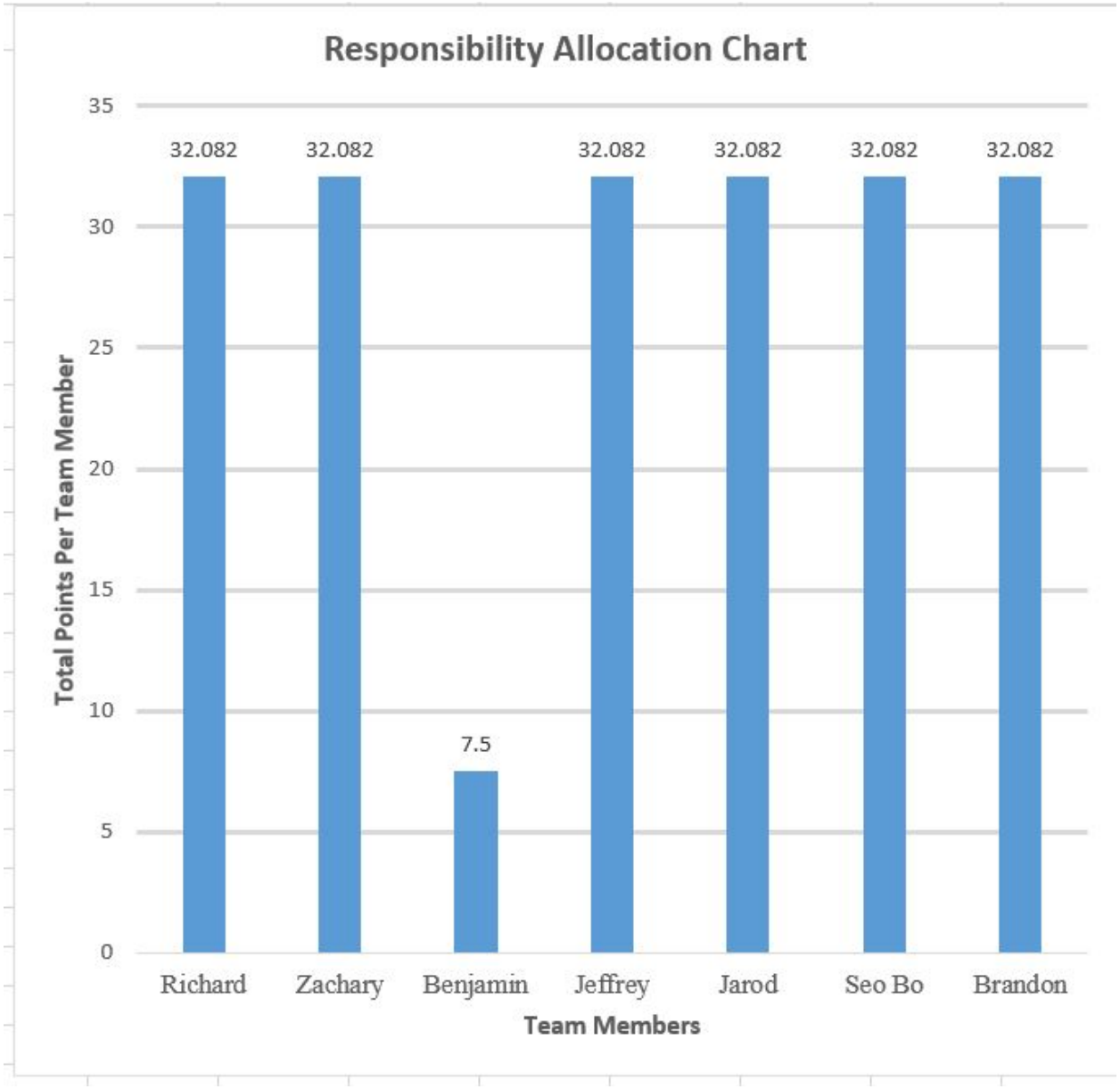

Responsibility Allocation Chart
35
30
25
20
15
10
5
0
Total Points Per Team Member
32.082
32.082
7.5
32.082
32.082
32.082
32.082
Richard
Zachary
Benjamin
Jeffrey
Jarod
Seo Bo
Brandon
Team Members

Table of Contents

Summary of Changes

- Updated <2. Glossary of Terms> with some additional terms. Ordered them alphabetically.
- Updated <4.Functional Requirements Specification>
    - <b. Actors and Goals> added additional actors for new use cases
    - <c. Use Cases> added: Gather Statistics use case. Added more extension cases for all Use Cases. Revised use case diagram. Updated traceability matrix to match.
        - Removed Table Survey UC b.c. not fully implemented. FB-UC1 (table survey) removed, FB-UC2 renamed to FB-UC1. Additional feature added to FB-UC1
    - <d. System Sequence Diagrams> added detail, included more description on alternate flow of events
- Added <5. Effort Estimation> using Use Case Points
- Added <5. Effort Estimation> Use case weight and actor weight for Alexa
- Update <6. Domain Analysis> to include new use cases, and comprehensive domain coverage. Updated System Contracts to include updates from System Sequence Diagram
- Updated <7. Interaction Diagrams> to include new domain names in sequence diagrams
- Added <7.a. Design Patterns>
- Revised< 8. Class Diagram and Interface Specification> to include names of the exact methods and attributes used in the project. Also updated with OCL Contracts
- Added<9.d Persistent Data Storage>
- Updated <9.e. Network Protocol> with additional description of HTML Protocol. Added some examples of how they're using in our software
- Updated <10.a. Algorithms> with revised implementation of feedback analysis. Included more detail and relevant examples.
- Updated <10.b. Data Structures> with detailed description of data structure usage in all features. Included more specific examples for each of the modules included within the project.
- <13. History of Work, Current Status, and Future Work>
    - Updated history of work
    - Added completed functionalities to current status
    - Added features for future work

1. Customer Statements of Requirements

a. Problem Statement

As an established restaurant, we wish to incorporate more technology to help run the restaurant more smoothly and to help us compete in the saturated restaurant market. Our understanding is that with the help of technology we are able to track information on large scales. The broad problem that restaurants like us face is creating meaningful contextual relationships using this information, which can then be analyzed to improve the restaurant overall. We would like to have the proposed software application address some specific problems. The problems we would like to address are improving the customer experience and increasing the efficiency of running the restaurant and bringing more customers, which ultimately leads to the financial success of the restaurant.

PREDICTING INGREDIENT USAGE

Chef:

As a chef, one of my responsibilities is gauging the amount of food that needs to be prepared for a typical day. This plays a crucial role in the day-to-day operation of a typical food business. As the restaurant gets busier throughout the day, our kitchen has to predict which dishes will be ordered and begin preparing them in advance to keep up with customer demand. However, if we predict incorrectly, we risk preparing too many dishes and wasting both time and money with the wasted dishes. According to the NRDC, approximately 40% of food gets wasted, and a major contributor to that number is restaurants.[1] Our kitchen can attempt to prepare fewer dishes, but if we get too much business and have not made enough dishes, our customers will have to wait longer to receive their orders. This makes customers unhappy and discourages them from returning to our restaurant in the future.

With prior experience, it is possible to predict (to a certain degree) which dishes and how much food should be prepared from day to day. However, this method is very prone to error for even experienced chefs. A system that can analyze our restaurant's metrics and order histories would be extremely useful, as it will use this data to determine the amount of food we should prepare for a given day. This would not only reduce our restaurant's waste but also increase our profit margins. This allows us to invest more into our business and make our customers and employees happier.

As a chef, if I am constantly busy around the kitchen and preparing food, we would like this predictive system to not only be available within the software application; but also available to use and communicate with through a voice-activated service such as Amazon Alexa. Being able to utilize this feature through Amazon Alexa's voice services means I will be able to work in

the kitchen while receiving updates and predictions from the service on the day's food production. It should be able to provide the same type of estimates or similar to the ones available by the software application.

Manager:

We would like to reduce waste and excessive use of ingredients, and what would help is we can view exactly how much of each individual ingredients were used over different time spans. With the aid of this system, chefs can more accurately estimate the ingredients needed on a particular day, expediting the food preparation process and minimizing the waiting time for customers to receive their orders. We would like the application to give us detailed analyses and reports about the restaurant's order history over the course of days, weeks, months or years. We would like the application to display order history information in an easy-to-view format, such as dashboards with graphs and charts [Fig 1a-1]. This system is also helpful for me as a manager so that I can understand the day-to-day performance and long-term trends of the restaurant. The ability to see this data helps me determine overall business trajectory and make informed decisions to keep the restaurant financially successful.

[Fig 1a-1] Administrative Panel

We would also like the application to use the accumulated data to give the chef an estimate - an actual number - of the type and quantity of specific ingredients that are required throughout a typical day. The estimates should be able to tell how much of certain ingredients will be consumed within certain hours of business operation. Ultimately, the quantities for food preparation will be determined by the chef, but the system’s estimate should serve as a general benchmark from which the chef can base his decisions.

Overall, this should result in a more efficient use of food resources, especially with time-sensitive ingredients. It can also allow us to prepare food in advance with more time-fragile, but fresher, ingredients.

We do however recognize that one of the drawbacks of using the predictive software is the lack of reliability in its initial stage due to insufficient data. Therefore, we would like the system to provide a simple interface to allow us to sync our historical records from our existing POS system to the new one. This way, we can quickly get the prediction software running as best as possible. This preliminary step of obtaining data is extremely important because a predictive software that is unreliable for the first few weeks or even months is highly undesirable.

Overall, this system can help us serve our customers faster by telling us how much food to prepare at a given time. This system will also allow our business to waste fewer ingredients, resulting in greater profits for our restaurant. Our managers will also be provided with a clear and simple visualization of their restaurant's financial status to make their own analyses and business decisions. Eventually, the profits from this system can be invested back into our business to provide higher quality ingredients, higher employee salaries, or even lower prices for customers.

CUSTOMER FEEDBACK

Waiter:

Customer feedback is very important in how I'm presenting myself to my customers. Some nights I receive awful tips, or no tips at all and I have no idea why this is the case. Though every customer is different, if I had some concrete feedback to work with, I could improve my customer service skills or communicate any problems to the management. Different customers use different signals to get my attention when they want something. Usually people will wave me over or call out when I pass. Sometimes when the customer thinks that I am being inattentive or am ignoring them, it is just that I didn't recognize that the signal they were using indicated that they wanted my attention. If people communicated more details about their expectations and the signals they used instead of simply rating the service as excellent or poor, it would allow me to learn these new signals and prevent me from repeating my mistakes.

Chef:

When I test new dishes, it would be very helpful to know if people like them. It's never easy to find out what exactly the customers are thinking, but if people are leaving unhappy or are absolutely raving about my dishes, they usually make their opinions clear. My waiters and the manager already do a good job of communicating what the customers think about my dishes, but it would be nice to have something concrete to work with.

Customer:

The waiter usually asks us to complete an online survey about our experience when we receive the check. The website is written on the receipt, and there are usually prizes, but I've never actually completed one. Entering the website on mobile is cumbersome so I prefer to respond on a home computer. However, by the time I've gotten home I've usually forgotten about the receipt. Sometimes going to a restaurant isn't the last thing I do while I am out and I don't remember the details about my dining experience by when I return home I would be happy to offer my opinion if the restaurant provided a solution so that customers could take the survey before leaving the restaurant.

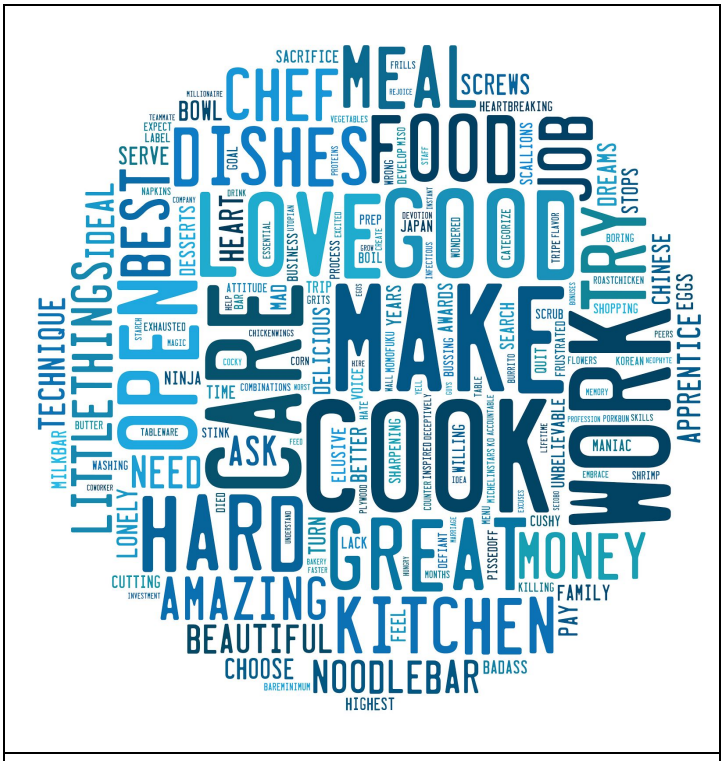
Manager:

A critical task that I have to perform as a manager is finding the source of problems that I see. If the restaurant is only half-full on a Friday night, I have to find the cause behind the problem. Could it be an employee? Is there something wrong in the kitchen? This is where customer feedback is useful, where I can learn the perspective of the customers and act accordingly to find solutions. In fact, it would be even more helpful to have contextual information about a specific feedback. Who was on shift during this time? What did they order?

I need to have improved methods of collecting feedback, because the current methods do not suffice. Verbal feedback is easy to understand, but difficult to quantify. Online surveys require patrons to visit a website and submit a specific form that restricts users entry. Though they are detailed, they are unpopular because they are so long!

Our previous implementation was to use a suggestion box, but the results were underwhelming. Very few customers offered suggestions and the results were disorganized. We have switched to an online form that requires customers to create an account to participate. To compensate for the added inconvenience, we provide incentives to respond. These prize rewards are also advertised on receipts. We have received more feedback as a result, but the increase in responses does not justify the cost of the prizes.

I want to improve the quantity and quality of the feedback collected, perhaps make feedback submission convenient to the customer and easily trackable [Fig 1a-2]. Then once I can receive the feedback, I have to make business decisions based on them and eventually report them to the restaurant's owner. Tracking feedback helps me by providing evidence for my decision-making process. If I need to justify my managerial decisions to the owner, it would be extremely helpful to have supporting data, such as the customer feedback.



[Fig 1a-2] - Word Cloud

I also want to make sure that feedback is sincere and addresses a problem. I would like the ability to keep track of feedback and determine whether the feedback is useful or not.

If a customer is blatantly dissatisfied with a service, in my experience they usually communicate this directly to me or the owner. However, this situation can allow minor or subtle problems to exist undetected. What if customers have feedback that is not substantial enough to warrant calling me or a staff member over?

This could be something like: the table is wobbly, the light is too dim at times, the seats feel a little hard. Individually, these bits of feedback may not be important, but collectively they could make a difference. Some of these items may not be important in the big scheme of things, but making these subtle changes are key to providing an excellent customer experience that exceeds expectations.

TABLE MANAGEMENT

Host(ess):

As a hostess, one issue that I constantly deal with is seating customers. Each table seats a different number of people and different-sized groups of people arrive all at different times. Updating this is a tedious, error-prone process, especially during peak hours when the restaurant is busiest. I want to be able to quickly assign people to tables, and change the table locations according to how it looks in the restaurant [Fig 1a-3].

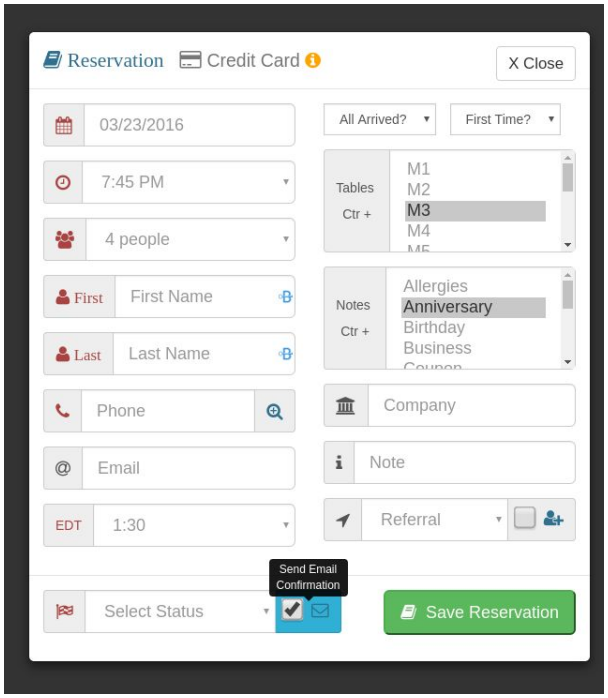
[Fig 1a-3] Table View

Customer:

Long wait times at restaurants are inconvenient and annoying to deal with. There are times I've had to wait outside because there were too many people lined up inside the restaurant.

But that's what I always reserve my spot at restaurants now. I usually call in, or make a reservation online. Even then, I do get a wait time estimate but I can wait in the comfort of my home and plan accordingly. But some of these online reservation systems are not convenient at all, I have to enter my full name, address, credit card number, when I just wanted to reserve a

spot [Fig 1a-4]. And there are times where after I reserve a spot and come into the restaurant, I find out that my spot wasn't reserved at all. It would be nice if I could be more confident that the online-reservation is working. A simple confirmation message with a generic message just doesn't do it for me.



[Fig 1a-4] Reservation Form

Also, the waiting times are not very accurate and can be inconvenient. I'll be told that I have to wait an hour, then when I arrive on time, I'm told I have to wait another 30 minutes! It would be nice to see where I am in line, and how many people are before me.

Overall, I just want to ensure that I'm not wasting my time when I decide to eat out at a restaurant. I'd like to have short, accurate waiting times, with minimal errors in how I'm being seated.

SHIFT MANAGEMENT

Manager:

As a manager, I have to maintain all of the logistics of the business. This often means that I have to handle tasks such as having repetitive conversations with employees about when they can come into work. In addition, there are times I have to call them up to see if they are available on specific days. I wish to put more responsibility in the hands of the employees.

The big problem is that I have to manage all this and tell employees their shift times [Fig 1a-5]. It is my responsibility to ensure the restaurant is staffed, but sometimes this can lead to ambiguity. For example I may forget to tell an employee they are scheduled for a specific time. Or they may claim that I did not notify them. Also if employees cover for each other, I'd like this to be clear to both parties and me. Overall, an organized system will reduce business liability.

Something like an online shift scheduling tool will be useful. So if shifts are changed, or if an employee has an emergency, I can notify all employees of an open shift. Internet access has become increasingly more common in the past decade, so it is now a more reasonable request to ask employees to check online for their shift times.

JOB POSITION	MON			TUES			WED			THURS			FRI			SAT			SUN		
	HRS	Cost ($)	HRS (#)	HRS	Cost ($)	HRS (#)	HRS	Cost ($)	HRS (#)	HRS	Cost ($)	HRS (#)	HRS	Cost ($)	HRS (#)	HRS	Cost ($)	HRS (#)	HRS	Cost ($)	HRS (#)
Server	9 - 5	17	8	9 - 5	17	8	9 - 5	17	8	9 - 5	17	8	9 - 5	17	8	9 - 5	17	8	9 - 5	17	8
Server	10 - 3	11	5	10 - 3	11	5	9 - 2	11	5	9 - 2	11	5	9 - 2	11	5	9 - 2	11	5	10 - 3	11	5
Server	10 - 3	11	5	10 - 3	11	5	10 - 3	11	5	10 - 3	11	5	10 - 3	11	5	10 - 3	11	5	10 - 3	11	5
Server	10 - 3	11	5	10 - 3	11	5	10 - 3	11	5	10 - 3	11	5	10 - 3	11	5	10 - 3	11	5	10 - 3	11	5
Server	11 - 8	19	9	11 - 8	19	9	10 - 3	11	5	10 - 3	19	5	10 - 3	11	5	10 - 3	11	5	11 - 8	19	9
Server				3 - 9	13	6	11 - 8	19	9	11 - 8	13	9	11 - 8	19	9	11 - 8	19	9	3 - 9	13	6
Server	4 - 10	13	6	4 - 10	13	6	3 - 9	13	6	3 - 9	13	6	3 - 9	13	6	3 - 9	13	6	4 - 10	13	6
Server	6 - 11	11	5	6 - 11	11	5	4 - 10	13	6	4 - 10	13	6	4 - 10	13	6	4 - 10	13	6	6 - 11	11	5
Server							6 - 11	11	5	6 - 11	11	5	4 - 11	15	7	4 - 11	15	7			
Server													6 - 11	11	5	6 - 11	11	5			
Grill	8 - 4	72	8	8 - 4	72	8	8 - 4	72	8	8 - 4	72	8	8 - 4	72	8	8 - 4	72	8	8 - 4	72	8
Fry	9 - 3	54	6	9 - 3	54	6	9 - 3	54	6	9 - 3	54	6	9 - 3	54	6	9 - 3	54	6	9 - 3	54	6
Grill	4 - 11	63	7	4 - 11	63	7	4 - 11	63	7	4 - 11	63	7	4 - 11	63	7	4 - 11	63	7	4 - 11	63	7
Fry	5 - 11	54	6	5 - 11	54	6	5 - 11	54	6	5 - 11	54	6	5 - 11	54	6	5 - 11	54	6	5 - 11	54	6
Dishes	8 - 4	48	8	8 - 4	48	8	8 - 4	48	8	8 - 4	48	8	8 - 4	48	8	8 - 4	48	8	8 - 4	48	8
Dishes	5 - 11	36	6	5 - 11	36	6	5 - 11	36	6	5 - 11	36	6	5 - 11	36	6	5 - 11	36	6	5 - 11	36	6
TOTALS:	MON	419	84	TUES	431	90	WED	442	95	THUR	457	95	FRI	457	102	SAT	457	102	SUN	431	90
																			Weekly Total:		$3,079
																			Hours:		659

[Fig 1a-5] Employee Schedule

An organized online system like this would also be useful because I can keep track of everyone's hours easily. I will be able to calculate payroll automatically. Typically the software

used for managing shifts and payroll are offline. This means I have to manually enter in shifts, which again is very repetitive.

Also, keeping a record of employee shifts as well as a record of orders will be useful in tracking when someone makes an error. If errors are being made continuously, either with the order or server, an organized record can help keep track of employee underperformance.

Employee:

I need a convenient way to be able to see my shifts. I may forget when I have a shift scheduled. I may need my shift covered for a non-emergency or emergency reason, and I may want to pick up more shifts in a week. It would be extremely convenient to have a website where I can access all this information and make changes. This means I can access this on my phone or desktop whenever I want to, instead of having to call my manager every time I want to make a change.

2. Glossary of Terms

Chefs: A critical member of the kitchen staff who directs kitchen activities including food preparation, cooking, and presentation. Also responsible for creating new recipes.

Customer: A person who uses, or intends to use the services and products of the restaurant.

Customer Satisfaction: A measurement of whether a patron's enjoyment or satisfaction of the restaurant experience meets or exceeds standards. This is impacted by quality of service, food quality, wait time, and overall experience.

Dish(es): A complete food item produced by the kitchen staff.

Employee Portal: Where individual employees can login and manage their shifts, contact information, non-role specific information.

Feedback: Information that signifies, to any level of detail, a positive, neutral, or negative response to products or services.

Food preparation: A process executed by the kitchen staff between the time the food ingredients are stored, and cooked.

Food waste: For our purposes, food waste will refer to prepared food that is thrown out because ingredients are not used and expire. This does not include food waste from uneaten or half-eaten food from customers.

General Employee: A term used to refer to any sort of employee working at the restaurant. This includes but is not exclusive to: waiters, hosts, cooks, manager, etc.

Host/hostess: restaurant employee in charge of greeting customers at the door, and seating arrangements in general

Ingredients: The edible components of a prepared dish before being cooked.

Logistics: related to the coordination of different employees of the restaurant in order to provide a desired business goal.

POS (Point-of-Sale): A system that a modern restaurant may use to manage its customer orders,process them to collect payment,and send the orders to the kitchen. These systems often include the cash registers and monitors that employees use to enter in orders.

Reservation: A customer declares and notifies the restaurant that they are arriving at a specified date and time. They usually must be made with customer's name, contact info, date/time, and number of guests.

Restaurant Portal: Where employees can access their role-specific functions in the software.

Server: This term will be used to refer to the computer hardware system that hosts the software. This does not refer to the waiter/waitress who is commonly called a server.

SMS: Short Message Service (SMS) is a text messaging service component of most telephone providers

Survey: Refers to a form that the customer submits to the restaurant to convey feedback. This could include an overall rating & detailed description.

3. System Requirements

a. User Stories

Higher size values indicate a longer expected implementation time.
Crossed out User Stories have not been implemented

Role: Manager (ST-MA)

Identifier	User Story	Size (1-10)
ST-MA-1	I do not need to manually enter orders or ingredient lists apart from the POS system.	7
ST-MA-2	I can pull up economic information (net gain, net loss, revenue, expenses, etc.) at ease.	6
ST-MA-3	I can see the inventory that we currently have to make sure that we have enough ingredients and/or materials to continue the quality of service.	5
~~ST-MA-4~~	~~I can receive alerts pertaining to abrupt critical information such as negative feedback, a negative revenue trend, etc~~	~~4~~
ST-MA-5	I can see the customer feedback on specific employees and dishes served.	8
ST-MA-6	I can view the detailed feedback that customers send in.	5
ST-MA-7	I can see customer feedback organized into positive/negative categories.	7
ST-MA-8	I can see who is working at the moment.	2
ST-MA-9	I can find openings in the schedule and ask employees to come and work in those openings.	4
ST-MA-10	I can filter out invalid feedback that customers send via SMS	5

Role: Chef (ST-CH)

Identifier	User Story	Size
ST-CH-1	I can receive recommendations on how to prepare the proper amount of ingredients for the restaurant at a certain time.	10
ST-CH-2	I can view the current inventory levels to ensure there is enough for the next dish/day.	5
~~ST-CH-3~~	~~I can see and update inventory levels as needed and the manager can be notified~~	~~4~~
ST-CH-4	I can see a historic record of what and when dishes have been made.	5
~~ST-CH-5~~	~~I can view customer feedback about the meals I prepare.~~	~~7~~

Role: Host (ST-HO)

Identifier	User Story	Size
ST-HO-1	I can check in both customers that made reservations and customers waiting in line, and then update table availability.	5
ST-HO-2	I can cancel a reservation and remove customers from the queue list.	4
ST-HO-3	I can see all current reservations and their phone number in case I need to call them	4
ST-HO-4	I can manually add an open reservation, specifying the customer's info and arrival info.	2
ST-HO-5	I can use a generated reservation number to refer to a customer's reservation.	2
ST-HO-6	I can easily keep track of moved and merged tables	7

Role: Customer (ST-CU)

Identifier	User Story	Size
ST-CU-1	I can reserve a table with ease	4
ST-CU-2	I can cancel a reservation without reason	3
ST-CU-3	I can receive a confirmation to make sure that the information that I provided for the reservation is accurate.	2
~~ST-CU-4~~	~~I can get a text confirmation of my reservation so that I can be confident my reservation was made~~	~~4~~
ST-CU-5	I can be seated in a timely fashion when I arrive at the establishment	3
ST-CU-6	I can easily and privately submit feedback about my experience at any time	8
~~ST-CU-7~~	~~I can easily leave a rating for my overall experience~~	~~3~~
ST-CU-8	I can leave detailed feedback to the restaurant	6

Role: General Employee (ST-GE)

Identifier	User Story	Size
ST-GE-1	I can see my work shift and hours worked in daily, weekly, monthly, yearly format as well as wage.	6
ST-GE-2	I can see who else is working at the same time that I will be working	3
ST-GE-3	I can request absence from work	3
ST-GE-4	I can post my shift and take coverages for other people	4
ST-GE-5	I can easily log into my account in the restaurant portal	3

4. Functional Requirements Specification

a. Stakeholders

One of our major stakeholders are restaurant managers and chefs. Since their goal is to raise the efficiency of their restaurant and maximize profits, our product would be very appealing to managers. Chefs in particular will find the freedom of having a voice-operated service that tells them what ingredients they need to prepare very useful.

Some equally important stakeholders also include the companies that developed the POS systems that our target restaurants use. These companies would be highly interested in a product that plugs in and adds powerful statistical analysis to their POS systems. Some popular POS systems include Vend or Bindo.

Dining customers will also find our product attractive since we provide the feature of allowing reservations to be made quickly online. In fact, our whole system is designed to reduce the wait time for all of these customers, so we believe that customers will be very supportive of our product.

Overall, we believe that all participants in the flow of a restaurant will benefit from our system.

b. Actors and Goals

Chef:

Chefs are initiators. Their primary goal is to initiate a call through either a physical interaction of the device or through the voice-operating service to give them a list of orders or ingredients to prepare.

Manager:

Managers are initiators. They have many goals. They initiate a call to get various statistics of the restaurant's order history for their own analysis. Additionally, they also have access to see which orders need to be prepared. They can also see the analysis of restaurant feedback.

Order:

Orders are participators. They're passive actors that are only used as data for our use cases.

Host:

Hosts are mainly participators. Their primary goal is to manage customer interaction before they arrive, and just when they arrive at the restaurant. This includes setting up reservations and table management.

General Employee:

General employees are initiators. They initiate access into the employee portal, where they can manage their work shifts.

Cell Provider:

The cell provider is a participating actor. The customer's respective cell provider receives a SMS message.

Twilio:

Twilio API is a participating actor. Twilio will communicate with the cell provider to gather the SMS information. Twilio is able to send text messages to the System via a Webhook, and provides an API library for use.

Amazon Alexa:

Amazon provides a voice recognition and interaction interface through its Amazon Alexa web services. The Alexa service translates spoken audio into words and then processes the spoken words to map to HTTP requests running on the ChefBoyRD service.

Database:*

The database is a participator. It's goal is to provide data that will be used to complete use cases, or use cases require participation of the database to store new data. Used as a general case to signify that different data are accessed for usec case

- Seating Schedule Database:
  Provides data regarding reservation information
- Feedback Database:
  Provides feedback data.
- Current Day's Order Data:
  Holds the information order information for just today
- Order Database
  Holds all order data

Host/Hostess Interface:**

The Host/Hostess interface is a participator. It's goal is to act as an interface for the host/hostess. This is where they will manage information that is relevant to customers

Administrative Portal:

The Administrative Portal is a participator. It's goal is to be interface where the manager can oversee information regarding the overall restaurant. This includes statistics, feedback data, reservation list, prediction data.

*All the specific database actors are listed to increase readability of Sequence Diagrams. They are in reality separate tables in the same database

** All actors technically interact through an interface. We only specify the host/hostess and the administrative portal because they have active components.

c. Use Cases

i. Casual Descriptions

<<observes>> is used for denoting an Offstage Actor

Statistics Use Case (ST-UC)

ST-UC1: Gather Statistics

- Manager <<initiates>> the use case
- Database <<participates>> by providing performance, meals, tabs, ingredients, or revenue data
- Administrative Portal <<participates>> by receiving the statistics data and displaying it with tables.

The manager can initiate a request for the displaying of restaurant statistics. They can request performance, meal, tab, ingredients, or revenue history. This information will be retrieved from the database and displayed in the statistics page with visuals.

ST-UC2: Audio-Based Reporting

- Manager or Chef <<initiates>> the use case depending on the type of report which is going to be requested.
- Amazon Alexa service <<participates>> by invoking natural language processing capabilities
- Database <<participates>> by providing performance, meals, tabs, ingredients, or revenue data

The manager can initiate a request to hear about the restaurant statistics through an Amazon Alexa capable device. They can request performance, or revenue history reports from Alexa. This information will be retrieved from the database and an audio message will be constructed which will be played back to the manager. To initiate a request, simply speak to your Alexa and ask for it.

Food/Dish Use Cases (FD-UC)

FD-UC1: Data Synchronization

- Manager or System Sync Settings <<initiates>> the use case
- Current Day's Order Data <<participates>> by providing order data
- Order Data Database <<participates>> by providing and receiving order data and the results of the prediction algorithm
- Administrative Portal <<observes>> by receiving access to the order data and prediction algorithm results

Throughout the day, data relating to the dishes ordered and the associated times is tracked. At the end of the day, this information can be added to the database used for predictive purposes by a manager, or by the program itself. The current data in the database is compiled with the new data and is then analyzed by the prediction algorithm. The results of this process are stored back into the database.

FD-UC2 Orders Prediction

- Manager or Chef <<initiates>> the use case
- Current Day's Order Data <<participates>> by providing order data
- Order Data Database <<participates>> by providing historical order data
- Administrative Portal <<observes>> by receiving the historical order data

As the data from orders is tracked on a day-to-day basis as a chef or manager it is useful to be able to view historical trends on this data. It can be used not only to make predictions but to make business decisions in terms of marketing and designing the menu for the establishment. By providing easy access to historical data the research and investment required in order to make the most informed decision is lowered.

Feedback Use Cases (FB-UC)

FB-UC1 SMS Feedback Analysis

- Customer <<initiates>> the use case
- Feedback Database <<participates>> by providing and receiving feedback data
- Administrative Portal <<observes>> by receiving access to the sorted responses and word cloud

A customer submits an open-ended response via SMS relating to their experience at the restaurant. If possible, the submission is sorted into categories based on its content, indicating which part of the dining experience the feedback relates to. General information regarding the responses, such as a word cloud, is accessible through the administrative portal.

Table Reservation Use Case (TR-UC)

TR-UC1 Online Table Reservation

- Customer <<initiates>> the use case and optionally receives a reservation confirmation message
- Seating Schedule Database <<participates>> by providing the current seating schedule and updating this data if necessary
- Host/Hostess Interface <<observes>> by noting any changes that occur in the seating schedule

A customer calls the restaurant to make a reservation, indicating party size and time of arrival. The current table schedule is accessed and compared with the request to see if an opening is available. If so, the table schedule is updated to include the new reservation and a confirmation

is sent to the customer. If not, the customer is offered an alternative time. The customer may accept or provide a new time of arrival for their party, repeating the process until either a reservation is accepted or the customer does not provide another request.

TR-UC2 In-Store Table Reservation

- Customer <<initiates>> the use case
- Seating Schedule Database <<participates>> by providing the current seating schedule and updating this data if necessary
- Host/Hostess Interface <<observes>> by noting any changes that occur in the seating schedule

A customer requests to be seated at the restaurant without a reservation, indicating party size. The table scheduling database is accessed and compared with the request to determine if a table is available. If so, the customer is seated and the table schedule is updated to reflect that the table is in use. Otherwise, the customer is allowed to wait for the next available table, placing them into a queue. The table schedule is updated to indicate that the next expected table vacancy will be reserved for the customers currently waiting in the queue.

Manage Shifts Use Case (MS-UC)

MS-UC1 Manage Shifts

- General Employee <<initiates>> the use case by logging into the employee portal
- Employee Schedule Database <<participates>> by providing the current employee work shift schedule and updating this data if necessary
- Manager <<participates>> by accessing the updated employee schedule, creating the initial schedules, and making any necessary manual overrides.

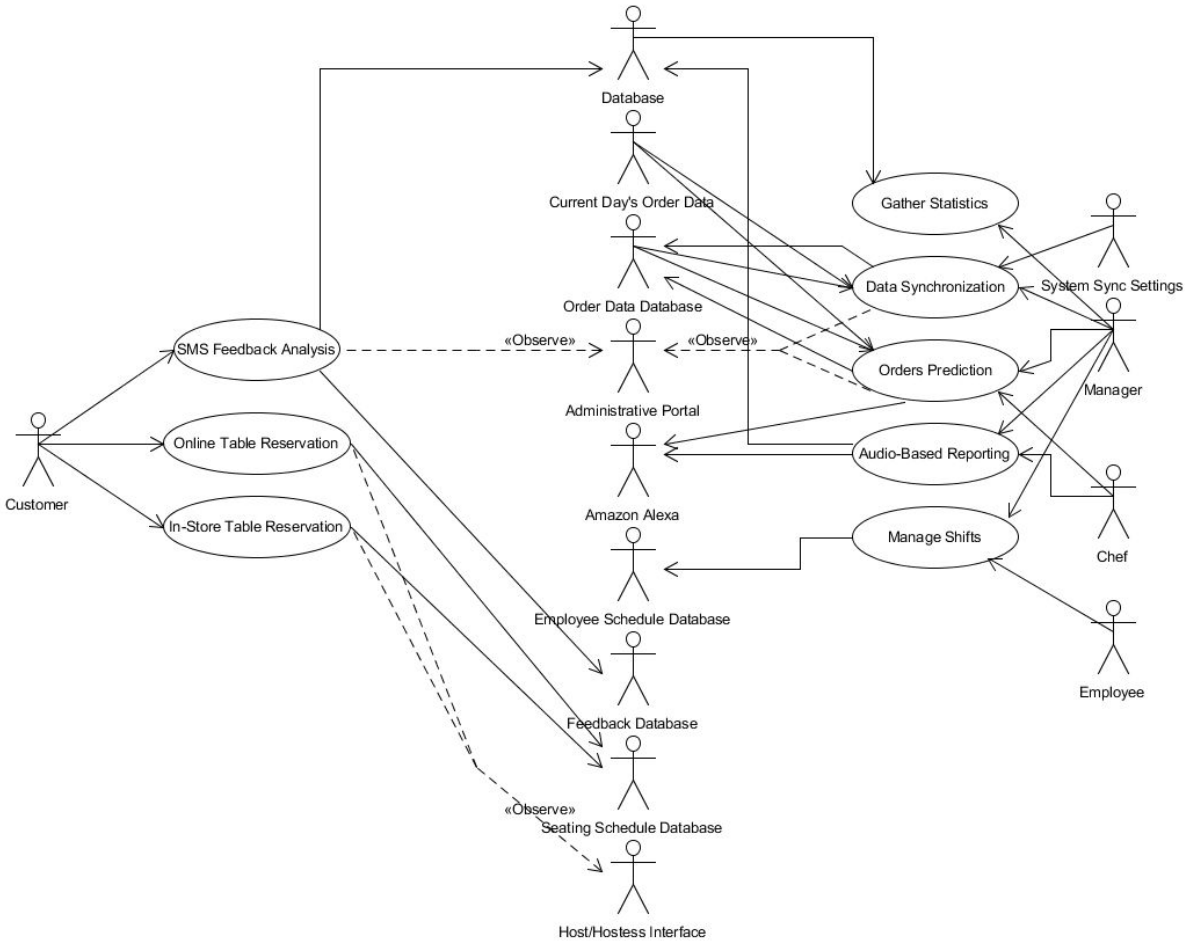
General employees can log in and access an employee portal where they may add, drop their work shifts. This in turn notifies the manager. The manager is who initially schedules these work shifts to the employees, and reserves the power to modify or change these shifts.

ii. Use Case Diagrams

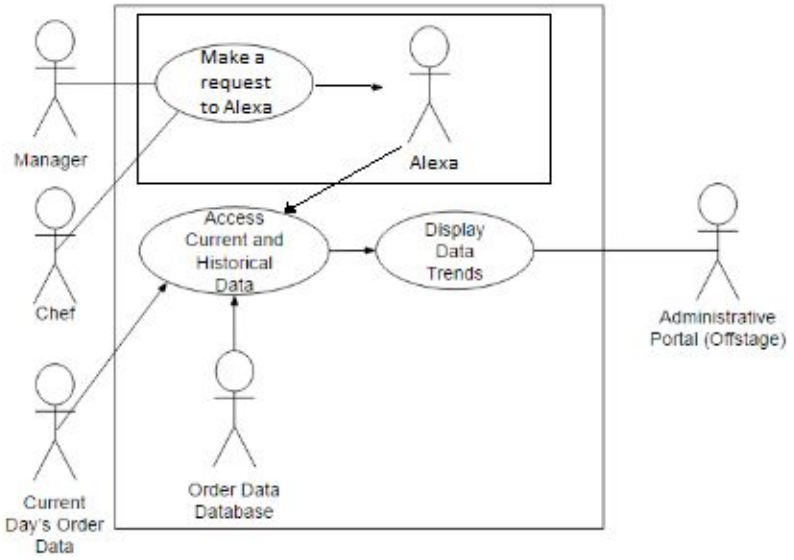
Actor → Use Case: Actor<<initiates>>

Use Case → Actor: Actor<<participates>>

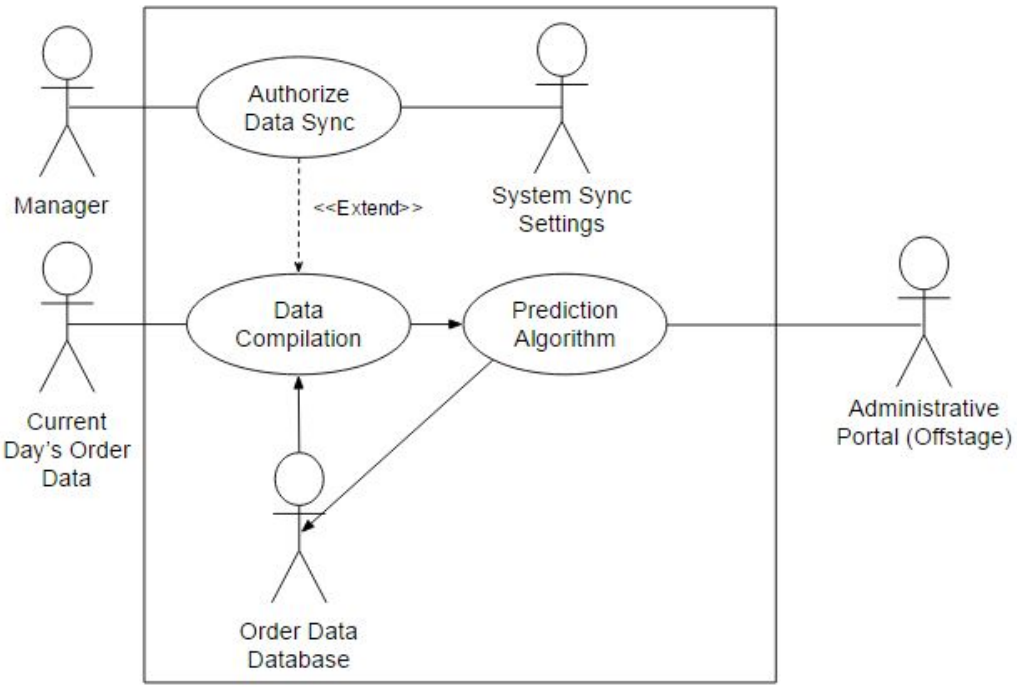
Dotted arrow: Actor <<observes>>

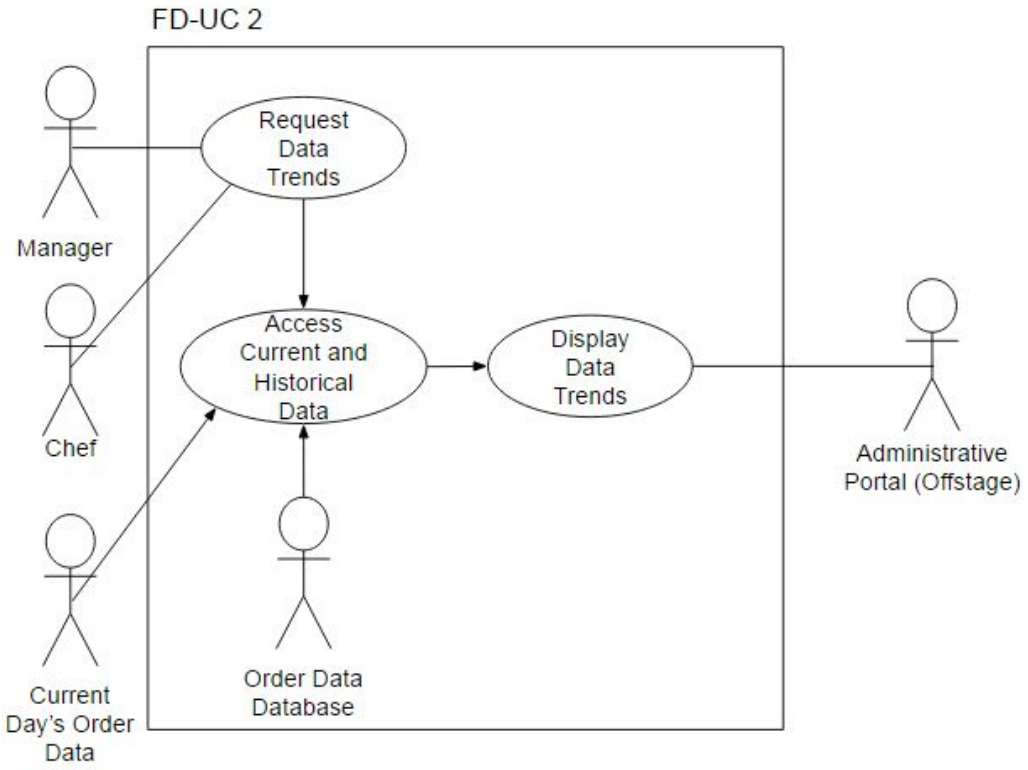


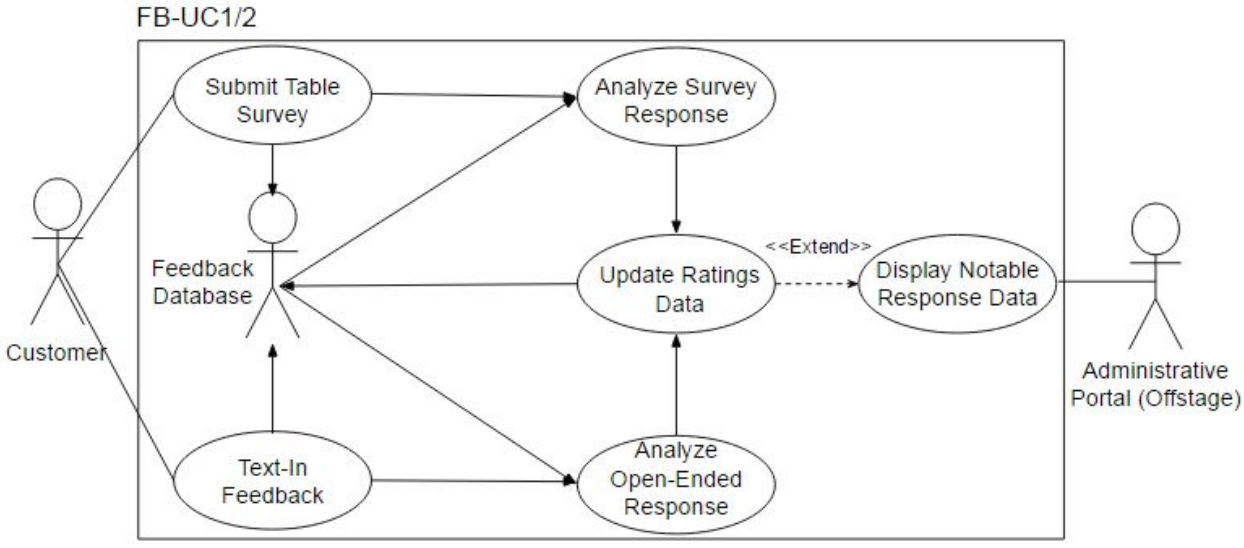
ST-UC2


Make a request to Alexa
Manager
Alexa
Chef
Access Current and Historical Data
Display Data Trends
Administrative Portal (Offstage)
Current Day's Order Data
Order Data Database

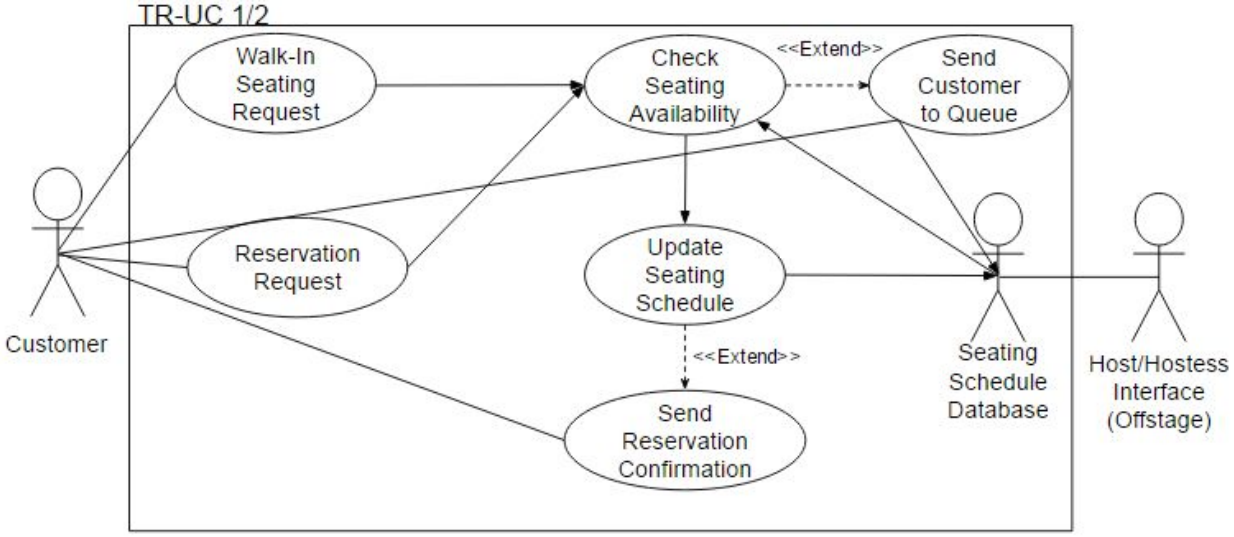
FD-UC 1


Authorize Data Sync
Manager
<<Extend>>
System Sync Settings
Data Compilation
Prediction Algorithm
Current Day's Order Data
Administrative Portal (Offstage)
Order Data Database

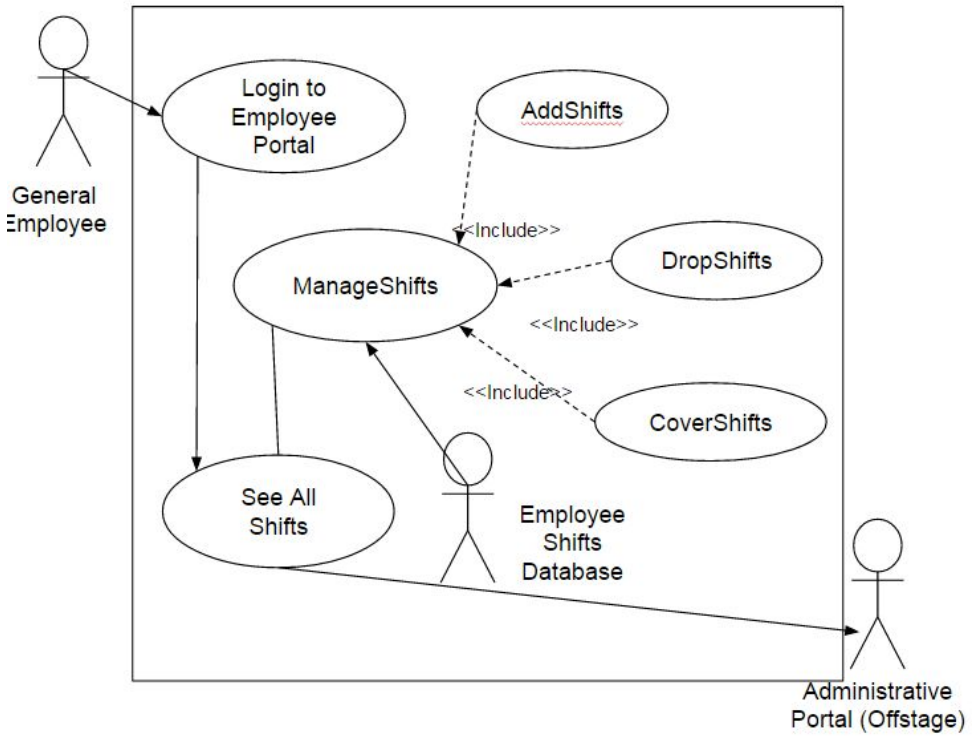

FD-UC 2
Request Data Trends
Manager
Chef
Access Current and Historical Data
Display Data Trends
Administrative Portal (Offstage)
Current Day's Order Data
Order Data Database


FB-UC1/2
Submit Table Survey
Analyze Survey Response
Feedback Database
Update Ratings Data
<<Extend>>
Display Notable Response Data
Customer
Administrative Portal (Offstage)
Text-In Feedback
Analyze Open-Ended Response

TR-UC 1/2



MS-UC 1



iii. Traceability Matrix

Crossed out Use Cases are what have been removed

	ST Size	ST-UC1	ST-UC2	FD-UC1	FD-UC2	FB-UC1	TR-UC1	TR-UC2	MS-UC1
ST-MA-1	7			1	1				
ST-MA-2	6	1	1						
ST-MA-3	5	1	1						
~~**ST-MA-4**~~	~~4~~								
ST-MA-5	8								
ST-MA-6	5					1			
ST-MA-7	7					1			
ST-MA-8	2								1
ST-MA-9	4								1
ST-MA-10	5					1			
ST-CH-1	10			1					
ST-CH-2	5			1	1				
~~**ST-CH-3**~~	~~4~~								
ST-CH-4	5				1				
~~**ST-CH-5**~~	~~7~~								
ST-HO-1	5						1	1	
ST-HO-2	4						1	1	
ST-HO-3	4						1		
ST-HO-4	5						1	1	
ST-HO-5	2						1		
ST-HO-6	7						1	1	
ST-HO-7	5							1	
ST-CU-1	4						1		
ST-CU-2	3						1		

ST-CU-3	**2**						1		
ST-CU-4	**4**						1		
ST-CU-5	**3**						1	1	
ST-CU-6	**8**					1			
~~**ST-CU-7**~~	~~**3**~~								
ST-CU-8	**6**					1			
ST-GE-1	**6**								1
ST-GE-2	**3**								1
ST-GE-3	**3**								1
ST-GE-4	**4**								1
ST-GE-5	**3**								1
	201	11	11	22	17	31	43	29	25

*Each User Story was counted multiple times (in overall total) if it was resolved by multiple Use Cases

iv. Fully Dressed Descriptions

The use cases described in this section are our primary use cases. A comprehensive list of implemented use cases are mentioned in our casual description

Use Case FD-UC1: Data Synchronization
Related Reqs: ST-MA-1, ST-MA-4, ST-CH-1, ST-CH-2, ST-CH-3
Initiating Actor: Manager or System Synchronization Settings
Actor's Goal: Update the data being used for orders prediction to include new data.
Participating Actors: Order Database, Current Day's Order Data
Preconditions: Current day's order data has been collected and not imported into learning database table. Manager has been authenticated,
Postconditions: The system will make sure the current day's data is moved into the database. We ensure that we do not duplicate data.
Flow of Events for Main Success Scenario:

1. → Manager or System Settings <<initiates>> the request by either manually requesting the synchronization through the Manager's UI or the System's scheduler.
2. ← The system requests data from Current Day's Order Data
3. → The Current Day's Order Data sends requested data.
4. ← The system requests data from Order Database
5. → The Order Database sends requested data.
6. ← System compares previous data with current order's data. Updates prediction
7. ← System updates Order Database

Flow of Events for Extensions:

1. → Actor requests synchronization when there is no data
    a. ← System responds letting the user know there is no data to copy
2. → Actor attempts to request synchronization without authentication/authorization
    a. ← Return error message letting the actor know that they must be authorized to perform the operation.
3. → The database server could not be found or connect.
    a. ← The system lets the user know that the database could not be connected. Notifies the user whether the database service can be started.
    b. → Actor waits a specified amount of time for system to start the database.
    c. ← Database has not started
    d. → Actor is given the option to cancel the synchronization operation and to call support.

Use Case FD-UC2: Orders Prediction

Related Reqs: ST-MA-1, ST-MA-2, ST-MA-3, ST-CH-2, ST-CH-3, ST-CH-4
Initiating Actor: Manager or Chef
Actor's Goal: Determine the number of ingredients that will be consumed in a given time period.
Participating Actors: Order Database, Current Day's Order Data
Preconditions: Order data from previous day is stored within the order database. Manager or chef has been authenticated
Postconditions: The system responds with a set of data based on its prediction algorithm. The system will store its prediction to determine how accurate it the prediction was for future use.
Flow of Events for Main Success Scenario:

1. → Chef or Manager <<initiates>> the request either through the use interface or the voice service by providing the day and time that the actor would like information on.
2. ← The system requests data from Current Day's Order Data
3. → The Current Day's Order Data sends requested data.
4. ← The system requests data from Order Database
5. → The Order Database sends requested data.
6. ← System compares previous data with current order's data. Updates prediction
7. ← System updates Order Database
8. ← The system responds with data representing the ingredients needed for the orders in a given period of time

Flow of Events for Extensions:

1. → Initiating actor requests data for a time in the past
    a. ← System responds with invalid time exception
2. → Initiating actor requests prediction when there is no order database connected or the order database is empty
    a. ← System responds with a message notifying the user that it can't make predictions without data.

Use Case FB-UC1: SMS Feedback Analysis

Related Reqs: ST-MA-6, ST-MA-7, ST-CU-6, ST-CU-8
Initiating Actor: Customer
Actor's Goal: Provide feedback relating to overall dining experience.
Participating Actors: Feedback Database, Administrative Portal, Cell Provider, Twilio
Preconditions: The customer possesses a unique key from their dining receipt that will enable them to submit feedback.
Postconditions: Feedback is stored and available for later access.
Flow of Events for Main Success Scenario:

1. → Customer <<initiates>> SMS analysis by submitting a SMS and unique key to Cell Provider
2. → Cell Provider provides SMS data to Twilio
3. → Twilio sends SMS data to System
4. ← System checks key to see if it's valid.
5. ← SMS input is analyzed by the program to determine relevant categories.
6. ← The Feedback Database is updated to include the new SMS and its associated categorization results.

Flow of Events for Extensions:

1. → The customer enters an invalid key
    a. ← The system will respond to the customer's text with a response indicating an invalid key.
2. → The Administrative Portal requests to see SMS feedback data within a date range and specifies categories
    a. ← The system displays the SMS submission time and body text in a table.
3. → The Administrative Portal requests to see a word cloud along with the SMS data within a date range and specified categories
    a. ← The system will collect the text through all the SMS data specified, and calculate the word frequency. The system then displays this word frequency in a word cloud.

Use Case TR-UC1: Online Table Reservation

Related Reqs: ST-HO-1, ST-HO-2, ST-HO-3, ST-HO-4, ST-HO-5, ST-HO-6, ST-CU-1, ST-CU-2, ST-CU-3, ST-CU-4, ST-CU-5

Initiating Actor: Customer

Actor's Goal: Make an online reservation including time and date of arrival and party size.

Participating Actors: Host/Hostess Interface, Seating Schedule Database

Preconditions: Customer is on the website. The restaurant is open.

Postconditions: Customer's reservation is set in the database and able to be seen by the host

Flow of Events for Main Success Scenario:

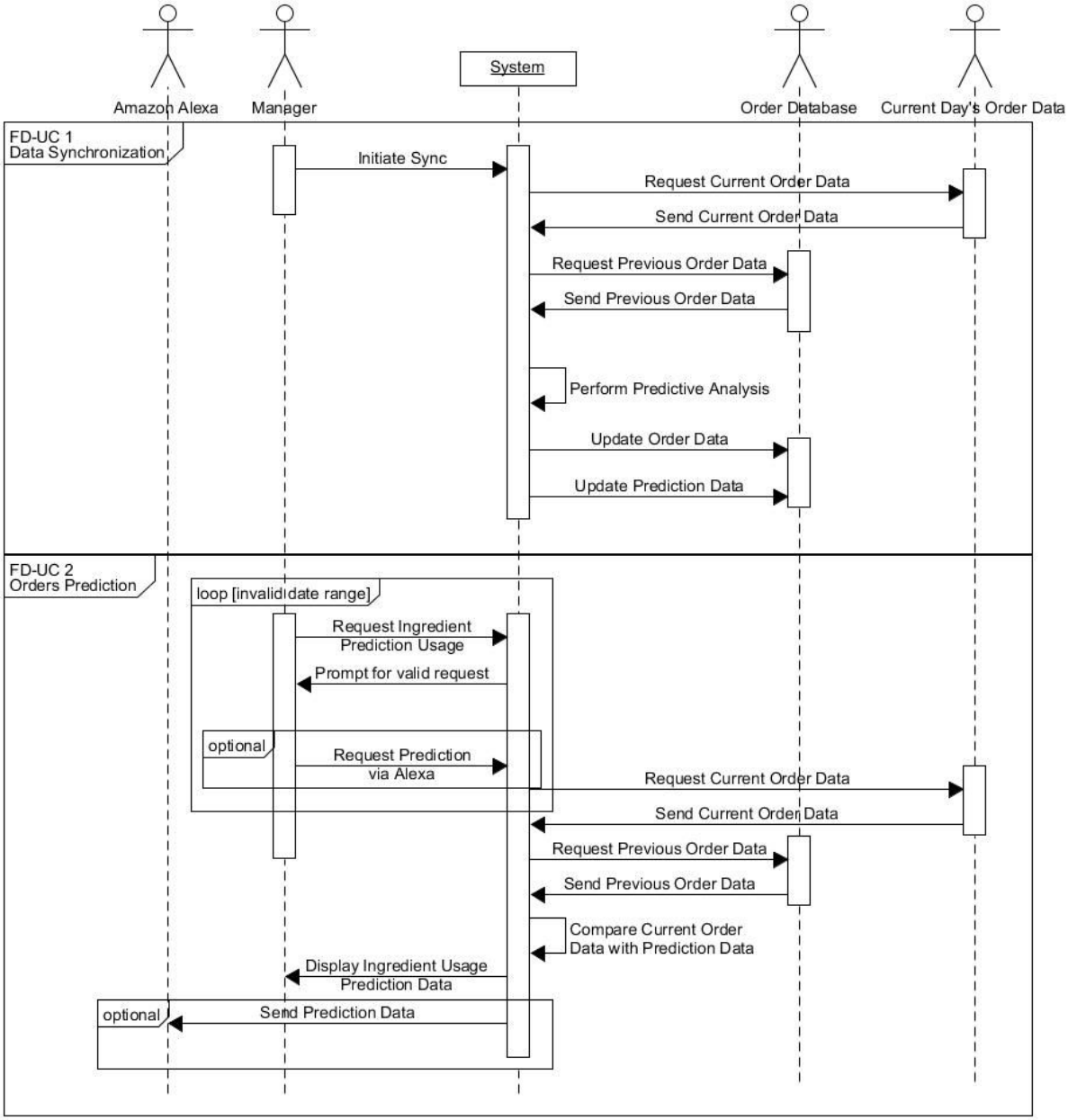
1. → Customer accesses ChefBoyRD website, and clicks reservation tab.
2. → Customer populates information about their reservation. Date/time, phone number.
3. ← System responds by requesting reservation data from main server.
4. ← System checks the reservation information and confirms the slot is empty.
5. ← System updates reservation information to the database.
6. ← System sends the user a reservation confirmation.

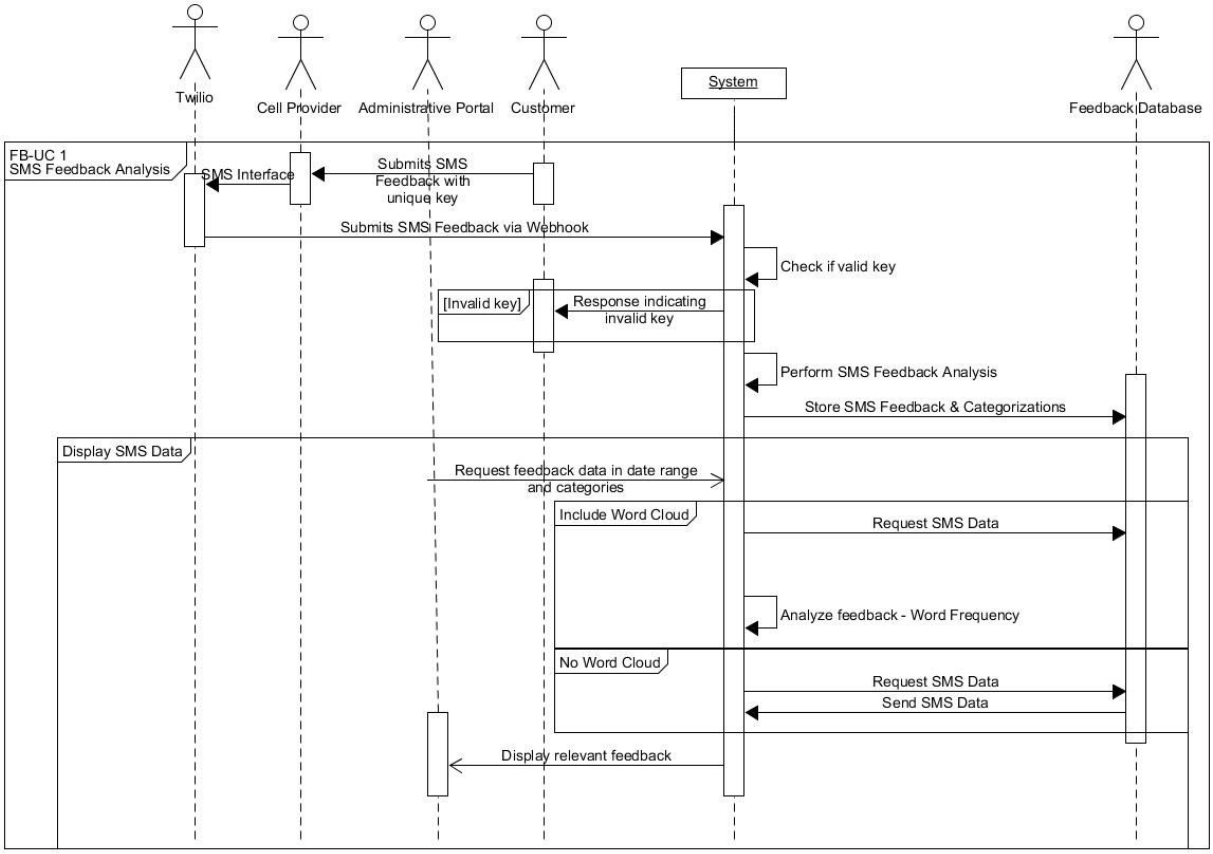
Flow of Events for Extensions:

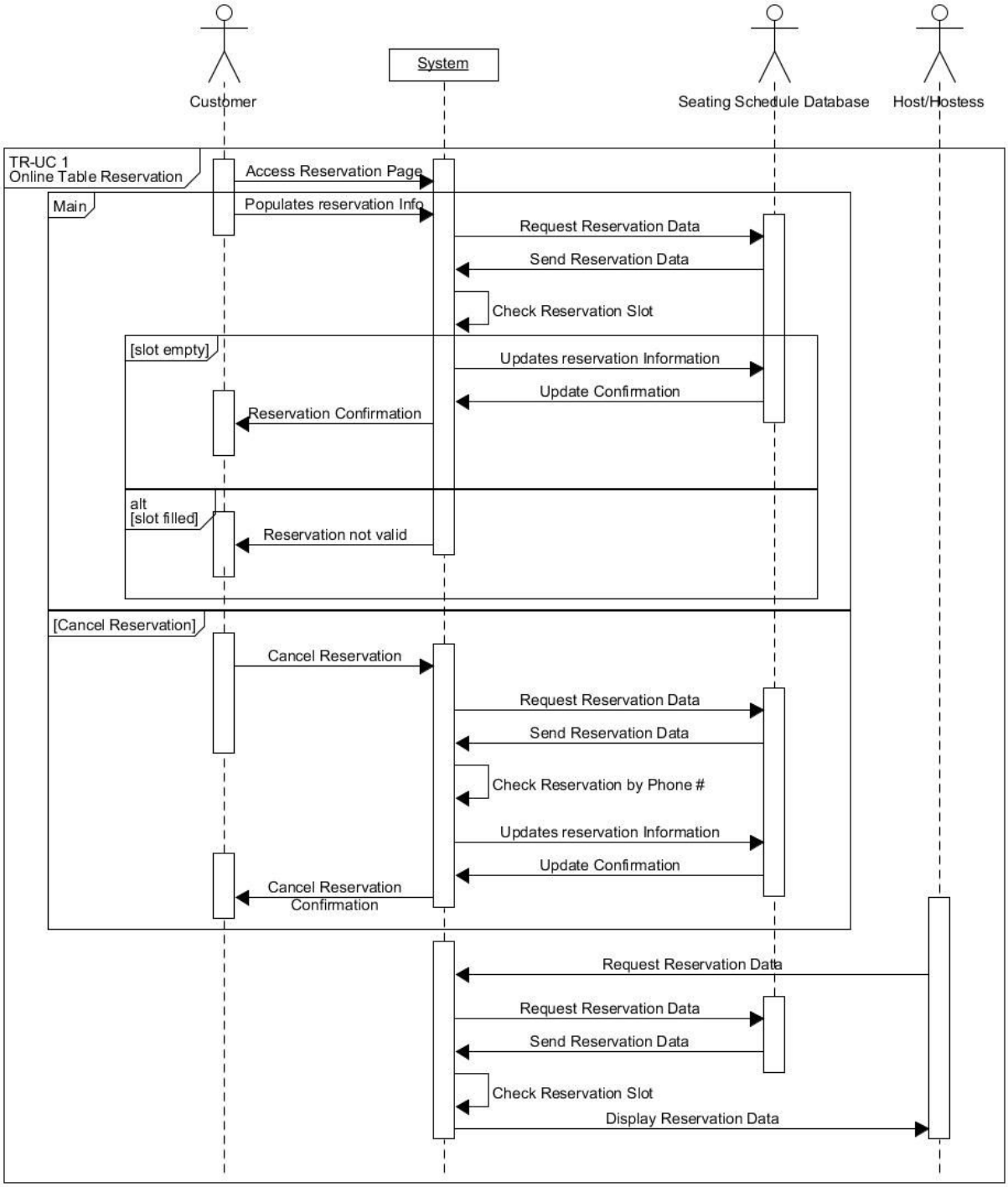
1. → Customer requests a reservation that is already filled
    a. ← The system responds by checking if the reservation is available.

   b. ← The system notifies customer that it is not a valid reservation.
2. → Customer wishes to cancel a reservation
   a. ← The system responds by checking to see if the reservation matches the customer who is requesting it. (By phone number)
   b. ← Upon confirmation, the system makes a change to the reservations list and updates the database.
   c. ← The system sends a confirmation to the user that reservation has been cancelled.
3. → Host/Hostess Interface requests to see the table reservation information
   a. ← The system sends the table reservation information to display

d. System Sequence Diagrams




Twilio
Cell Provider
Administrative Portal
Customer
System
Feedback Database
FB-UC 1
SMS Feedback Analysis
SMS Interface
Submits SMS Feedback with unique key
Submits SMS Feedback via Webhook
Check if valid key
[Invalid key]
Response indicating invalid key
Perform SMS Feedback Analysis
Store SMS Feedback & Categorizations
Display SMS Data
Request feedback data in date range and categories
Include Word Cloud
Request SMS Data
Analyze feedback - Word Frequency
No Word Cloud
Request SMS Data
Send SMS Data
Display relevant feedback


Customer
System
Seating Schedule Database
Host/Hostess
TR-UC 1
Online Table Reservation
Access Reservation Page
Main
Populates reservation Info
Request Reservation Data
Send Reservation Data
Check Reservation Slot
[slot empty]
Updates reservation Information
Update Confirmation
Reservation Confirmation
alt
[slot filled]
Reservation not valid
[Cancel Reservation]
Cancel Reservation
Request Reservation Data
Send Reservation Data
Check Reservation by Phone #
Updates reservation Information
Update Confirmation
Cancel Reservation
Confirmation
Request Reservation Data
Request Reservation Data
Send Reservation Data
Check Reservation Slot
Display Reservation Data

5. Effort Estimation

Use Case Weights

Use Case	Description	Category	Weight
Gather Statistics (ST-UC-1)	Moderate User Interface. 3 steps for main success scenario. 2 participating actors (Database, Administrative Portal)	Average	10
Audio-based Reporting(ST-UC-2)	Simple User Interface. 1 step for main success scenario. 4 participating actors(Chef, Manager, Alexa, Administrative Portal)	Simple	5
Data Synchronization (FD-UC-1)	Simple User Interface. 7 steps for main success scenario. 2 participating actors. (Current Day's Order Data, Order Database)	Simple	5
Orders Prediction (FD-UC-2)	Complex User Interface & Processing. 7 steps for main success scenario 2 Participating Actors	Complex	15
SMS Feedback Analysis (FB-UC-2)	Complex User Interface & Processing. 6 steps for main success scenario 4 Participating Actors (Feedback Database, Administrative Portal, Cell Provider, Twilio)	Complex	15
Online Table Reservation (TR-UC-1)	Moderate User Interface. 6 steps for main success scenario. 2 participating actors (Host/Hostess Interface, Seating Schedule Database)	Average	10
In-Store Table Reservation (TR-UC-2)	Moderate User Interface. 6 steps for main success scenario. 2 participating actors (Host/Hostess Interface, Seating Schedule Database)	Average	10
Manage Shifts (MS-UC-1)	Moderate User Interface. 7 steps for main success scenario. 2 participating actors(Manager. Employee Schedule Database)	Average	10
	Unadjusted Use Case Weight		**80**

Actors Weights

Actor name	Description of relevant characteristics	Complexity	Weight
Manager	Uses a combination of textual and graphical interfaces for viewing feedback, viewing statistics, etc	Complex	3
Chef	Uses a textual interface to see ingredient statistics	Average	2
Host/Hostess	Uses a graphical interface to manage tables. Uses a text-based and graphical interface to manage reservations	Complex	3
General Employee	Manages their shift via a graphical user interface. Enters text to login and logout	Complex	3
Twilio	Interacts with our system with a defined API	Simple	1
Cell Provider	Interacts with Twilio	Simple	1
Database	The database sends data to the system based on queries	Simple	1
Alexa	Interprets speech and returns output for requests such as statistics, predictions, etc.	Average	2
	Unadjusted Actor Weight		**16**

Unadjusted Use Case Points = UAW + UCW = **96**

Technical Complexity Factors

Technical Factor	Description	Weight	Perceived Complexity	Calculated Factor (Weight x Perceived Complexity)
T1	Distributed system - Web-based system	2	2	4
T2	Performance objectives - requires moderately quick response for DB queries	1	3	3
T3	End-user efficiency - Simple use by users	1	4	4
T4	Complex internal processing - processing information for restaurant automation	1	5	5
T5	Reusable design or code - for adding features	1	3	3
T6	Easy to install - simple for restaurant to set up	0.5	3	1.5
T7	Easy to use - straightforward user interfaces	0.5	6	3
T8	No need for portability	2	0	0
T9	Easy to change is preferable for adding new features	1	2	2
T10	Concurrent use (by multiple users) - required as a web server	1	4	4
T11	Special security features - phone numbers, business-sensitive information	1	4	4
T12	No direct access for third parties	1	0	0
T13	No unique training needs	1	0	0
		Technical Factor Total		**33.5**
0.6 + 0.01*33.5		**Technical Complexity Factor**		**0.935**

Environmental Complexity Factors

Environmental factor	Description	Weight	Perceived Impact	Calculated Factor(Weight x Perceived Impact)
E1	Familiar with the development process (e.g., UML-based) - all beginner at UML	1.5	1	1.5
E2	Application problem experience - half have experience. Half does not	0.5	3	1.5
E3	Paradigm experience (e.g., object-oriented approach) - most has some knowledge	1	2	2
E4	Lead analyst capability - very capable and experienced lead	0.5	5	2.5
E5	Mostly not motivated	1	2	2
E6	Stable requirements expected	2	3	6
E7	No part-time staff involved	-1	0	0
E8	Simple-Average programming language. But many languages need to be used	-1	6	-6
	Environmental Factor Total:			**9.5**
= 1.4 - 0.03*9.5	**Environmental Complexity Factor**			**1.115**

Use Case Points = 100.08 = ECF x TCF x UUCP

Duration = 2351 = UCP * PF = 97.997 * 28

6. Domain Analysis

a. Domain Model

i. Concept Definitions

D for do, K for know

RS	Responsibility Description	Type	Domain Name
1	Gets desired data from database. Also updates database	D	DatabaseConnection
2	Display statistics data	K	StatisticsDisplay
3	Gets query information from user	D	StatisticsQueryHelper
4	Display predictions	K	PredictionDisplay
5	Uses prediction algorithm on data received from statistics database	D	PredictionHandler
6	Alexa Interface to interpret voice commands and says speaks prediction & statistics	D	AlexaInterface
7	Analyze data using prediction algorithm	D	AnalyzePrediction
8	Gets feedback query information from user	D	FeedbackQueryHelper
9	Display feedback data	K	FeedbackDisplay
10	Store SMS data sent by customer	D	TwilioInterface
11	Handles manipulations to feedback objects	D	FeedbackController
12	Sets categories to feedback objects	D	FeedbackAnalyzer
13	Analyzes frequency of words in feedback query	D	WordCounter
14	Holds all the keys that are in valid Tabs	K	TabsKeys
15	Display reservation form	K	ReservationForm
16	Interprets Reservation Data and sends to DB	D	ReservationHandler
17	Display reservation data	K	ReservationDisplay

18	Check if Reservation is empty & in valid time & seats available	D	ReservationChecker
19	Display tables & interpret changes in position	D	TableGUI
20	Check if Table is empty & can fit all people in party	D	TableChecker
21	Display employee shift data	K	ShiftDisplay
22	Checks ownership of shift	D	ShiftChecker
23	Add, Drop, Cover shifts	D	ShiftManager
24	Interface for user to do add drop, cover shifts	D	ShiftActions

ii. Association Definitions

Concept Pair	**Association Description**	**Association Name**
StatisticsQueryHelper <-> DatabaseConnection	StatisticsQueryHelper passes query instructions to the database connection, which makes those queries and sends back data to StatisticsDisplay	conveys request
DatabaseConnection <-> StatisticsDisplay	Updates with information from database based on request	updates
DatabaseConnection <-> PredictionHandler	Updates display with statistics information	updates
PredictionHandler <-> PredictionDisplay	Updates display with prediction data	updates
PredictionHandler <-> AnalyzePrediction	Based on data received from database, create predictions based on the query specified	uses
PredictionHandler	Send request for Alexa to output voice	send request

<-> AlexaInterface	commands	
StatisticsQueryHelper <-> AlexaInterface	Uses alexa to send a database query for statistics, instead of using a display interface	Send request
TwilioInterface <-> FeedbackController	Gets the SMS from Twilio and send to controller to be processed	updates
FeedbackController <-> DatabaseConnection	Updates database with SMS after analysis.	updates
TabsKeys<-> DatabaseConnection	Updates the list of keys that are valid from valid tabs	updates
FeedbackController <-> WordCounter	Uses the WordCounter function to get needed information for word cloud	uses
FeedbackController <-> FeedbackAnalyzer	Uses the FeedbackAnalyzer function to sort SMS feedback into categories	uses
FeedbackController <-> FeedbackDisplay	Updates display with feedback-related information	updates
FeedbackController <-> TabKeys	Checks to see if submitted feedback has key matching a key associated tab in database	check valid
FeedbackQueryHelper <-> FeedbackController	Conveys the query request to controller, which then sends to databaseConnection	conveys request
ReservationHandler <-> DatabaseConnection	ReservationHandler passes query instructions to the database connection, which makes those queries and sends back data to ReservationDisplay	conveys request
ReservationChecker <-> Database Connection	ReservationHandler passes query instructions to the database connection, which makes those queries and sends back data to ReservationDisplay	conveys request
ReservationHander <-> ReservationDisplay	Updates reservation information for the user to see	updates
ReservationHandler <->	Checks the input reservation with the current reservations retrieved from database	check available

ReservationChecker		
ReservationForm <-> ReservationHandler	Communicates the customer's reservation to the handler so it can be checked and stored	updates
TableGUI <-> DatabaseConnection	Updates database with added or removed tables	updates
TableGUI<->TableChecker	Checks for the availability of the table, the size limit, whether tables are full	check available
ShiftManager<->ShiftDisplay	Updates shift display with shift information	updates
ShiftManager<->ShiftChecker	Checks to see if shift action is valid	check validity
ShiftActions <->ShiftManager	Sends user's actions to the ShiftManager	convey request
ShiftManager<-> DatabaseConnection	Updates database with new shift information or requests for shift data in database	updates

iii. Attribute Definitions

Concept	**Attributes**	**Attribute Descriptions**
StatisticsQueryHelper	Date range	Range of dates to gather statistics or predict information on
	Option	Choose from querying for: tabs, ingredients, performance, orders
PredictionHandler	StatisticObject	The object of whichever item was queried.
AnalyzePrediction	Regression Model Type	Sinusoidal or Polynomial
	Date range	Range of future dates to do the prediction analysis on
StatisticsDisplay	Graphs	Several graphs by hour, day, or by day of the week
PredictionDisplay	Object list	List of objects that will be needed in the future queried date range

TwilioInterface	Account Credentials	Credentials needed for access to using Twilio's API
	Customer Phone Num	Phone number of customer who sends in text
feedbackDisplay	Word cloud	Word cloud object for showing frequency of word
	List of feedback	List of feedback that was queried for displayed in a table.
feedbackQueryHelper	Date range	The date range of feedback to search for
	Categories	Which categories to sort the feedback from. Positive, Negative, mixed etc
feedbackAnalyzer	Keywords	Keywords differing by category. Keywords to search for, that are associated to good, bad, food-related, service-related
TabsKeys	Unique_ID	Unique ID associated to each tab
ReservationForm	Contacti nfo	Contact info of the customer who is making the reservation
	Reservation_detail	Date of reservation, number of people in party
reservationDisplay	List of reservations	List of reservations that are coming up
	Reservation_details	Details for each reservation that is shown. This includes time, date, contact info, etc
TableGUI	List of tables	List of tables that are currently active
	Table_status	Size of table, whether it is available or not, position of table
ShiftAction	Action	Specify the action the user wants to execute. Add, drop, etc
	Shift	Specify the shift that the user wants to act on
ShiftDisplay	List of shifts	List of shifts active for the user
	Shift_details	Details of the shift such as starting time, ending time, role
ShiftChecker	Shift schedule	Shift schedule to compare the current shift to

iv. Traceability Matrix

	A	B	C	D	E	F	G	H	I
1	**Domain Concepts**	**Use Cases**							
2		ST-UC1	ST-UC2	FD-UC1	FD-UC2	FB-UC1	TR-UC1	TR-UC2	MS-UC1
3	DatabaseConnection	x	x	x	x	x	x	x	x
4	StatisticsDisplay	x							
5	StatisticsQueryHelper	x	x						
6	PredictionDisplay			x	x				
7	PredictionHandler			x	x				
8	AlexaInterface		x		x				
9	AnalyzePrediction				x				
10	FeedbackQueryHelper					x			
11	FeedbackDisplay					x			
12	TwilioInterface					x			
13	FeedbackController					x			
14	FeedbackAnalyzer					x			
15	WordCounter					x			
16	TabsKeys					x			
17	ReservationForm						x		
18	ReservationHandler						x	x	
19	ReservationDisplay						x	x	
20	ReservationChecker						x	x	
21	TableGUI						x	x	
22	TableChecker						x	x	
23	ShiftDisplay								x
24	ShiftChecker								x
25	ShiftManager								x
26	ShiftActions								x

Overall, our mapping from use cases to domain concepts follows a MVC pattern. We have a View, which is the interface the the user will see. (ReservationDisplay). This view also requires a concept to help the user make queries to the database (ReservationForm). This information is sent to the handler to be checked and then finally sent to the database. This information is processed and sent to the database, where information is either stored or retrieved. This information is most often sent to a Controller, (ReservationHandler), where it will be checked against the current items in the database if necessary. Once all the logic and checking is done, then the information is returned back to the user, (ReservationDisplay), to await further instruction.

ST-UC1 is most simply a query to the database to retrieve information for the user to see. Therefore it uses the StatisticDisplay for the display, StatisticQueryHelper to accept user queries, and a Database connection to get the data.

ST-UC2 reports statistic data using the Alexa. This use case is very similar to ST-UC1 in it's operation. Except instead of using the StatisticsDisplay, the AlexaInterface is used to communicate with the database.

FD-UC1 is for updating the desired statistics data so that it can be used for the prediction algorithm. This requires the PredictionDisplay for interfacing with the user and the PredictionHandler which is how the prediction is communicated to the user. This does not do the actual prediction algorithm, that is left up to FD-UC2

FD-UC2 is for updating the view with the desired prediction data. It can also be used to interface with Alexa and give predictions to the user. Therefore it's related to the AlexaInterface and PredictionHandler for generating the prediction data.

FB-UC1 is for when customers submit their feedback, the feedback is analyzed, and the manager can access this feedback information. It needs TabKeys to verify if the feedback submitted is valid. WordCounter and FeedbackAnalyzer to analyze the feedback and prepare the data to be shown in FeedbackDisplay. The TwilioInterface is for how the SMS will be send from Twilio to the system. FeedbackQueryHelper is when the manager wants to access specific types of feedback in the database. The FeedbackController handles all the analysis and verification.

TR-UC1 and TR-UC2 are different in the way that in TR-UC1, the online reservation form is used, and in TR-UC2 the form is not used and the host will enter the reservation information manually. TableGUI helps host to manage which tables are available. TableChecker is needed to check availability of tables. ReservationHandler will coordinate this processing.

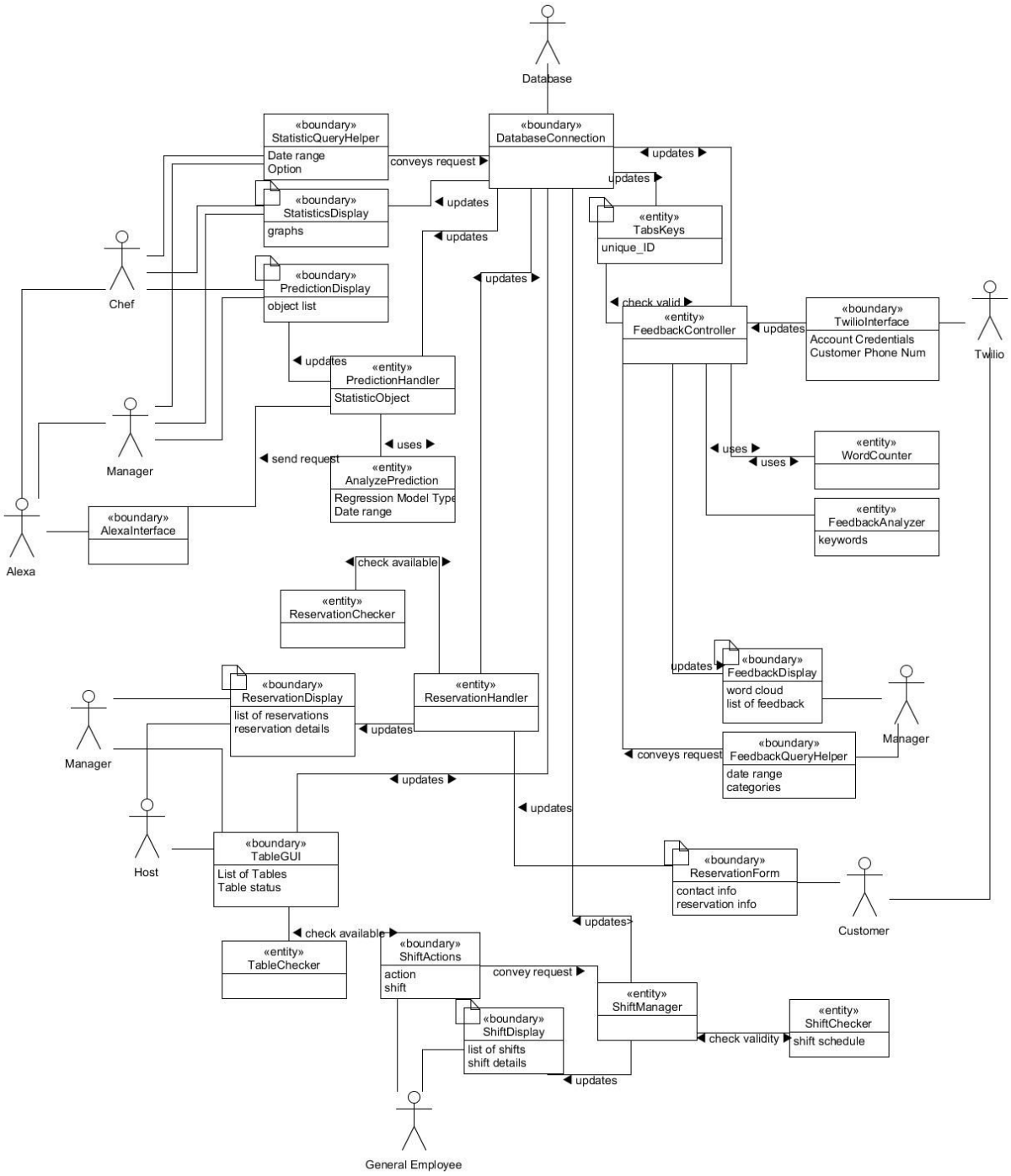
MS-UC1 the shifts are entered with ShiftAction, where it processed by ShiftManager and is checked by ShiftChecker. ONce it is deemed valid, the updated shift schedule is sent to ShiftDisplay.

There are cases where the query is sent straight to the database, and the result is handled by a controller. This is when the query does not contain information that needs to be explicitly checked against other elements in the database. Statistics does not require checking queries, where reservation must check an entered reservation to make sure the slot is open.

We can see that a majority of the project focuses on checking the validity of entered data, and then displaying this data is a graphical or useable form for the user, which in most cases is the manager).

v. Domain Model Diagram

Manager listed three times to avoid complicating diagram with interactions crossing many lines



Some notes for reading Domain model diagram: Statistics and Prediction Use case located in top left. Feedback use case located on right. Shift Management Use case located on bottom. Table Management and Reservation located on bottom left

b. System Operation Contracts

FD-OC-1	
Operation:	initiateSync()
Cross References:	FD-UC-1
Preconditions:	A system sync is desired and started by either the manager or the program itself.
Postconditions:	The system sync is initiated.

FD-OC-2	
Operation:	requestCurrentOrderData() + sendCurrentOrderData(currentOrderData)
Cross References:	FD-UC-1, FD-UC-2
Preconditions:	The current day's order data contains accurate and up-to-date information.
Postconditions:	The relevant current order data is available for use by the system.

FD-OC-3	
Operation:	requestPreviousOrderData() + sendPreviousOrderData(orderData)
Cross References:	FD-UC-1, FD-UC-2
Preconditions:	The order database contains accurate and up-to-date information.
Postconditions:	The relevant previous order data is available for use by the system.

FD-OC-4	
Operation:	performPredictiveAnalysis(surveyData)

Cross References:	FD-UC-1
Preconditions:	Both current and prior order data are available for analysis.
Postconditions:	A new prediction is generated using the combined data.

FD-OC-5

Operation:	updateOrderData(currentOrderData) + updatePredictionData(predictionData)
Cross References:	FD-UC-1
Preconditions:	A predictive analysis has been performed and its results must be recorded.
Postconditions:	The order database is updated to include the current day's order data. The order database prediction is updated to the most recent prediction.

FD-OC-6

Operation:	requestIngredientUsagePrediction()
Cross References:	FD-UC-2
Preconditions:	Authorized user desires access to prediction data.
Postconditions:	The system begins accessing prediction data for use.

FD-OC-7

Operation:	compareCurrentOrderDataWithPredictionData(currentOrderData, predictionData)
Cross References:	FD-UC-2

Preconditions:	Both current order data and most recent prediction are available for comparison.
Postconditions:	The relationship between the orders that have been placed and the orders that were expected is known.

FD-OC-8

Operation:	displayIngredientUsagePredictionData(predictionData)
Cross References:	FD-UC-2
Preconditions:	The prediction data has been fetched from the database and compared with current orders.
Postconditions:	Raw prediction data is displayed to the user. Comparison between expected and current results is displayed to the user.

FD-OC-9

Operation:	Send Prediction Data
Cross References:	FD-UC-2
Preconditions:	Amazon Alexa is interfaced with device. Prediction via Alexa was selected over visual display
Postconditions:	Alexa now has necessary commands to output prediction data

FB-OC-1

Operation:	requestFeedbackData() + sendFeedbackData(feedbackData)
Cross References:	FB-UC-1,
Preconditions:	The feedback database contains accurate and up-to-date data.

Postconditions:	The relevant feedback data from the database is available for use by the system.

FB-OC-2

Operation:	submit_SMS_Feedback_via_webhook(sms)
Cross References:	FB-UC-1,
Preconditions:	Twilio received a SMS from cell provider. Twilio connects to System with valid auth credentials
Postconditions:	SMS feedback can be accessed in system

FB-OC-3

Operation:	check_if_valid_key(key)
Cross References:	FB-UC-1,
Preconditions:	SMS object received from Twilio
Postconditions:	Key is known to be valid or invalid. If invalid, respond indicating invalid response If key is valid, key is replaced and message proceeds to be stored

FB-OC-4

Operation:	Perform SMS feedback Analysis(SMS)
Cross References:	FB-UC-1
Preconditions:	The text-in response and relevant prior feedback data are available for analysis.
Postconditions:	Response has been parsed for keywords and sentiment. Appropriate category for the response has been identified.

	New response is accurately depicted in the word cloud.

FB-OC-5

Operation:	StoreSMS_feedback_and_categorizations(sms)
Cross References:	FB-UC-1
Preconditions:	Feedback has been analyzed and appropriate tags are available for sms
Postconditions:	Analyzed feedback is stored in database

FB-OC-6

Operation:	analyze feedback - word frequency(sms)
Cross References:	FB-UC-1
Preconditions:	Sms feedback is available in database
Postconditions:	Each word in SMS feedback now has a frequency

FB-OC-7

Operation:	display_relevant_feedback(sms)
Cross References:	FB-UC-1
Preconditions:	The response has been analyzed and an appropriate sorting category is selected.
Postconditions:	The feedback database has received the most recent text-in response. The new response is placed within the category selected through analysis.

	Feedback categorizations shown in administrative portal in a table

FB-OC-8

Operation:	display_relevant_feedback(sms) <<extend>>
Cross References:	FB-UC-1
Preconditions:	The word cloud and database have been updated to reflect current data.. The response has been analyzed and an appropriate sorting category is selected.
Postconditions:	Up-to-date information of both sorted responses and the wordcloud are available through the administrative portal. The feedback database has received the most recent text-in response. The new response is placed within the category selected through analysis. Feedback categorizations shown in administrative portal in a table

TR-OC-1

Operation:	PopulatesReservationInfo()
Cross References:	TR-UC-1
Preconditions:	The customer has a desired time and party size for their reservation.
Postconditions:	The desired time and party size are known by the system.

TR-OC-2

Operation:	requestReservationData() + sendReservationData()

Cross References	TR-UC-1
Preconditions:	A reservation request is currently in progress.
Postconditions:	The current schedule is available to the system for use.

TR-OC-3

Operation:	checkReservationSlot(time, partySize)
Cross References:	TR-UC-1
Preconditions:	The desired time and party size are known by the system and the current schedule is available.
Postconditions:	Availability of desired reservation is known.

TR-OC-4

Operation:	updateReservationInformation()
Cross References:	TR-UC-1
Preconditions:	A reservation has been made for an available table at a known time.
Postconditions:	The seating schedule database has been updated to include the new reservation.

TR-OC-5

Operation:	reservationConfirmation(time, tableNumber)
Cross References:	TR-UC-1
Preconditions:	A reservation for a specific time and table number has been successfully made.
Postconditions:	Customer has received confirmation of their reservation.

TR-OC-6

Operation:	display_reservation_data(time, tableNumber)
Cross References:	TR-UC-1
Preconditions:	The seating schedule database has been changed.
Postconditions:	The seating schedule display has been updated to match the database.

TR-OC-7

Operation:	reservation_not_valid(time, tableNumber)
Cross References:	TR-UC-1
Preconditions:	The reservation has been checked and is not valid
Postconditions:	The user is notified that the reservation is not valid

TR-OC-8

Operation:	cancel_reseravtion(name)
Cross References:	TR-UC-1
Preconditions:	The reservation is valid.
Postconditions:	The reservation is available to the system

c. Mathematical Models

We are attempting to model the number and types of orders in a restaurant. These two features are interesting because it involves a classification problem intertwined within a continuous time series estimation problem.

The first thing that we should note is that the amount of business a restaurant gets can be approximately modeled by a periodic function. This is because restaurants have opening and closing hours every day. As most people know restaurants tend to have busier hours around meal times. This may include but is not limited to early morning for breakfast, midday for lunch, and early to late evening for dinner. However, we see a similar cyclic pattern almost every day. Some days might have larger peaks than others but the same cyclic patterns persist.

The number of people ordering at a restaurant at any given time is determined by the type of restaurant, the time of day, the day of the week, as well as possibly the season of the or month of the year as well. For example, an ice cream shop is much more likely to get business in the summer than in the winter months. Our system will look at these features through various models to analyze, predict, and refine our estimates. Different models have different advantages and disadvantages, so the system will compare the performances of each model and pick accordingly.

In order to implement these models, we need to format our inputs with the features the algorithm will analyze. Our current structure is to have our input feature vector contain the hour, day, month, and year. Our output feature will be the number of items for the particular order. Since our output feature will only contain one value, we will repeat this process for each order on the menu. The hours are represented as an integer from 0 to 23, the days range from 0 to 30, the months range from 0 to 11, and the year is just represented as is.

$$x = [hour, day, month, year]$$

$$y = [number\ of\ items]$$

This first model we plan on using stems from regression analysis with sinusoids. Due to the cyclic trends of restaurants orders we believe that using a sum of sinusoids can lead to better prediction. The model stems from a fourier analysis of the 24-hour period of daily food consumption. To fit this multivariate model, we used this model function where b is a vector of our model parameters.

$$modelFunc = \sum_{i=1}^{length(x)} [b_{i,1} sin(x(:,i)/b_{i,2} + b_{i,3}) + b_{i,4} cos(x(:,i)/b_{i,5} + b_{i,6})] + b$$

To pick our initial parameter values, we simply guessed some random values that we believed would somewhat estimate the fitted line.

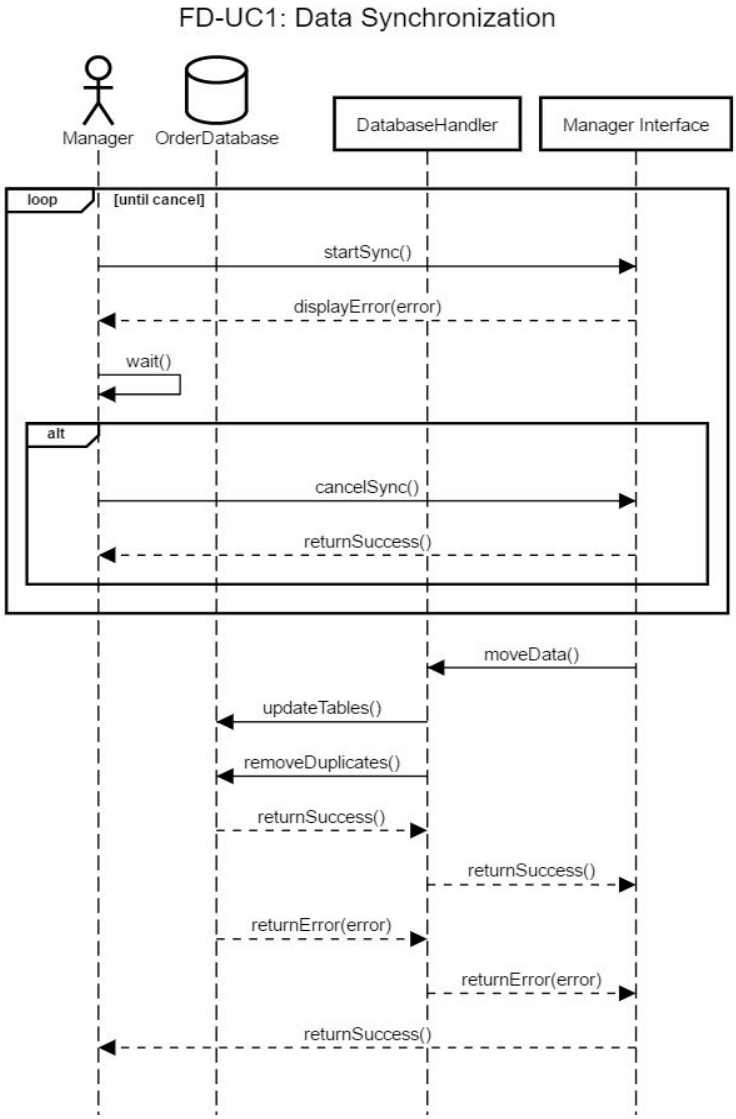
The second models attempts to model with regression using a polynomial function. This function may work better because it may be easier to account for the spikes which may occur as

a result of seasonal or weekly trends as opposed to the sinusoidal series model. To fit this polynomial model, we used this function instead where m is the complexity of our polynomial.

$$modelFunc = \sum_{i=1}^{length(x)} [\sum_{j=1}^{m} b_{i,j} x(:,i)^j] + b$$

7. Interaction Diagrams

All interaction diagrams are evolved from Report 1, part 3d: System sequence diagrams.

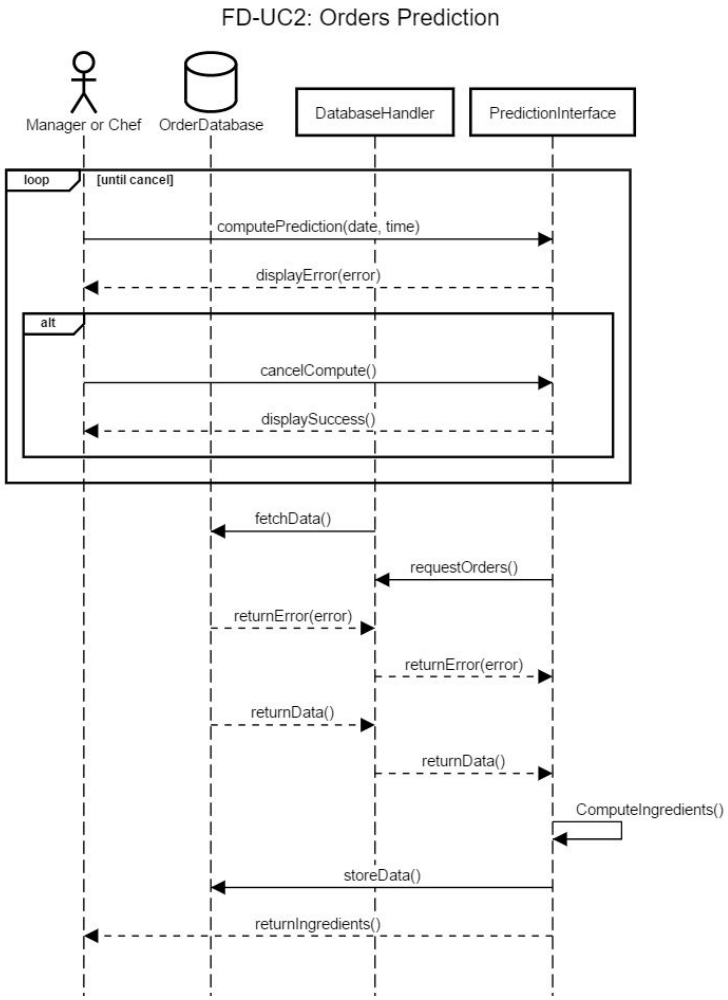


[3]

The manager can click a button on the manager's interface to manually update the prediction database with data from the current day. The system automatically performs this, but sometimes we want to update manually.

Once the user is successfully authorized the manager interface will attempt to update the tables in the order database. What this does is it asks the DatabaseHandler to move the previous current day's data into the order history to be included in the calculation of the prediction service. If nothing has gone wrong the manager's interface will return success.

On an error such as a database error, or authentication issue the interface will display an error and ask to retry. The Database handler ensures that the interfaces have low coupling.

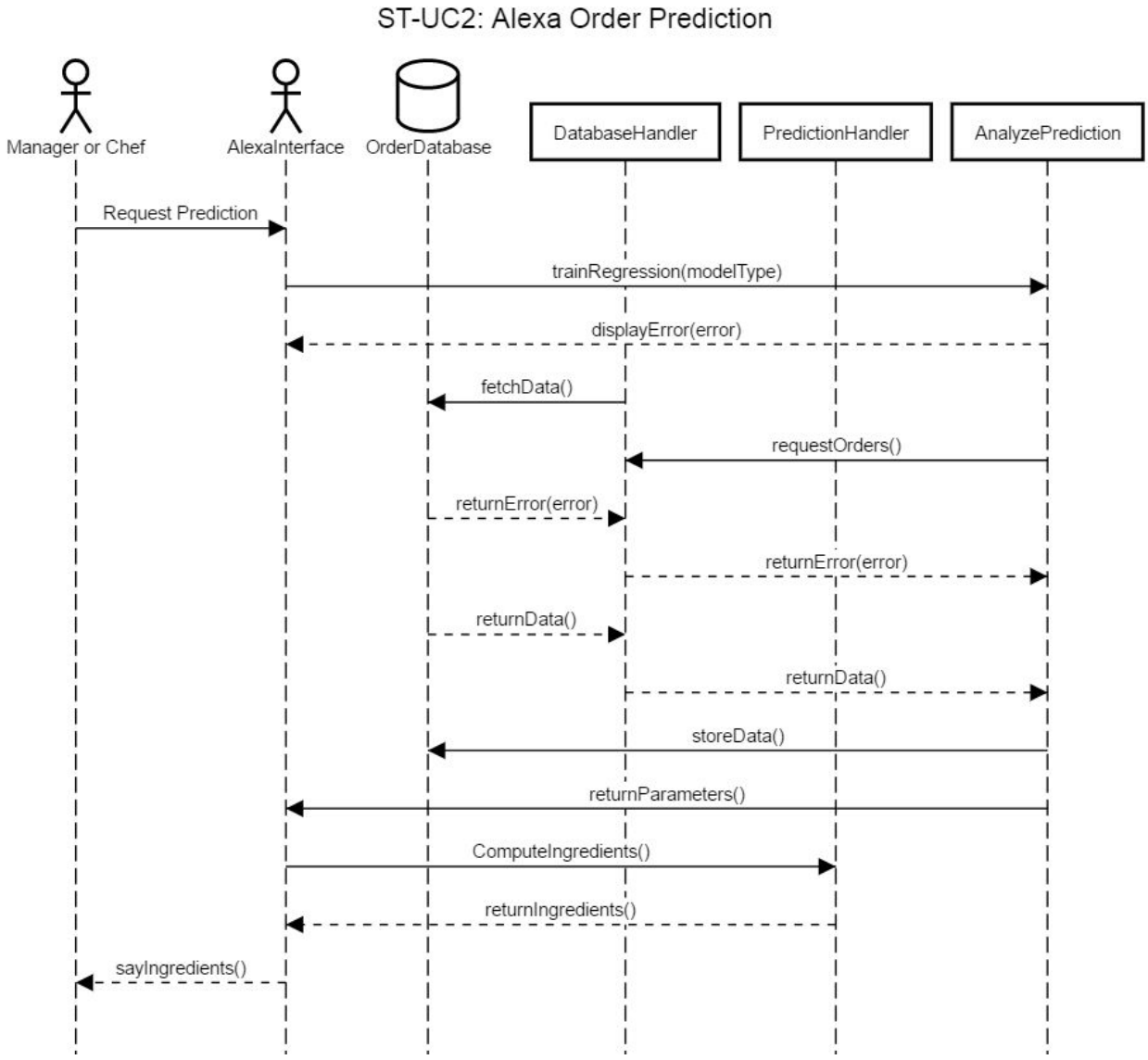


At the start of the day a manager or chef can open the prediction interface and compute the necessary ingredients that need to be prepared for the current day and time. When this interface is opened, if the model has not computed ingredients for the current day the prediction interface will first attempt to fetch the previous day's data and previous prediction data through the DatabaseHandler.

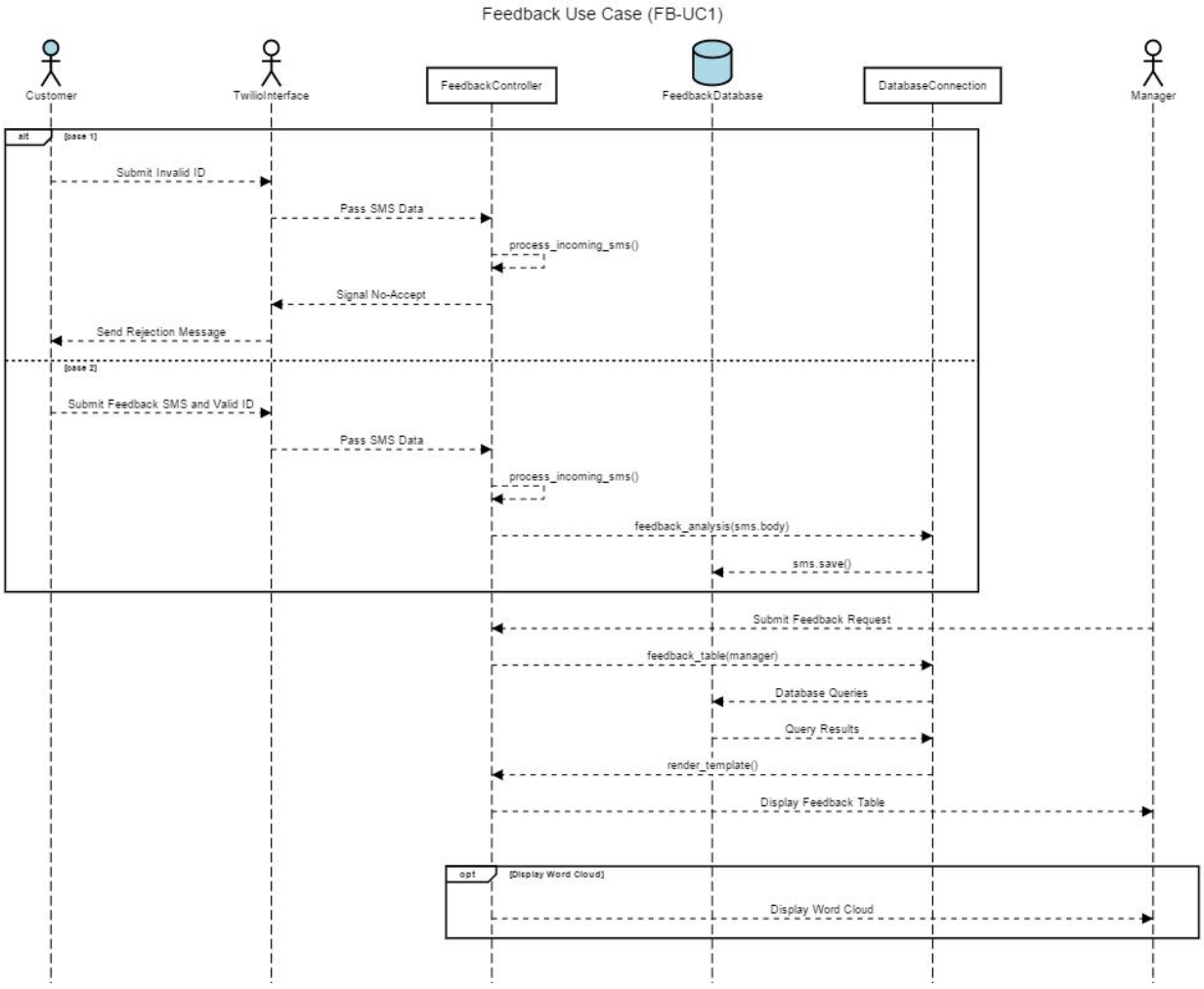
Once that happens the prediction interface will put the data through a model to compute the needed ingredients. The data will then be displayed on the prediction interface and also sent back to the order database to be stored.

On a database or time/date error the prediction interface will display the error generated and ask if you want to try again. Because the previous computations are stored in the database

this ensures that the prediction interface has high cohesion.



This use case is functionally the same as FD-UC2 but provides an interface through Alexa. Alexa is first prompted with a request and then Alexa, using the data it has been given, performs the prediction for us instead.



Customers will submit one text at a time. The feedback handler will determine receive the message that the customer submits and then return a confirmation message that says that the survey information has been received by the system.

After the system receives the feedback information, the database handler will take care of the analysis of the data as well as any requests that are made by the administrative portal.

Once the database handler receives any new information from the feedback handler, the model will then proceed on retrieving any relevant data that is in the database and find if the new data is notable in comparison to the old data. If there is data that is notable, the system will send the information to the administrative portal and display that information for the manager to see.

On the other end, the manager can request information from the feedback database to see how a certain subject is performing. The manager will request the information from the administrative portal which will request the information from the database handler.

The database handler will then retrieve the relevant information pertaining to the search query. The database handler will then return that information to the portal which will be displayed for the manager to view.

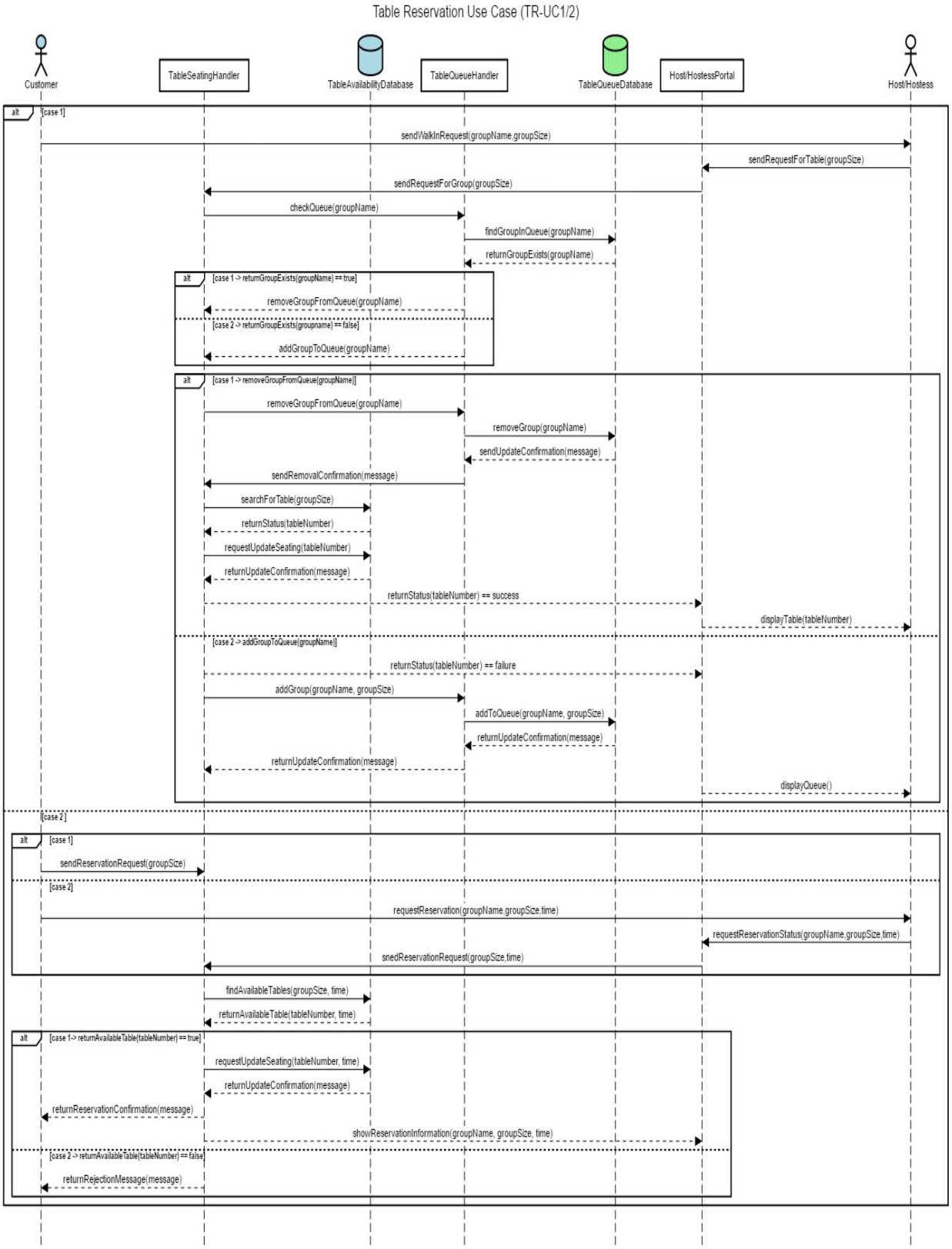


Table Reservation Use Case (TR-UC1/2)
Customer
TableSeatingHandler
TableAvailabilityDatabase
TableQueueHandler
TableQueueDatabase
Host/HostessPortal
Host/Hostess
alt
[case 1]
sendWalkInRequest(groupName,groupSize)
sendRequestForTable(groupSize)
sendRequestForGroup(groupSize)
checkQueue(groupName)
findGroupInQueue(groupName)
returnGroupExists(groupName)
alt
[case 1 -> returnGroupExists(groupName) == true]
removeGroupFromQueue(groupName)
[case 2 -> returnGroupExists(groupname) == false]
addGroupToQueue(groupName)
alt
[case 1 -> removeGroupFromQueue(groupName)]
removeGroupFromQueue(groupName)
removeGroup(groupName)
sendUpdateConfirmation(message)
sendRemovalConfirmation(message)
searchForTable(groupSize)
returnStatus(tableNumber)
requestUpdateSeating(tableNumber)
returnUpdateConfirmation(message)
returnStatus(tableNumber) == success
displayTable(tableNumber)
[case 2 -> addGroupToQueue(groupName)]
returnStatus(tableNumber) == failure
addGroup(groupName, groupSize)
addToQueue(groupName, groupSize)
returnUpdateConfirmation(message)
returnUpdateConfirmation(message)
displayQueue()
[case 2]
alt
[case 1]
sendReservationRequest(groupSize)
[case 2]
requestReservation(groupName,groupSize,time)
requestReservationStatus(groupName,groupSize,time)
snedReservationRequest(groupSize,time)
findAvailableTables(groupSize, time)
returnAvailableTable(tableNumber, time)
alt
[case 1-> returnAvailableTable(tableNumber) == true]
requestUpdateSeating(tableNumber, time)
returnUpdateConfirmation(message)
returnReservationConfirmation(message)
showReservationInformation(groupName, groupSize, time)
[case 2 -> returnAvailableTable(tableNumber) == false]
returnRejectionMessage(message)

The customer has three different forms of reserving a seat. The three forms are when the customer walks-in and request a table, customer makes a reservation through the online system, and customer makes a reservation through contacting the host/hostess and they input the reservation information. Starting with the walk-in customer, the host will input the information that the customer provides into the host/hostess interface where they will then be able to figure out if there is a table available or the customer will be placed into queue. The reservation system has two cases.

- Allows the customer to reserve through the online system,
- Allows customers to reserve through the host/hostess.

Both operations run in a similar manner where there is only one more communication path for the reservation through host/hostess. If the table if available during reservation time, they will be able to reserve the table, otherwise they will get a message saying that their reservation failed.

a. Design Patterns

Each of the use cases attempt to limit the number of tasks that each entity will try to handle at any given time. Given that the system is running in continuously in order to keep the databases updated in realtime, the workloads have to be distributed evenly and streamlined so there are no overlapping signals that would cause a conflict in information that is being sent between each handler and database. In order to achieve this, we followed the High Cohesion Principle which places the emphasis on trying to limit the number of computations to prevent a slow response time in any part of the systems.

To limit the number of signals being sent between the handlers and databases, we followed the Low Coupling Principle which means that we limited the number of connections that go between each handler and database. The database would go with one handler, and the handler will take care of at most two interactive interfaces used by the employee or customer. As a programmer, we should make the assumption that some customers might not be able to use the program properly. We limited the amount of information that the customer will provide to prevent a smaller chance of providing the incorrect data. As a result, the Expert Doer Principle allows the employee to handle more information than the customer.

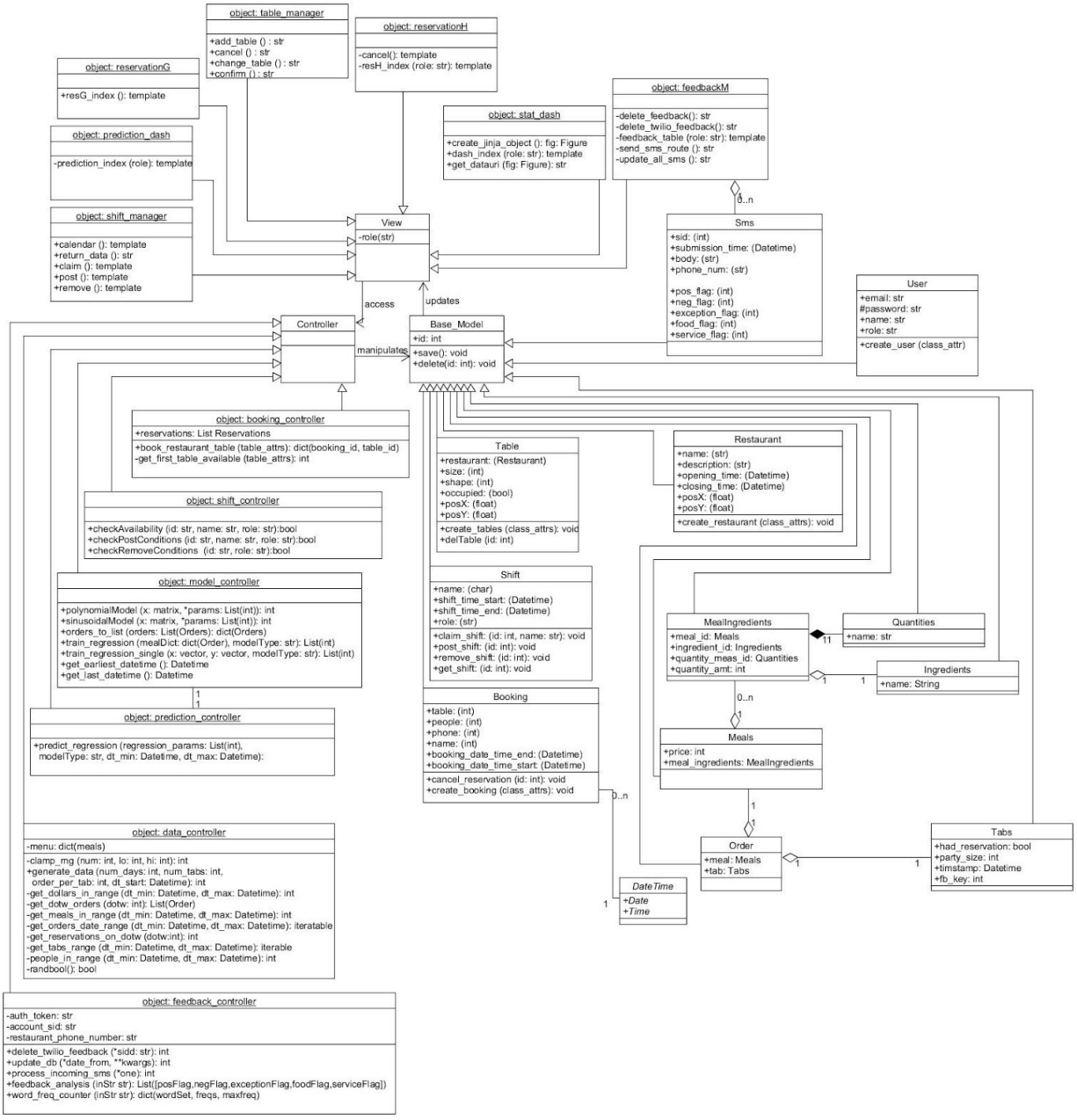
With these three principles implemented, the strength of each principle will bring forth a robust design that will prevent any miscommunications that will occur within the system.

Specifically in the feedback use case, we see that the feedbackController handles any manipulation of the feedback data. We will separate the TwilioInterface from the Controller, as well as the feedbackAnalysis and WordCounter. This allows for high cohesion and expert doer. The tasks are spread out within different domains, while each domain is an expert at the task it's intended to do.

We see the same in the prediction use case. Where there is a dedicated database connection, a concept that takes care of the prediction analysis, and a concept that tackles the interface with the user. This is both Expert Doer and High Cohesion. As long as there are not a large amount of tasks unbalanced, such as a lot of processing and little interface interaction, then the system follows a High Cohesion design pattern. There is often little interface interaction, and a decent amount of processing required, so the cohesion of this design can definitely be improved.

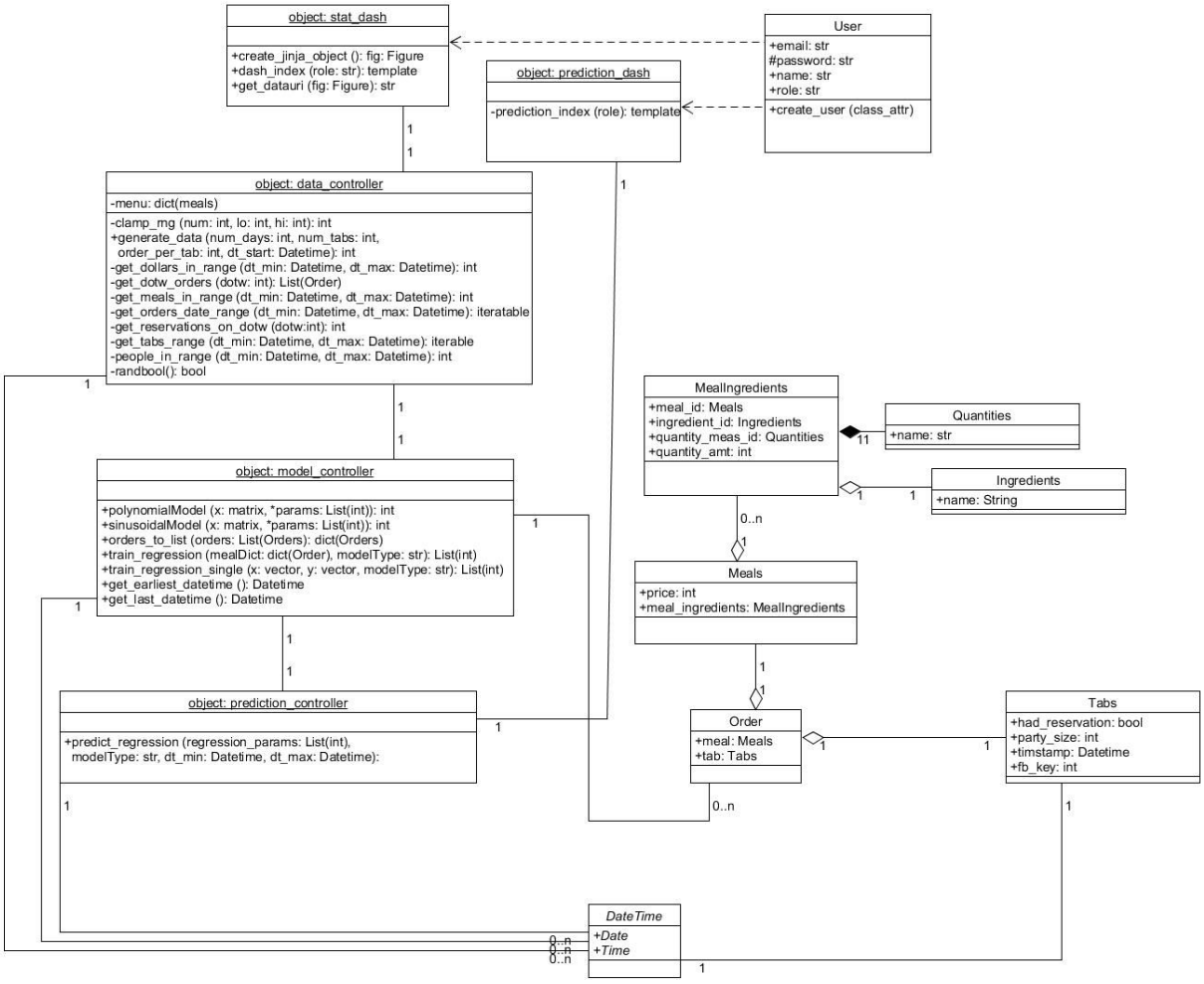
8. Class Diagram and Interface Specification

a. Class Diagram

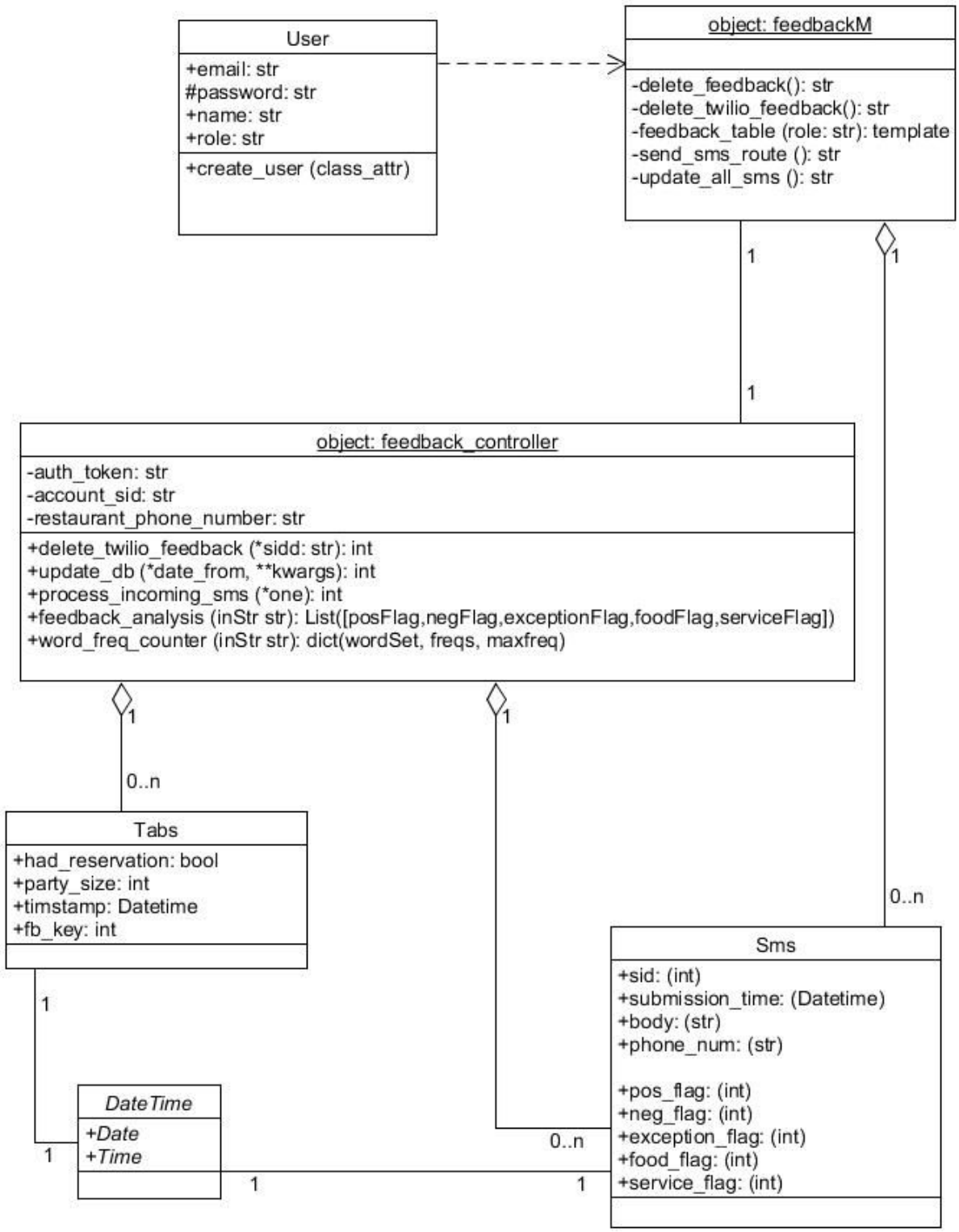


Overall Class Diagram:

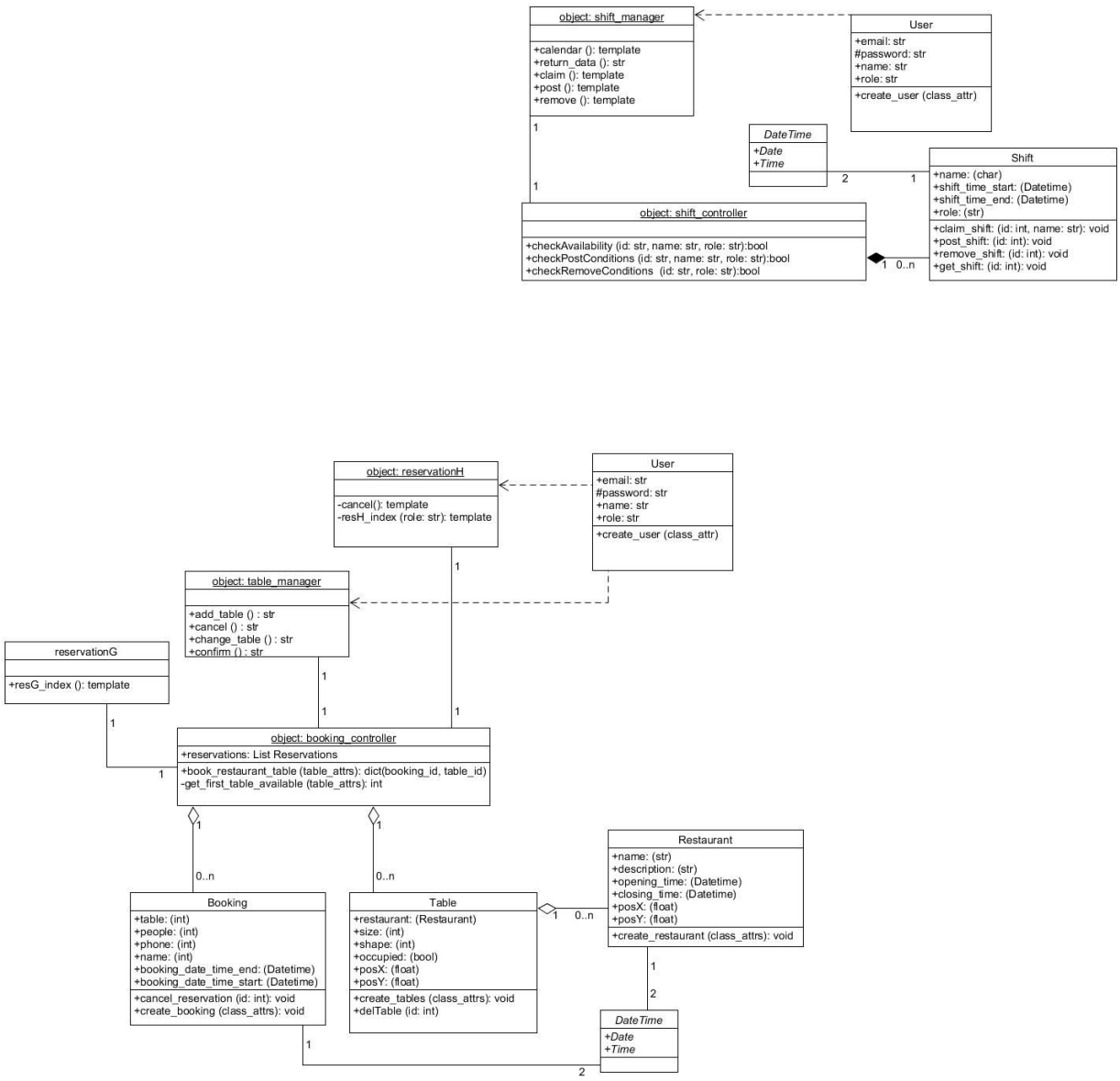
Not all interactions are shown, as that would complicate the diagram to the point where relationships would not be obvious. Relationship between Model, View, and Controller superclass show relationship patterns between their subclasses.



Statistics and Prediction Class Diagram



Feedback Class Diagram



Shift Management, Reservation & Table Management Class Diagram

[4] [5]

b. Data Types and Operation Signatures

Models

Base_model

A base model which sets up the database connection for all inherited classes

Attributes:

+id: (int) - Unique id that all models will have so that they can be uniquely referenced

Methods:

+save (): void - Saves a specified object to respective table of model
+delete (id: int): void - deletes a specific object by ID. if no ID specified, table will be deleted

Booking

Model for when customers book tables at the restaurant at a time period

Attributes:

+table: (int) - table reference
+people: (int) - number of people
+phone: (int) - phone number of who booked
+name: (int) - name of person who booked
+booking_date_time_end: (Datetime) - end time of booking
+booking_date_time_start: (Datetime) - start time of booking

Methods:

+cancel_reservation (id: int): void - Attempts to cancel a reservation given an ID
+create_booking (table: int,people: int , booking_date_time_start: Datetime, booking_date_time_end: Datetime, name: str, phone: str): void - Creates a new reservation

Tables

A restaurant object with properties about it’s closing time & opening time

Attributes:

+restaurant: (Restaurant) - restaurant reference
+size: (int) - The max number of guests a table can seat
+shape: (int) - An integer representing the shape of the table. 0 = Circle, 1 = Square, 2 = Rectangle, 3 = Vertical Rectangle
+occupied: (bool) - True if occupied, false if now

+posX: (float) - The relative x position of the table
+posY: (float) - The relative y position of the table

Methods:

+create_tables (restaurant: Restaurant, size: int, occupied: bool, posX: float, posY: float, shape: int): void- Creates a new table
+delTable (id: int): void - Deleted a table referenced by its id

Restaurant

A restaurant object with properties about it's closing time & opening time

Attributes:

+name: (str) - name of restaurant
+description: (str) - description of restaurant
+opening_time: (Datetime) - opening time of restaurant
+closing_time: (Datetime) - closing time of retaurant

Methods:

+create_restaurant (name: str, description: str, opening_time: Datetime ,closing_time: Datetime): void - creates a restaurant object with a description, name and opening/closing times

Sms

A model for SMS objects to be stored and analyzed. The many flags hold a range of values between -1 and 1

Attributes:

+sid: (int) - Unique id assigned to sms object in Twilio's cloud database
+submission_time: (Datetime) - time that the sms object was submitted to twilio.
+body: (str) - the text that represents the body of the sms message
+phone_num: (str) - the phone number of the sender
+pos_flag: (int) - the flag to indicate sms object body has positive keywords
+neg_flag: (int) - the flag to indicate sms object body has negative keywords
+exception_flag: (int) - the flag to indicate sms object body contains an exception
+food_flag: (int) - the flag to indicate sms object body has keywords related to food
+service_flag: (int) - the flag to indicate sms object body has keywords related to service

Methods: N/A

Ingredients

A table which maps ingredient names to ingredient ID's

Attributes:

+name: (str) - name of the ingredient

Methods: N/A

Meals

A model for restaurant meals

Attributes:

+price: (int) - price of the meal specified
+name: (str) - name of meal

Methods: N/A

Tabs

A model for a tab - i.e. a list of meals or items ordered at a given table.

Attributes:

+had_reservation: (bool) - if the party associated with this tab made a reservation
+party_size: (int) - size of party
+timestamp: (Datetime) - time of party's arrival
+fb_key: (str) - unique key given to each tab to validate their sms feedback

Methods: N/A

Quantities

A table mapping quantities names to quantity Id's

Attributes:

+name: (str) - name of the quantity

Methods: N/A

Orders

A model for storing every meal ordered and providing mappings to the tabs to meals

Attributes:

+meal: (Meals) - meals associated with an order
+tab: (Tab) - Tab object for each order

Methods: N/A

MealIngredients

A table mapping a meal to the ingredients used

Attributes:

+meal_id: (Meals) - Meal object being referenced
+ingredient_id: (Ingredients) - List of ingredients associated with this meal
+quantity_meas_id: (Quantities) - Quantity associated with this ingredient
+quantity_amt: (int) - amount of the quantity specified

Methods: N/A

User

A User model for who will be using the software. Users have different levels of access with different roles.Current active roles:host, admin, chef, cust, notanadmin

Attributes:

+email: (str) - The user email
#password: (str) - The password string - no need to hash beforehand
+name: (str) - name, doesn't have to be unique
+role: (str) - The user role. admin, manager, chef, host, etc..

Methods:

+create_user: (email: str, password: str, name: str, role: str) - Creates a new user, stores the password as a hash

Shifts

A model for keeping information about an employee’s work shift

Attributes:

+name: (char) - The name of the employee claiming the shift
+shift_time_start: (Datetime) - Starting time of shift
+shift_time_end: (Datetime) - Ending time of shift
+role: (str) - role of the user who makes the shift

Methods:

+claim_shift: (id: int, name: str): void - Attempts to claim a shift given an ID of shift
+post_shift: (id: int): void - Attempts to post a shift given an ID of shift
+remove_shift: (id: int): void - Attempts to remove a shift from user’s schedule given ID of shift
+get_shift: (id: int): void - Attempts to retrieve a shift to a user’s schedule given ID of shift

Controllers

booking_controller

This module and its functions handle the logic for table reservations for the restaurant.
It also handles moving data in and out of our database via the via our application models.

Attributes: N/A

Methods:

+book_restaurant_table (restaurant: int, booking_date_time: Datetime, people: int, name: string, phone: string, minutes_slot: int): dict(booking_id, table_id)
This method uses get_first_table_available to get the first table available, then it creates a Booking on the database.

-get_first_table_available (restaurant: int, booking_date_time: Datetime, people: int, minutes_slot: int): int
This method returns the first available table of a restaurant, given a specific number of people and a booking date/time.

data_controller

Used to get relevant data for the model controller and statistics dashboard

Attributes:

-menu: dict(meals) - meals that contain the ingredients, price, etc

Methods:

-clamp_rng (num: int, lo: int, hi: int): int
Forces a number to fall within min/max range by doing lo + (num % (hi-lo)) Hi and lo should always be positive. If num is < 0 then we take its absolute value and clamp that.

+generate_data (num_days: int, num_tabs: int, order_per_tab: int, dt_start: Datetime): int
Generates and stores data for the models in chefboyrd.models.statistics

-get_dollars_in_range (dt_min: Datetime, dt_max: Datetime): int
Returns the total sum of revenue from a range of dates

-get_dotw_orders (dotw: int): List(Order)
Gets all orders on a given day of the week

-get_meals_in_range (dt_min: Datetime, dt_max: Datetime): int
Returns the total sum of meals from a range of datetimes

-get_orders_date_range (dt_min: Datetime, dt_max: Datetime): iterable***

Gets a range of orders from one datetime to another datetime joined on Tabs.

-get_reservations_on_dotw (dotw:int): int
Gets the number of reservations on a given day of the week

-get_tabs_range (dt_min: Datetime, dt_max: Datetime): iterable***
Gets a range of tabs from one datetime to another datetime

-people_in_range (dt_min: Datetime, dt_max: Datetime): int
Returns the total number of people served in a range of dates

-randbool(): bool
Return a random boolean (approx 50/50)

***(set object as defined by peewee, our Object Relational Model library)

feedback_controller

Houses all the functions to add or delete sms objects into the database. Includes the feedback analysis functions

Attributes:

-auth_token: str - authorization token required for using Twilio API. Should be added to config file
-account_sid: str - account id for use with Twilio. Should be added to config file
-restaurant_phone_number: str - restaurant's phone number to text feedback to

Methods:

+delete_twilio_feedback (*sidd: str): int
Wipes the message with the specified sid(s) on Twilio. Will display response codes.
+update_db (*date_from, **kwargs): int
Updates the sms in the database starting from the date_from specified (at time midnight). no param = updates the sms feedback in database with all message entries
date_from (Date): a specified date, where we update db with sms sent after this date
update_from (str): an optional argument. This should be "test" if messages are not coming from twilio, but from the test_fb_data file

+process_incoming_sms (*one): int
Updates SMS table in database with the incoming SMS. Checks for the unique key to invalidate SMS or keep it. Only for processing SMS in real time. Optional argument if, present if only one sms needs to be processed, otherwise all will be processed.

+feedback_analysis (inStr str):
List([posFlag,negFlag,exceptionFlag,foodFlag,serviceFlag])

Determines categories via keyword of input string based on word content.

+word_freq_counter (inStr str): dict(wordSet, freqs, maxfreq)
Determines frequency of each word in input string given an input string containing words separated by spaces or non-apostrophe punctuation.

model_controller

This is a preprocessor for the prediction_controller. Given data in the form of our local models, it converts it into numbers usable by the prediction controller.

Attributes: N/A

Methods:

+polynomialModel (x: matrix, *params: List(int)): int
Args:x = input matrix where the features are rows and each predictor(data point) is a column.
Returns: params = the parameters for the model

+sinusoidalModel (x: matrix, *params: List(int)): int
This is the sinusoidal model that will be fit by our algorithm.
Args: x = input matrix where the features are rows and each predictor(data point) is a column
params = the parameters for the model
Returns: sum = the output vector where each index is a predictor

+orders_to_list (orders: List(Orders): dict(Orders)
Converts peewee query result set to a list of dictionary of lists
Args: accepts a list of orders made from a query for peewee

+train_regression (mealDict: dict(Order), modelType: str): List(int)
This trains all of the meals and fits their data to their own set of parameters
Args:
mealDict = The meal dictionary containing order data created by orders_to_list
modelType = the type of model to use(sinusoidal, polynomial)
Returns:
mealsParams = a dictionary that contains the parameters for each corresponding meal

+train_regression_single (x: vector, y: vector, modelType: str): List(int)
Trains a single meal and fits it to a model to find its parameters
Args:
x = The input vector(which is composed of rows 0 to 3 in the vector)
y = The expected output vector(the last row)
modelType = the type of model to use
Returns:
params = a list of parameters

+get_earliest_datetime (): Datetime
Finds the minimum datetime in the orders database

+get_last_datetime (): Datetime
Finds the maximum datetime in the orders database

prediction_controller

This controller performs our machine learning algorithms using nonlinear regression fits on the data.

Attributes: N/A

Methods:

+predict_regression (regression_params: List(int), modelType: str, dt_min: Datetime, dt_max: Datetime):
Predicts the usage of ingredients according to our regression model
Args:
regression_params - the parameters for your model
modelType - the type of model you're using
date range - the two dates that you want to predict the regression for
Returns:
mealUsage - a dictionary of meals with associated usage amounts

shift_controller

This controller houses functions related to the shift management portal

Attributes: N/A

Methods:

+checkAvailability (id: str, name: str, role: str):bool
This method checks the availability of the shifts and makes sure that there are no scheduling overlaps in terms of duration.
Args:
id: the id of the shift that will be in analysis
name: name of the user that is attempting to claim the shift
role: role of the user that is claiming the shift
Returns:
True: if there are no scheduling conflicts
False: if the worker in question already has a schedule arranged

+checkPostConditions (id: str, name: str, role: str):bool
This method checks the ability for the user to post their shift and makes sure that
the user is not posting another user's shift. The only exceptions are for admins and managers.
Args:
id: the id of the shift that will be in analysis
name: name of the user that is attempting to claim the shift

role: role of the user that is claiming the shift
Returns:
True: if the user or the role has the proper influence
False: if the user or role does not meet the criteria

+checkRemoveConditions (id: str, role: str):bool
This method checks the ability for the user to remove the shift and makes sure that
the user is not removing shifts without the proper permissions. The only exceptions are for admins
and managers.
Args:
id: the id of the shift that will be in analysis
role: role of the user that is claiming the shift
Returns:
True: if the user or the role has the proper influence
False: if the user or role does not meet the criteria

Views

feedbackM

feedbackM This view is specifically for the administrative staff's management of the feedback

Attributes: N/A

Methods:

+delete_feedback(): str
Calls the delete_feedback function, returns a message confirming # of feedback entries deleted. Returns: Confirmation string

+delete_twilio_feedback(): str
wipe all message history on twilio. Only needs to be called once to clear data. Returns: Confirmation string

+feedback_table (role: str): template
By default displays a webpage for user to make feedback DB query. Display a table of feedback sent in during a specified date-time range. Also, depending on whether category flags are specified, this page will only display. Returns: The template to display with the appropriate parameters table: table of feedback to display form: form that specifies the query instructions.

+send_sms_route (): str
This is the directory we need to configure twilio for. When Twilio makes a POST request, db will be updated with new sms messages from today :returns: Confirmation string

+update_all_sms (): str
Update sms data from external Twilio database
Returns: Confirmation string

prediction_dash

Prediction dashboard for the manager interface. Will allow you to predict ingredient usage between two chosen dates

Attributes: N/A

Methods:

+prediction_index (role): template
Renders the index page of the prediction page

reservationH

Reservation dashboard for the manager interface

Attributes: N/A

Methods:

+cancel(): template
This handles when a user needs to cancel a reservation.

-resH_index (role: str): template
Renders the index page of the reservation page

reservationG

Reservation dashboard for the customer's interface

Attributes: N/A

Methods:

+resG_index (): template
Renders the index page of the reservation page

stat_dash

Statistics dashboard for the manager interface
Will be able to render dashboards which include statistics from the database of the Point of sale system and other data systems for the business.

Attributes: N/A

Methods:

+create_jinja_object (): fig: Figure***
Creates graph to display the statistics information

+dash_index (role: str): template
Renders the index page of the dashboards

+get_datauri (fig: Figure***): str
Build an image data URI

***Figure from matlibplot library

table_manager

Table dashboard for the manager interface

Attributes: N/A
Methods:

+add_table () : str
This handles when a user adds a table to the layout.

+cancel () : str
This handles when a user needs to cancel a reservation.

+change_table () : str
This handles when a user needs to change the status of a table.

+confirm () : str
This handles when a user needs to confirm a reservation.

+del_table () : str
This handles when a user adds a table to the layout.

+table_manager_index (role: str) : template
Renders the index page of the table management page

+update_table () : str
This handles when we need to change the position of a table.

shift_manager

Shift management dashboard for the manager interface

Attributes: N/A
Methods:

+calendar (): template
Renders the index page of the shift management page

+return_data (): str

Returns shift data based on the start and end time specified in the HTTP request

+claim (): template
This handles when the user needs to claim a shift. Returns to a confirmation template

+post (): template
This handles when the user needs to post a shift. Returns to a confirmation template

+remove (): template
This handles when the user needs to remove a shift. Returns to a confirmation template

---

c. Traceability Matrix

The interaction diagram objects are not exactly the same as the class diagram objects because including each class in the interaction diagram would have complicated the diagram and detracted from the main idea. The mappings from the objects in the interaction diagrams to the class diagram:

Concepts \ Classes	data_controller	booking_controller	feedback_controller	model_controller	prediction_controller	shift_controller	base_model	customers	reservation	sms	statistics	tables	user	shifts	feedbackM	prediction_dash	reservationG	reservationH	stat_dash	table_manager	shift_manager
DatabaseConnection							x														
StatisticsDisplay																			x		
StatisticsController	x																				
StatisticsQueryHelper																			x		
PredictionDisplay															x						
PredictionHandler					x																
AlexaInterface	x				x																
AnalyzePrediction				x	x																
FeedbackQueryHelper														x							
FeedbackDisplay														x							
TwilioInterface			x																		
FeedbackController			x																		
FeedbackAnalyzer			x																		
WordCounter			x																		
TabsKeys			x																		
ReservationForm																	x				
ReservationHandler		x																			
ReservationDisplay																		x			
ReservationChecker		x																			
TableGUI																			x		
TableChecker		x																			
ShiftDisplay																				x	
ShiftChecker						x															
ShiftManager						x															
ShiftActions						x															

Concepts:

DatabaseConnection

The database connection is mapped to the base_model. The base_model is a superclass of all the other models, customers, reservation, sms, statistics, tables, user, shifts. This base_mopdel has a method to communicate with the database that all the models will inherit. The model_controller is responsible for auto-generating data, so it can also be mapped to the DatabaseConnection

StatisticsDisplay

Stat_dash is responsible for displaying all statistics-related information to manager and chef.

StatisticsController

Maps to data_controller, which organizes the information received from the database into categories, day of the week, per hour, etc.

StatisticsQueryHelper

Stat_dash also has form objects the user will be able to submit

PredictionDisplay

Prediction_dash displays all prediction-related information to the manager and chef.

PredictionHandler

Prediction information is analyzed and handled by prediction_controller
feedbackM is the view of the feedback use case

AlexaInterface

Prediction_controller and data_controller map to this concept, the alexa interface must get the statistics information and the prediction information from these controllers.

AnalyzePrediction

Prediction_controller is the controller that has the analysis functions for prediction use case.

FeedbackQueryHelper

feedbackM is the view , where use can submit forms for which feedback to see in the display

FeedbackDisplay

feedbackM is the view of the feedback use case. All feedback-related information here

Feedback_controller houses all functions related to interfacing with twilio, analyzing the feedback given into categories and getting a word count for the word cloud in the display. This also checks for the correct key in the SMS and checks to see if it matches the keys valid in the tab.

ReservationForm

Form is accessible to the customer via reservationG, which is the user-facing view.

ReservationHandler & ReservationChecker

Reservations are handled by the booking_controller, where it can check if the booking is valid, and then update the database with it, and send the data to display

ReservationDisplay

reservationH, the host's view for seeing what reservations are in the database

TableGUI

Table_manager, where the user can see which tables are occupied and the status of the tables.

TableChecker

Booking_controller, where it checks to see if table selected are valid

ShiftDisplay, ShiftActions

Shift_manager, the view where users can see their respective shifts. In this display the user can click buttons to add, drop, or cover shifts. These actions are interpreted and sent to the shift_controller

ShiftChecker, ShiftManager

Shift_controller is responsible for checking if shifts are valid, adding them to the database or removing them as necessary

d. Design Patterns

Based on the MVC - style that we implemented, we organized classes to separate the functions that have the information (Model), the functions that do the analysis and change the information(Controller) and the functions that receive commands from the User (View).

The advantage of using the MVC pattern is the short communication chain. For a feature there should be at most 3 or so different classes interacting to complete a task. And it sticks to the expert doer principle, as each class a well-defined role within the feature problem domain. Though if a feature implements heavy computation, it may unbalance the workload across objects and break the low coupling principle.

For a majority of our classes, we stuck with high cohesion where a class would be responsible for one main function. This function would be encapsulated in this class so other classes interacting with it does not need to know anything about the class operations.

We applied this for most of our classes, with feedback_controller being one of the exceptions. This is an example of low cohesion being used. The feedback_controller class is responsible for both maintaining an interface with Twilio and carrying out the feedback analysis.

e. Object Constraint Language Contracts

Notes:

```
self.select().where(CLASS.id == self.id).exist()
becomes
self.exist()

self.attrs
becomes
self.attr1, self.attr2, self.attr3 ... self.attrn
```

```
context BaseModel.save(): void
inv:
pre:    self.exist() == False
post:   self.exist() == True
    self.id.is_unique() == True


context BaseModel.delete(id: int): void
inv:
pre:    self.exist() == True
post:   self.exist() == False
    If (id == NULL): self.select().exist() == False #table deleted


context Booking.cancel_reservation(id: int): void
inv:
pre:    id != NULL
    self.exist() == True
post:   self.attrs == NULL
    self.exist() == False


context Booking.create_booking (table: int,people: int , booking_date_time_start:
Datetime, booking_date_time_end: Datetime, name: str, phone: str): void
inv:
pre:    table > 0
    people > 0
    booking_date_time_start < datetime.now()
    booking_date_time_end > datetime.now()
    name != NULL && name.len > 0
    phone != NULL && phone.len >0
    id != NULL
```

```
    self.exist() == False
post:   self.attrs == NULL
    self.exist() == True


context Tables.create_tables (restaurant: Restaurant, size: int, occupied: bool,
posX:  float, posY: float, shape: int): void
inv:
pre:    restaurant != NULL
    size > 0
    occupied == false
    posX >= 0
    posY >= 0
    shape > 0 && shape < 5
post:   self.exist() == True


context Tables.delTable (id: int): void - Deleted a table referenced by its id
inv:
pre:    id != NULL
    self.exist() == True
post:   self.attrs == NULL
    self.exist() == False


context Restaurant.create_restaurant (name: str, description: str, opening_time:
Datetime ,closing_time: Datetime): void
inv:
pre:    name != NULL
    name.len > 0
    description != NULL
    description.len > 0
    opening_time != NULL
    closing_time != NULL
post:   self.exist() == True


context User.create_user: (email: str, password: str, name: str, role: str) -
inv:
pre:    email != NULL
    email.len > 0
    User.select().where(User.email == email) == False
    password != NULL
    password.len > 0
    name != NULL
    name.len > 0
    role != NULL
    role.len > 0
```

```
post:   self.exist() == True


context Shift.claim_shift: (id: int, name: str): void
inv:    self.id == id
    self.exist()
    User_in_context.name == name
    self.shift_time_start not in User.shift
    self.shift_time_end not in User.shift
pre:    self.id == id
    User.select().where(User.name == name) == True
post:   self.name == name


context Shift.post_shift: (id: int): void
inv:    self.id == id
pre:    self.name == NULL
post:   User_in_context.name != self.name


context Shift.remove_shift: (id: int): void
inv:    self.id == id
pre:    shift.exist() in User.shift
post:   shift.exist() not in User.shift


context Shift.get_shift: (id: int): void
inv:    self.id == id
    Shift.exist()
pre:
post:


context booking_controller.book_restaurant_table (restaurant: int, booking_date_time:
Datetime, people: int, name: string, phone: string, minutes_slot: int):
dict(booking_id, table_id)
inv:
pre:    restaurant > 0
    booking_date_time > datetime.now()
    people > 0
    name != NULL
    name.len > 0
    phone != NULL
    phone.len > 0
    Table.select().where(Table.name == name) == False
post:   Table.select().where(Table.name == name) == True
    Booking.exist(booking_id)
    Table.exist(table_id)
```

```
context data_controller.generate_data (num_days: int, num_tabs: int, order_per_tab:
int, dt_start: Datetime): int
inv:
pre:    num_days > 0
    num_tabs > 0
    order_per_tab > 0
    dt_start < datetime.now()
post:   Tabs.select() == True
    Orders.select() == True
    Ingredients.select() == True
    MealIngredients.select() == True


context feedback_controller.delete_twilio_feedback (*sidd: str): int
inv:
pre:    if *sidd != NULL: *sidd.len > 0
post:   TwilioRestClient.messages.list() == NULL
    Sms.select().where(Sms.sid in *sidd) == NULL


context feedback_controller.process_incoming_sms (*one): int
inv:
pre:    if *one == 1: TwilioRestClient.messages.list() != NULL
    Tabs.select().fb_key.exist() == True
post:   if fb_key: Tabs.select().fb_key == "~~~~~~~~"
    Sms.exist() == True


context prediction_controller.predict_regression (regression_params: List(int),
modelType: str, dt_min: Datetime, dt_max: Datetime):
inv:    Tabs.select() > 0
    Orders.select() > 0
pre:    dt_min > datetime.now()
    dt_max > datetime.now() && dt_max > dt_min
    regression_parama != NULL
post:

context shift_controller.checkAvailability (id: str, name: str, role: str):bool
inv:    Shift.isAvailable == True
pre:    id > 0
    name != NULL
    role != NULL
post:
```

9. System Architecture and System Design

a. Architectural Styles

Communication

Using service-oriented architecture(SOA) allows us to create applications and software with known services. We will be using microservices as an implementation of SOA to pass messages and data between applications. Each of our subproblems can be defined as an individual service rather than the entire project as one service. The advantage of this is that each microservice can be built independently of each other, and do not rely on other pages or controllers. If we add or remove services, the existing services would work the same way. Developing in this style will allow for a quick and modular implementation of our service.

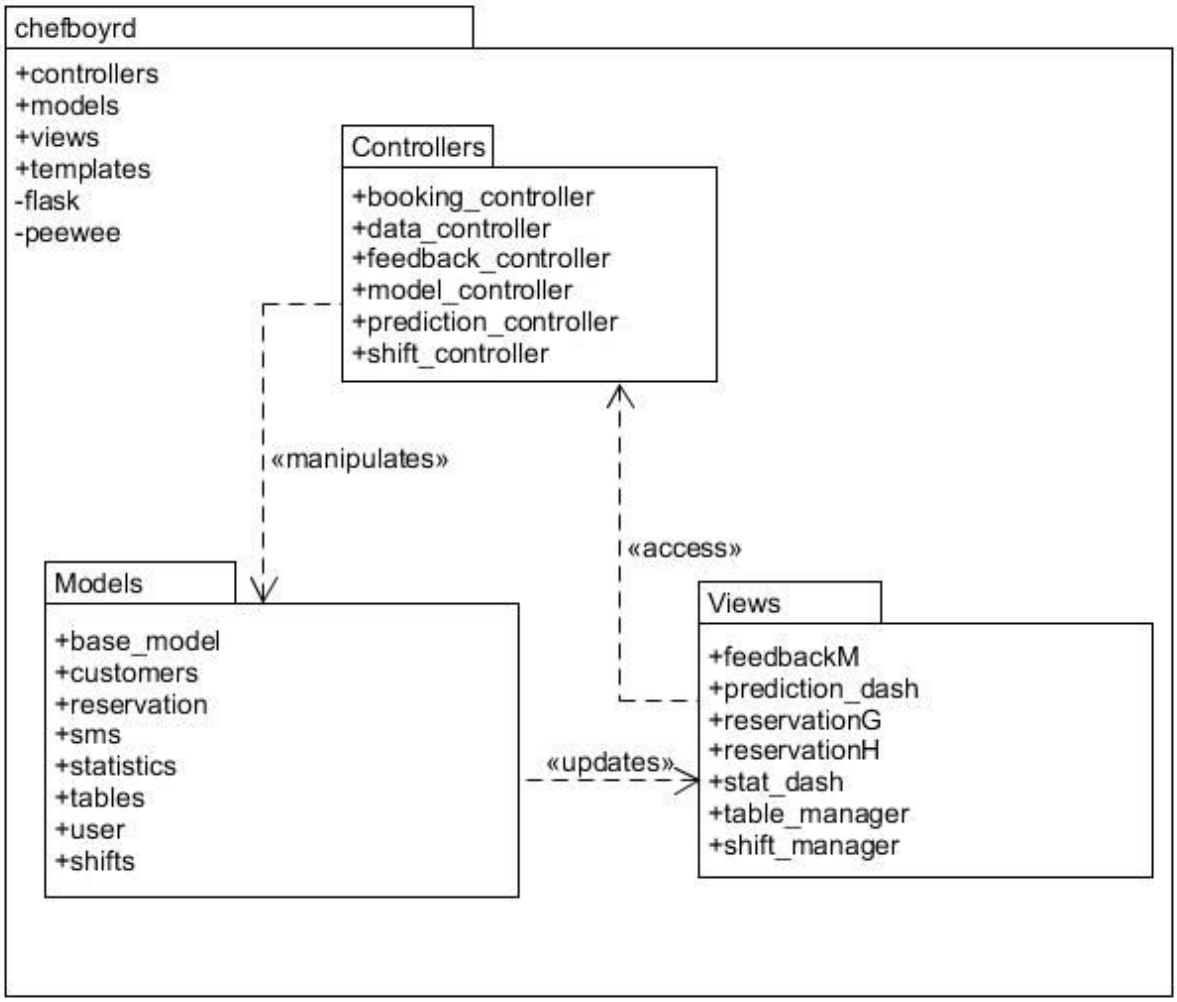
Deployment

We have decided to use a client-server model for our deployment style. In this method the client and server are distinct applications which communicate with each other to solve our problem. In our case the client would be a web browser. The web browser would make requests to the server, and the client would use the interface given by the server to make further requests to the server. With this method of client/server where a web browser is the client we can deploy on multiple platforms and have the interface displayed relatively unchanged opposed to if we developed an app for every device.

Structure

We are employing a Model-View-Controller(MVC) architectural style. In this model the responsibilities are split horizontally into model, view, and controller components. The model component is the central piece which manages data loading and storing of data. The view component is how the resulting data from the model component is displayed to the client. The third component, controller accepts user input, and routes the data to the appropriate controller which handles the logic of our problem domain.

b. Identifying Subsystems



Using a (almost) model-view-controller architectural style, our subsystems. Templated in view will allow users to call controllers to make modifications to models, which will update back into the views.

c. Mapping Subsystems to Hardware

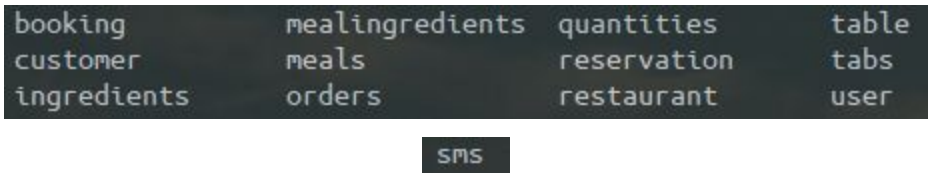
Because we are using a client-server deployment style our services will need to run on multiple devices. The server will be run on the central computer that the restaurant decides. This server will be the main resource for all the clients, which includes application data and the main database. The client side will be whatever device is needed for each subproblem. Our website is specifically designed to be mobile-friendly and tablet-friendly, so that employees, customers, or the administrator may access it from any device. Employees use their tablets which will load all data over the web browser provided by the server. Customers can also load the interface, such as

the reservation interface through the web browser as well. Because everything is served via web this makes it easy to use basic authentication and security practices to ensure that each role receives the correct interface. The server will be hosted on a local machine to the restaurant. The local machine will host all of the data, and databases.

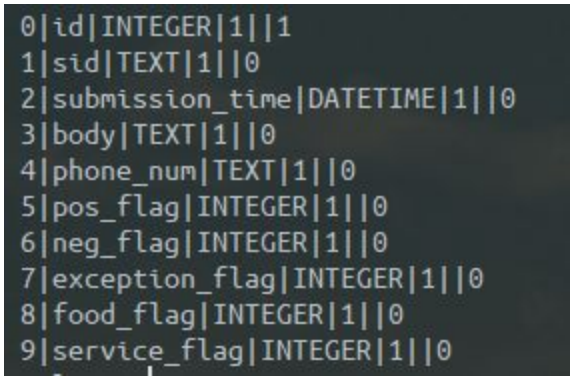
d. Persistent Data Storage

Our system needs to store data for long periods of time to satisfy our subproblems, so persistent data storage is needed here. Everything in our data storage will be stored in databases, this includes order history, inventory, menus, shift information, and our prediction data. Each of these elements will be stored in a separate table in the database, or a separate database altogether. All of this data will be stored using a SQL database. Our architecture affords us the freedom to use any one of MySQL, Postgres, SQLite. We will utilize an ORM (object-relational mapping) layer library called peewee. Peewee stands as a middleware between access to data objects and the actual database. It allows us to use object models within our code and maps them to table relations. The library also abstracts the database connection and querying process to create safer queries. This way our application is protected from SQL injection attacks and our written code is much simpler manner.

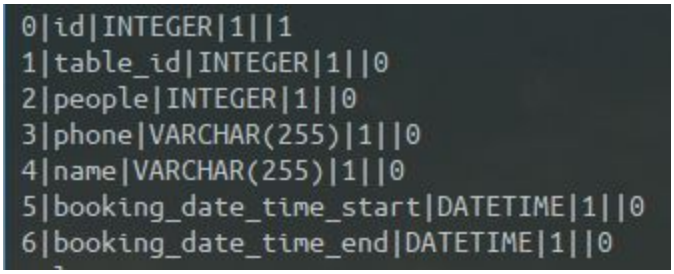
Database Tables



SMS Table



Booking Table




```
CREATE TABLE "booking" ("id" INTEGER NOT NULL PRIMARY KEY, "table_id" INTEGER NOT NULL,
"people" INTEGER NOT NULL, "phone" VARCHAR(255) NOT NULL, "name" VARCHAR(255) NOT NULL,
"booking_date_time_start" DATETIME NOT NULL, "booking_date_time_end" DATETIME NOT NULL,
                FOREIGN KEY ("table_id") REFERENCES "table" ("id"));
              CREATE INDEX "booking_table_id" ON "booking" ("table_id");
```

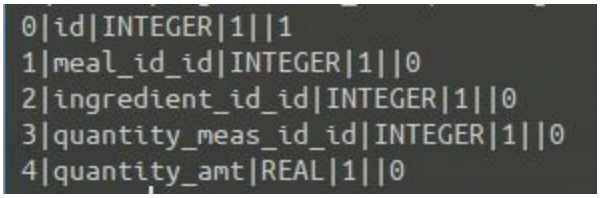
Customer Table


```
CREATE TABLE "customer" ("id" INTEGER NOT NULL PRIMARY KEY, "name" VARCHAR(255) NOT NULL);
```

Ingredients Table

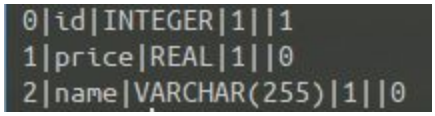

```
CREATE TABLE "ingredients" ("id" INTEGER NOT NULL PRIMARY KEY, "name" VARCHAR(255) NOT NULL);
            CREATE UNIQUE INDEX "ingredients_name" ON "ingredients" ("name");
```

MealIngredients Table




```
CREATE TABLE "mealingredients" ("id" INTEGER NOT NULL PRIMARY KEY, "meal_id_id" INTEGER NOT
    NULL, "ingredient_id_id" INTEGER NOT NULL, "quantity_meas_id_id" INTEGER NOT NULL,
 "quantity_amt" REAL NOT NULL, FOREIGN KEY ("meal_id_id") REFERENCES "meals" ("id"), FOREIGN
KEY ("ingredient_id_id") REFERENCES "ingredients" ("id"), FOREIGN KEY ("quantity_meas_id_id")
                            REFERENCES "quantities" ("id"));
      CREATE INDEX "mealingredients_meal_id_id" ON "mealingredients" ("meal_id_id");
 CREATE INDEX "mealingredients_ingredient_id_id" ON "mealingredients" ("ingredient_id_id");
         CREATE INDEX "mealingredients_quantity_meas_id_id" ON "mealingredients"
                                ("quantity_meas_id_id");
```

Meals Table




```
CREATE TABLE "meals" ("id" INTEGER NOT NULL PRIMARY KEY, "price" REAL NOT NULL, "name"
VARCHAR(255) NOT NULL);
CREATE UNIQUE INDEX "meals_name" ON "meals" ("name");
```

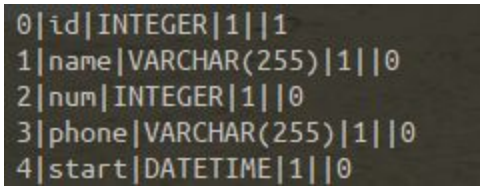
Orders Table


```
CREATE TABLE "orders" ("id" INTEGER NOT NULL PRIMARY KEY, "meal_id" INTEGER NOT NULL, "tab_id"
INTEGER NOT NULL, FOREIGN KEY ("meal_id") REFERENCES "meals" ("id"), FOREIGN KEY ("tab_id")
REFERENCES "tabs" ("id"));
CREATE INDEX "orders_meal_id" ON "orders" ("meal_id");
CREATE INDEX "orders_tab_id" ON "orders" ("tab_id");
```

Quantities Table

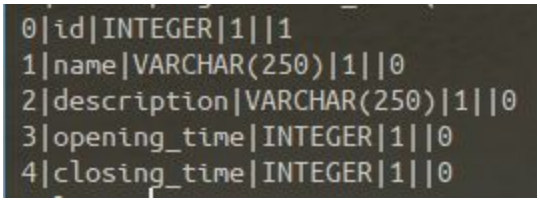

```
CREATE TABLE "quantities" ("id" INTEGER NOT NULL PRIMARY KEY, "name" VARCHAR(255) NOT NULL);
CREATE UNIQUE INDEX "quantities_name" ON "quantities" ("name");
```

Reservation Table



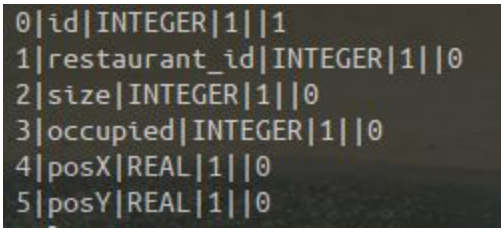

```
CREATE TABLE "reservation" ("id" INTEGER NOT NULL PRIMARY KEY, "name" VARCHAR(255) NOT NULL,
"num" INTEGER NOT NULL, "phone" VARCHAR(255) NOT NULL, "start" DATETIME NOT NULL);
```

Restaurant Table



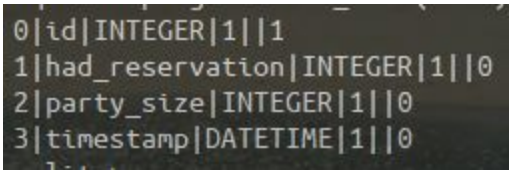
```
CREATE TABLE "restaurant" ("id" INTEGER NOT NULL PRIMARY KEY, "name" VARCHAR(250) NOT NULL,
"description" VARCHAR(250) NOT NULL, "opening_time" INTEGER NOT NULL, "closing_time" INTEGER
NOT NULL);
```

Tables Table



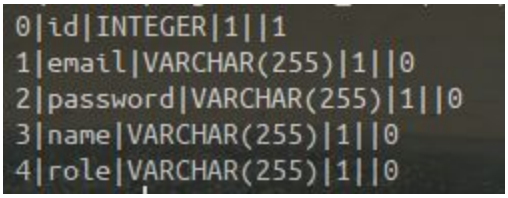

```
CREATE TABLE "tables" ("id" INTEGER NOT NULL PRIMARY KEY, "restaurant_id" INTEGER NOT NULL,
"size" INTEGER NOT NULL, "occupied" INTEGER NOT NULL, "posX" REAL NOT NULL, "posY" REAL NOT
NULL, FOREIGN KEY ("restaurant_id") REFERENCES "restaurant" ("id"));CREATE INDEX
"table_restaurant_id" ON "tables" ("restaurant_id");
```

Tabs Table




```
CREATE TABLE "tabs" ("id" INTEGER NOT NULL PRIMARY KEY, "had_reservation" INTEGER NOT NULL,
"party_size" INTEGER NOT NULL, "timestamp" DATETIME NOT NULL);
```

User Table




```
CREATE TABLE "user" ("id" INTEGER NOT NULL PRIMARY KEY, "email" VARCHAR(255) NOT NULL,
"password" VARCHAR(255) NOT NULL, "name" VARCHAR(255) NOT NULL, "role" VARCHAR(255) NOT NULL);
CREATE UNIQUE INDEX "user_email" ON "user" ("email");
```

e. Network Protocol

Our system sends web pages like many modern applications in order to present information and views to the user. The standard protocol for serving web pages over a network is the HyperText Transfer Protocol (HTTP). Our application allows for the use of HTTPS (Secure HTTP) if the business desires that all traffic be encrypted. HTTP is a protocol that is built upon

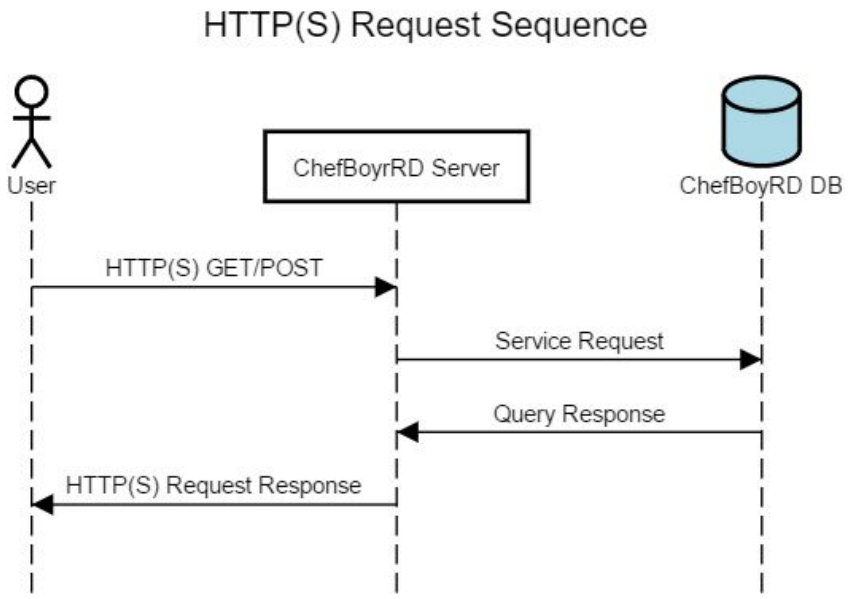
the TCP (transmission control protocol). We chose to create an application that uses HTTP because has been a standardized protocol with many resources which allows rapid development of software applications.

The reason that HTTP(S) was chosen is because it is the standard protocol for sending data and views to the user. Not only has HTTP been implemented across hundreds of platforms but it is a tried-and-true method of interacting with the user that many modern applications use today. The technology is stable and support for the protocol is widespread. Because HTTP is implemented within many cross-platform web browser across almost every operating system it allows our app to become agnostic to the platform that the business chooses to use. The alleviates restrictions on hardware design and lowers the cost to our business customers when implementing the ChefBoyRD system.

User devices will make HTTP GET requests to receive the information about the web pages. When users interact with the website, clicking buttons use HTTP POST requests to signal events and entered information on the user's end.

HTTP is a stateless protocol so and every request starts at one machine with a VERB. The valid verbs which are used in our application are GET and POST. An example of different requests sample requests which use HTTP GET and POST are provided below.

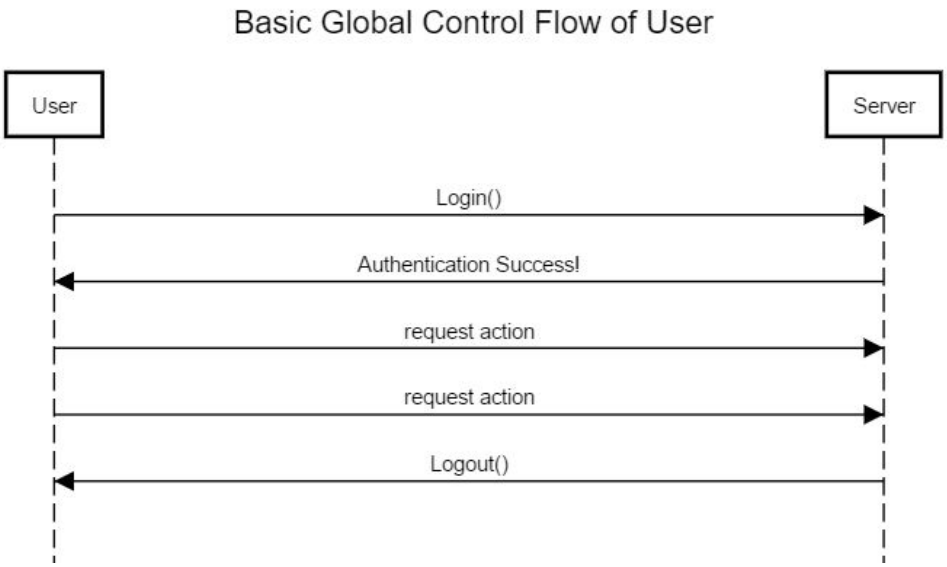
- User requests the ChefBoyRD main web page → HTTP GET
- User requests the ChefBoyRD login page → HTTP GET
- User presses the login form button → HTTP POST
- Manager wishes to view data business performance data by submitting a form with the type of statistics and range of dates → HTTP POST



f. Global Control Flow

Our system is based off of an internet site architecture. Because of this our application use is event-driven. Every time the user makes a request to the application a thread is created on the web server in order to handle the request, perform any database transactions, or calculations. When the user is not interacting with the application the server remains in an idle state.

Some sequences of actions may need to be performed in a linear fashion in order to complete properly. For example if a host needs to seat a user at a table they would first need to request login → send credentials → navigate to the seating chart → enter the customer information → request seating. This type of operation may need to be performed for the first time on login. Otherwise the user can just keep making requests in order to fulfill their desired actions. Login and logout should always be the first and last actions performed by the users respectively within the application domain.



The application does not have any components that are dependent on timed actions. We do, however, have an algorithm that uses time explicitly as a datapoint to execute its intended goal of predicting our orders. Our system is solely an event-response type.

Our web application was not designed to take advantage of multiple threads. However, the web server that hosts our page has concurrent multi-threading built-in their service. They use asynchronous threads, each of which handle a request from the threadpool.

Database write and read operations are atomic in order for several users to be served at a given time. This allows simultaneous access to the database.

g. Hardware Requirements

Server:

Hardware	Minimum Requirements	Recommended Requirements
Operating System	Windows Server 2003/MacOS 10.6.6/Ubuntu or CentOS	Windows Server 2012/MacOS 10.10+/Ubuntu or CentOS
Processor	Intel: Xeon E5502 AMD: Opteron 1352	Intel: Xeon 1230v5 AMD: Opteron 6320
Memory(RAM)	2GB ECC RAM	4GB ECC RAM
Hard Drive	500GB	4TB
Network Card	100/1000Mbps	100/1000Mbps

Client:

Hardware	Minimum Requirements	Recommended Requirements
Operating System	Windows 7/MacOS 10.6.6/iOS 9/Android 5.0.1	Windows 10/MacOS 10.10+/iOS 10/Android 6.1.1
Processor	Intel: i3 2.4Ghz AMD: Phenom X3	Intel: i5 3Ghz AMD: Ryzen 1500X
Display	1280x720 Capable Display	1980x1080 Capable Display
Memory(RAM)	2GB RAM	4GB RAM
Hard Drive	16GB	32GB
Network Card	100/1000Mbps	100/1000Mbps

10. Algorithms and Data Structures

a. Algorithms

Prediction and Dashboard

One of the main complex algorithms of our application is predicting the quantity of different ingredients being used for a certain time period. This algorithm is composed of multiple steps. First, the data must be retrieved from the database and converted into usable python structures. Our database query is formatted such that we get a time-sorted list of raw order data.

Then, we must transform and preprocess the data so that the main mathematical procedure of the algorithm can work efficiently. Our algorithm depends on the day, week, month, year. Therefore, we must extract those values from the time python object. Furthermore, for certain features of our algorithm, we may decide to normalize and scale down its numerical value to a certain range. Next, that data, in the form of a 2D array, will be fed into the algorithm and output its predicted output. We will repeat this algorithm for each ingredient, since the algorithms we are using have only a single output.

For the final and most important part of our algorithm, we will have the user input a specific time range that they want the predicted ingredients for. Since our algorithm only works for a specific time, with the lowest time unit being the hour, we must divide the time range into hour-long buckets and perform the algorithm on each bucket. Finally, we sum up the ingredients used for each bucket, and that is used as our output.

We will run this algorithm using various mathematical models to see if we can find the model that allows us to best predict the restaurant's ingredient usage. In report 1, we outlined and detailed the specific models we would be using. To reiterate, our two main models are the polynomial with a complexity of our choosing and a sinusoidal model.

One issue that we have is that we don't want to train the algorithm every time we want to get the predicted ingredients for a time range. To fix this, we also made the database store a table with our model's parameters so that they can used to easily predict the output for a certain time, since the model is a simple mathematical equation that can be run in constant time.

We decided with a nonlinear regression approach to train our data because we felt as though a restaurant's customer activity followed a very hill-shaped curve where in the morning, there's hardly anyone there, and near lunch and dinner, the curve peaks. This is what gave us the idea to use a sinusoidal model for our nonlinear regression. We also decided to use a polynomial approach since that could also model the hill-shaped curve we envisioned.

When gathering statistics our system simply utilized SQL queries with WHERE clauses in order to retrieve lists of data matching the specified parameters. After initially creating those WHERE clauses with peewee ORM it was simple to join on other tables and refine our search

parameters down to specific items, prices, etc. We could then use this raw data to generate the statistics for Alexa and graphs by simply iterating over the raw data and computing the necessary fields.

After computing statistics we needed to actually create graphs for the user interface. This was accomplished by using the matplotlib library [14]. The images were generated on the server-side and sent to the user within the HTML in Base64 encoded strings as the **src** attribute within the <img> elements. This helps to save HTTP overhead by requiring the client to make less HTTP requests to our server because otherwise the client would then need to make multiple HTTP requests for each of the graphs we wish to serve to the user. We also don't need to store or cache any of the images on our end creating less of a burden on the server and our system. The reason we chose this over using a client-sided chart library is because matplotlib integrated very well with python and kept most of the code in one place. However we could possibly improve this by moving the chart computations to the client side in javascript rather than python in order to save network bandwidth by only sending the data for data rather.

Feedback

Message Categorization

Within the feedback system, a decision-making algorithm is used to categorize submissions based on message content. The input is preprocessed and divided into a list of words which is then compared with various keyword lists associated with certain categories. For example, the list associated with positive feedback include good, excellent, outstanding, etc. The results of this analysis are indicated as flags and are stored in addition to the original message in the database. The flags are “good”,”bad”,”food”,”service”, and “exception”. The meaning of the positive “good” flag is that something positive was said in the message. This does not mean that the message as a whole has been determined to be positive, only that a portion of it conveys a positive message. This means that message may have both the positive and negative flags set simultaneously. Messages for which both the positive and negative flags are set will be shown when the mixed filter is selected.

A simple keyword search is used to determine if a given message relates to food or service. The presence of any word on the food keyword list in the SMS message will cause the food flag to be set. Similarly, keywords on the service word list will set the service flag and keywords on the exception list will set the exception flag. The purpose of the food and service flags are self-explanatory, and the use of the exception flag will be discussed shortly.

In order to determine whether a message contains a positive or a negative portion, we use positive and negative keyword lists as with the other categories, but with additional steps. People may express a positive feeling by using a negative word and negating it, such as when food is described as “not bad”. In order to address this possibility, when a word in the body of the

feedback message matches one on either the positive or negative keyword lists, the previous word is also examined:

If this preceding word matches a keyword on the negation list, the opposite flag will be set (e.g. In the “not bad” example, the word bad is found on the negative keyword list, but the presence of the negation keyword not before the word bad causes the positive flag to be set instead).

In addition to negations, there are words that extend the meaning of the following word. An example of this is “very good”, for which a positive feeling is expressed with two words. The word “very” is the extension word in this example. If the word preceding a positive or negative word is on the extension keyword list, it is ignored and the previous word to the extension keyword is examined. This allows us to view “not very good” as a negative expression, as well as “wasn’t too great”. Finally, if the preceding word is not found on either the negation list or the extension list, the original flag is set (i.e. a positive word will set the positive flag if no negations or extensions are found) .

One flaw in this method of analysis is that it does not accurately interpret a message in which the positive or negative word is implied. An example of this is “The food was good but the service was not”. The second part of the statement does not explicitly state “the service was not good”, but the meaning is implied due to the first part of the sentence. As a result, the negative sentiment is not detected by the algorithm. However, this is where this exception flag comes into play. Sentences that change their sentiment generally have an exception word as an indication, such as “but”,”however”,”whereas”,”except” etc. The presence of an exception keyword does not necessarily mean the algorithm has made a mistake, but it is a good indicator to the user that the algorithm is unsure and the result should be checked manually for accuracy.

Word Cloud

For the implementation of the word cloud, certain common and unuseful words were dominant. Words such as “the”, offer no insight into the meaning of the responses and waste space in the word cloud. To combat this, a stand list of stopwords was used to filter out these unnecessary words. Additionally, decided to add the word “restaurant” to the stopword list, as it was mentioned frequently but offered no real value to the manager.

Table Reservation

When reserving a table through our system we employ a best fit algorithm to determine what table to seat relative to the party size. The system first checks if the time slot for the given is within hours of the restaurant. After confirming, it then checks all reservations against tables

during the time frame. If there is no conflict with a table it adds that table to a list. After completing the list the system will choose the table which is closest to the size of the party.

Shift Management

When creating shifts, the system will check if the shift is before the current time and has the proper roles that will be existing in the database. From there, the shift will become available for the entire staff to view. The shifts are stored with the name, start time, end time, and the role of the shift, From then, when an employee claims the shift, the potentially claimed shift is compared to the current employee shift schedule and is check for scheduling conflicts. If there is a scheduling conflict, the employee is unable to claim the shift, other than that, as long as the role of the employee matches the role of the shift, they are able to claim the shift. In order to post a shift, the employee name and role must match the system stored shift to prevent the wrong employee from posting another person's shift. An admin and manager will always be able to post a lower tier employee but unable to post a manager or admin shift. Removing a shift will only allow open shifts from being removed. If the shift is open, and the remove shift is invoked, the controller will compare the role of the current user and verify if they have the permissions to remove the shift.

b. Data Structures

Our software is structured as a web-based application, In this way our code is only executed when a request is made to our web service. We also pull information from our SQLite based database for most requests. The complex data structures are typically handled by the web application libraries and the python ORM library which is used to construct SQL queries. A list of libraries used in our application can be found in the references section at the bottom of the report [14]

Statistics/Alexa

- Due to the fact that our data is stored within the database we access the data from we get it passed as iterable python object. This python object allows us to loop over each entry and access the data of the row object from the Peewee ORM model. All of the query data is stored in these iterable lists. The graphs and statistics are simply gathered through iteration over the result list from the database queries. There is no need for more complex data structures because the database takes care of the complicated querying, storing, and accessing of our data.

- Once we collect a list of results from a database query we need to use that data to create graphs or aggregate and compute on the data in order to collect useful statistics for the

user. However we still need to simply iterate over each result list in order to create the statistics (and create our graphs). Once images are generated, the data for the graphs are actually stored as a Base64 encoded strings which allows us to simply included the string as the **"src"** attribute for all image elements on a page. This simplifies loading pages because we don't need to create image files which would need to be requested asynchronously by the user's browser and stored elsewhere for the server. This helps lighten the burden on the HTTP by reducing HTTP connection overheads by forcing less connections to be opened.

- Because Alexa doesn't need to display any graphs, the iterable list of rows returned by peewee ORM is the only data structure needed because once the data is returned we simply iterate and process over each row one by one before returning the statistics to the user in the form of a spoken audio message. No fancy search structures are needed other than the SQL queries constructed with Peewee.

Prediction/Training

- First, we obtain our data from the database as a raw list of orders sorted by date. Then, we convert that list into a python dictionary of lists. The dictionary contains a key for each meal type that we have. The list for each meal is a date-sorted list of orders.
  Example: {“meal1”: [hour, day, month, year, mealCount], …}

- Next, we pass this dictionary to the training functions and they loop through each meal type and obtain the model parameters for each meal type. The function then returns a dictionary of lists where the dictionary contains keys for each meal type and each key corresponds to the model parameters for that meal type.
  Example: {“meal1”: [1, 2, 3, …], “meal2”: [1, 2, 3, …], …}

- Finally, to predict data, we are given two python datetimes from our interface, and using the model parameters dictionary, we plug in the hour, day, month and year into the model that we specify along with our model parameters for each meal type. Finally, we return a dictionary that maps meal types to meal amounts predicted.
  Example: {“meal1”: 3, “meal2”: 2, …}

Feedback

- A dictionary is used for gathering a list of words and associated frequencies for use in the word cloud. Example: {[text:spaghetti, size:12],[text:happy, size:2]} Each word is associated with a frequency, so the word cloud searches through each item in the dictionary, reads the contents of the text key and uses the associated frequency.

- A list is used for storing a series of information that is fetched from the database. The list can be easily iterated through to get each individual element in the list. Also lists are useful, we do not know in advance the number of elements that will be retrieved from the database.

Table Reservation

- A table is represented by an object given by peewee ORM model. Each table has the fields id, occupied, x,y, size, and shape.

- The table object is sent to the client in the form of a json dict. Example {[id: 1, occupied: 0, x: 0.5, y: 0.7, size: 3, shape: 0]}

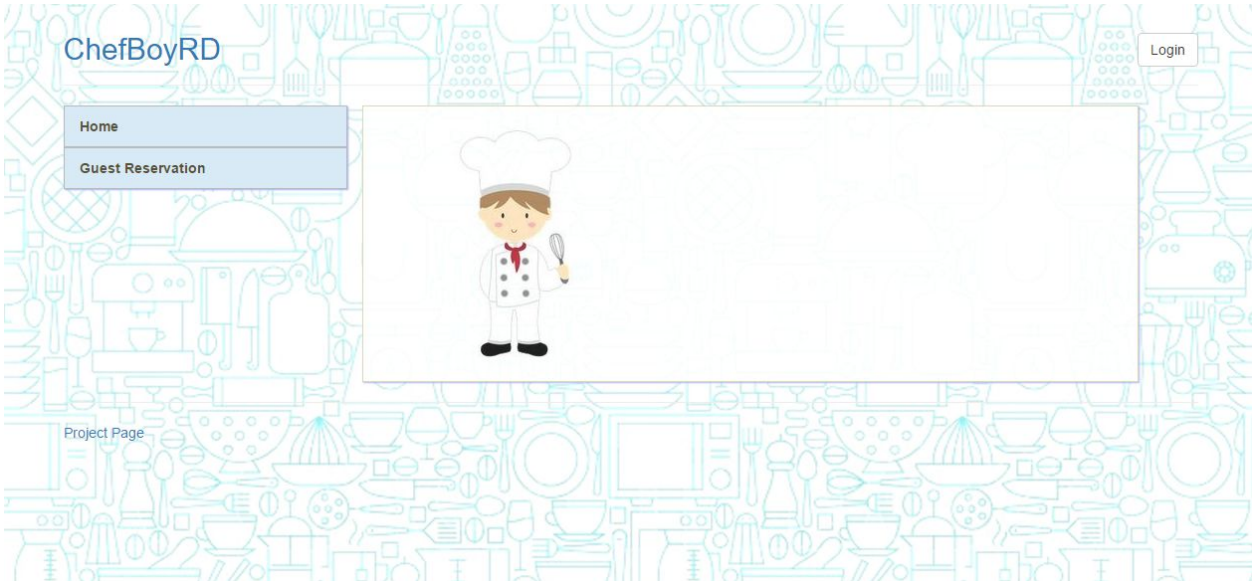
11. User Interface Design and Implementation

a. User Interface Design

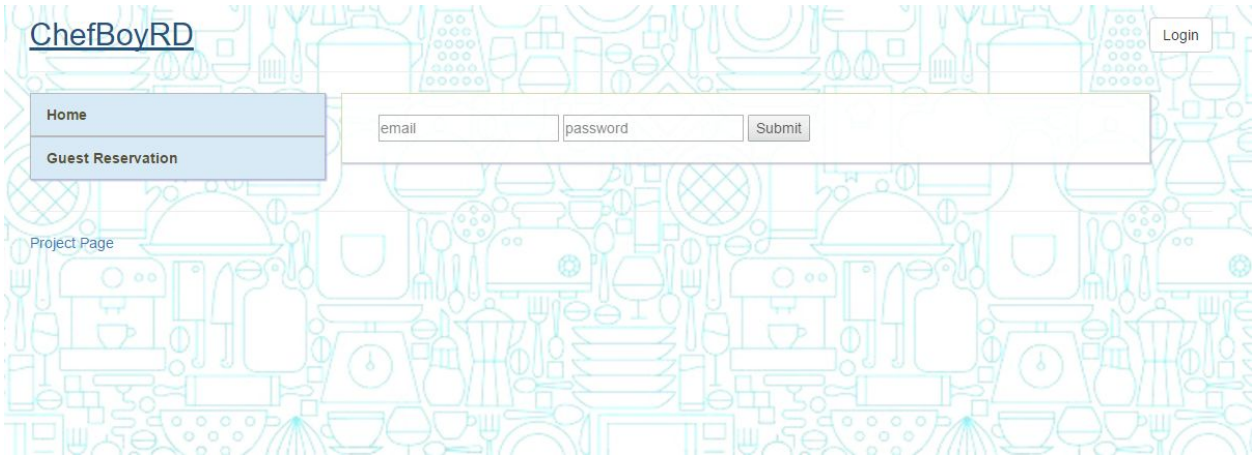
The user interface design will be served via web pages for all users. For the customers trying to make a reservation, the overall interface remains the same. The customer inputs their name, phone number, date and time of the reservation, and party size. The interface is extremely simple to use and displays information in an easy to read fashion. The interface for the employees is also very simple: username and password logins for access and tabs separating the different databases and information.

The GUI design is simplistic and focuses only on necessary information. There are no pictures or unnecessary data to distract the user from his task. On the customer side, the necessary forms and data inputs are large and centered in the screen for easy input and submission. On the employee side, the charts and data are displayed easily and only require a few clicks to access any desired data. Every button and function is labeled and easy to read. Some forms are auto-filled to reduce the number of clicks for common operations (default date range for feedback query).

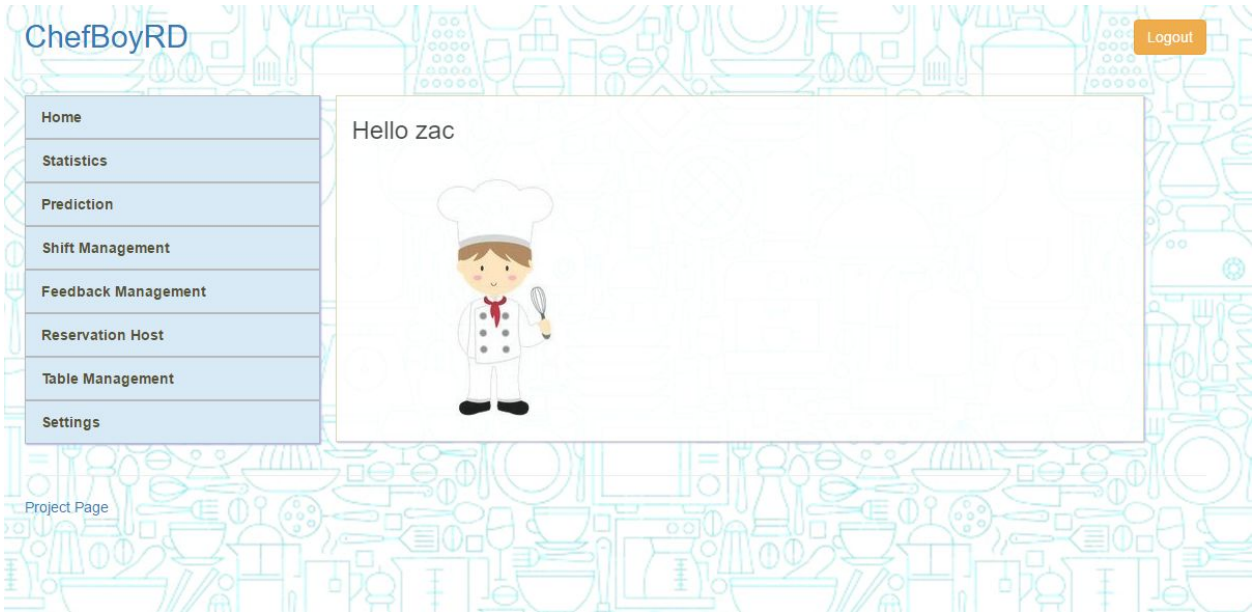
Overall, the user design interface is mostly unchanged. There may be stylistic changes later which means the mockups may not look exactly like the finished interface. However, the overall framework and design of of the employee portal and customer feedback interface will remain unchanged, as the functionally significant design is present in our previous mock-ups in report 1. The following are a couple examples of our implemented User Interface designs:



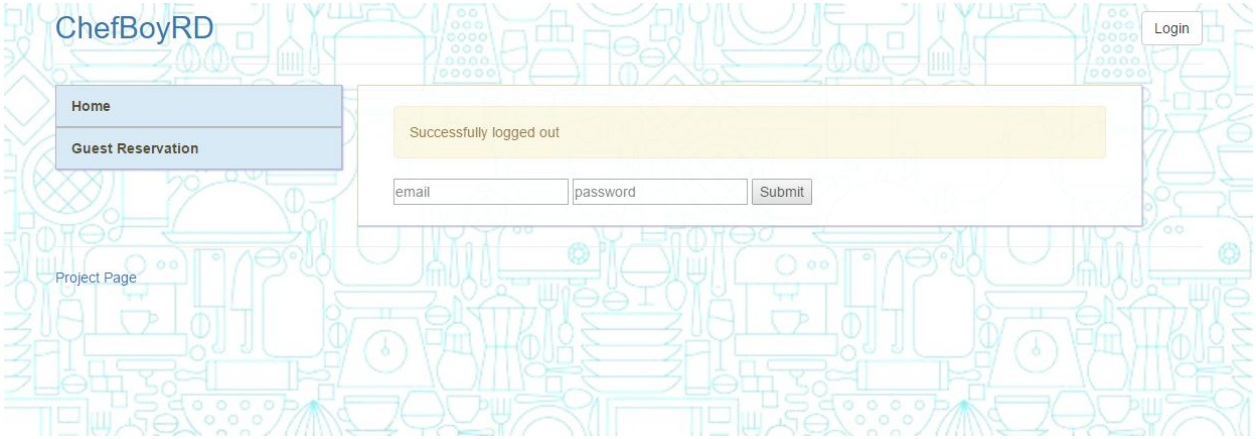
Home Screen



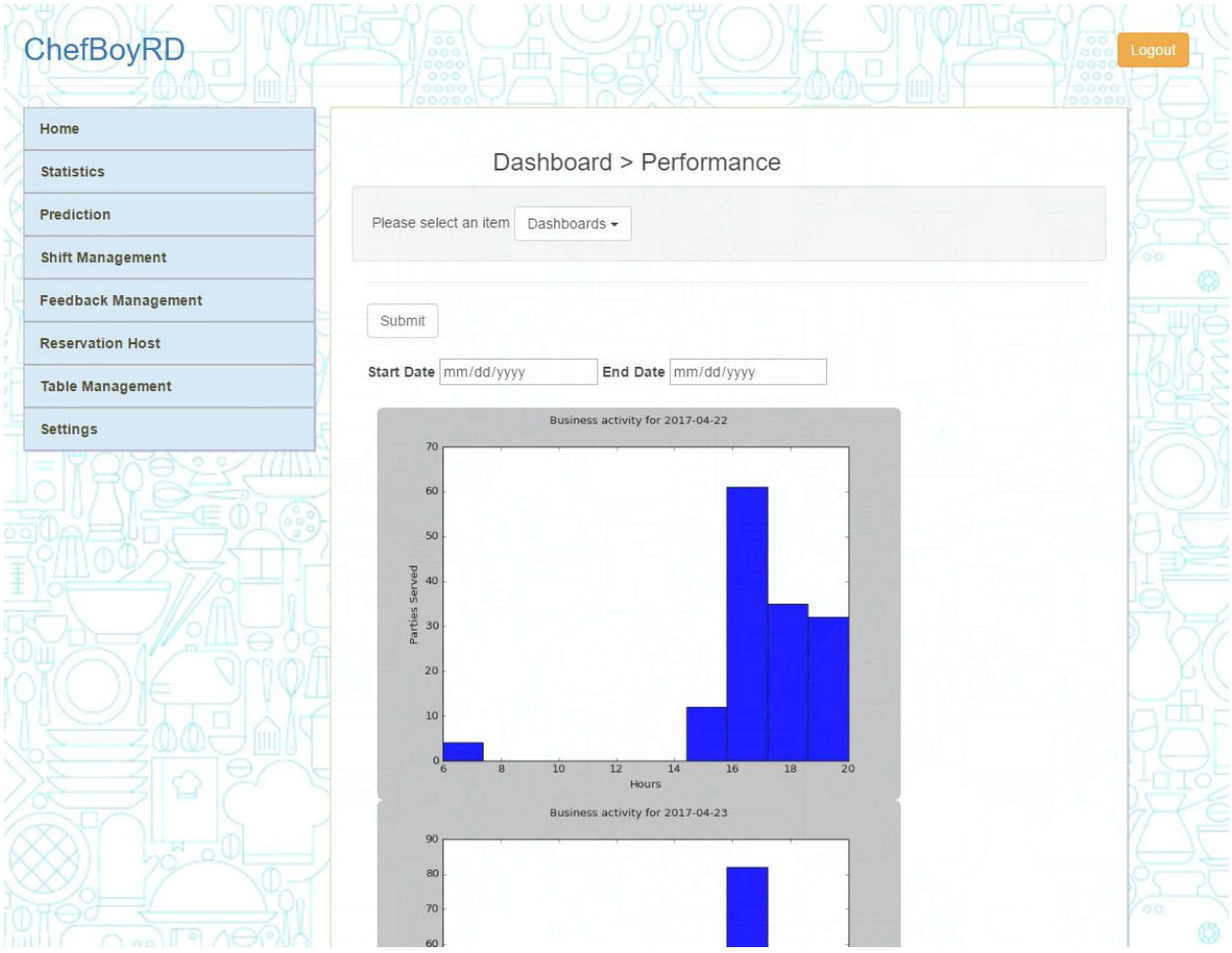
Login Screen



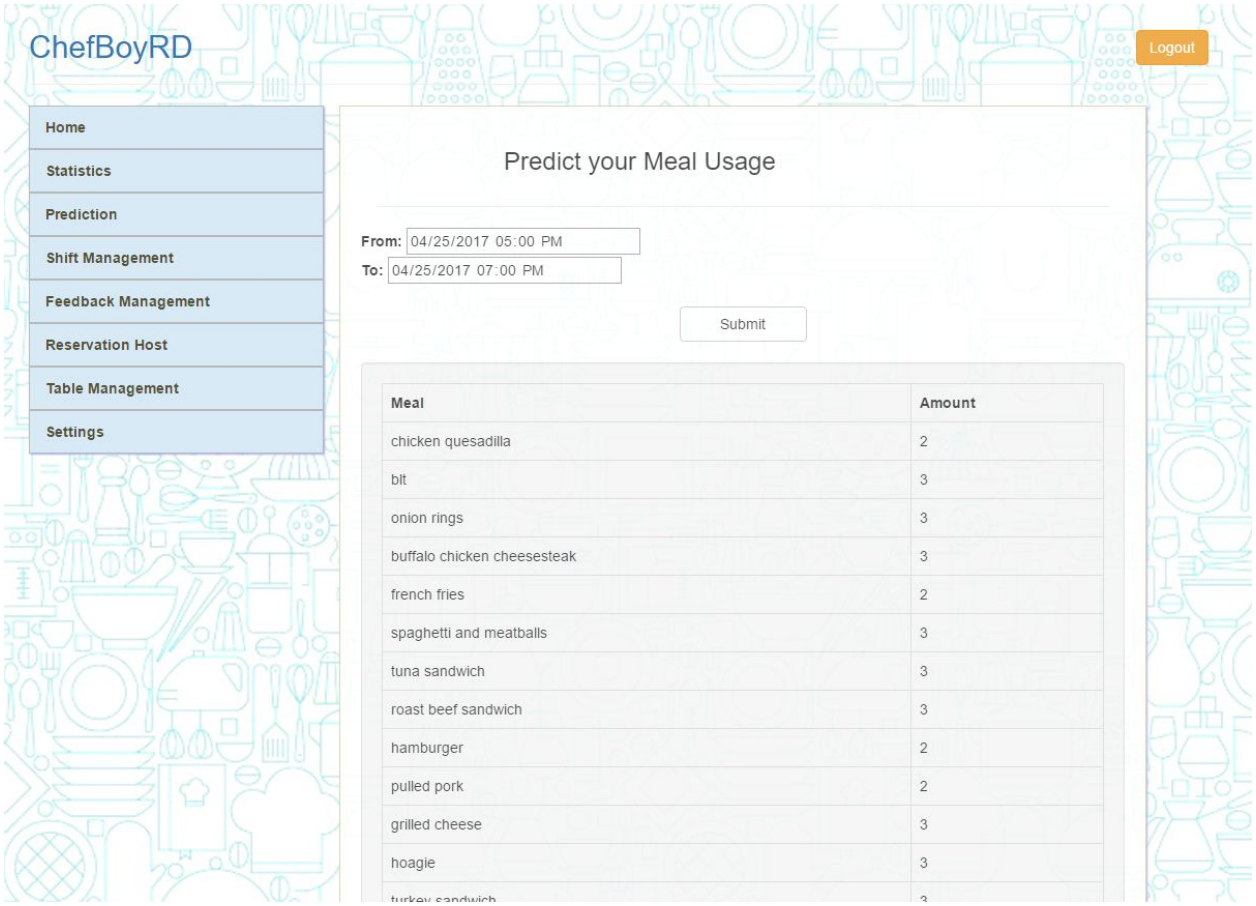
Welcome Page (Manager)



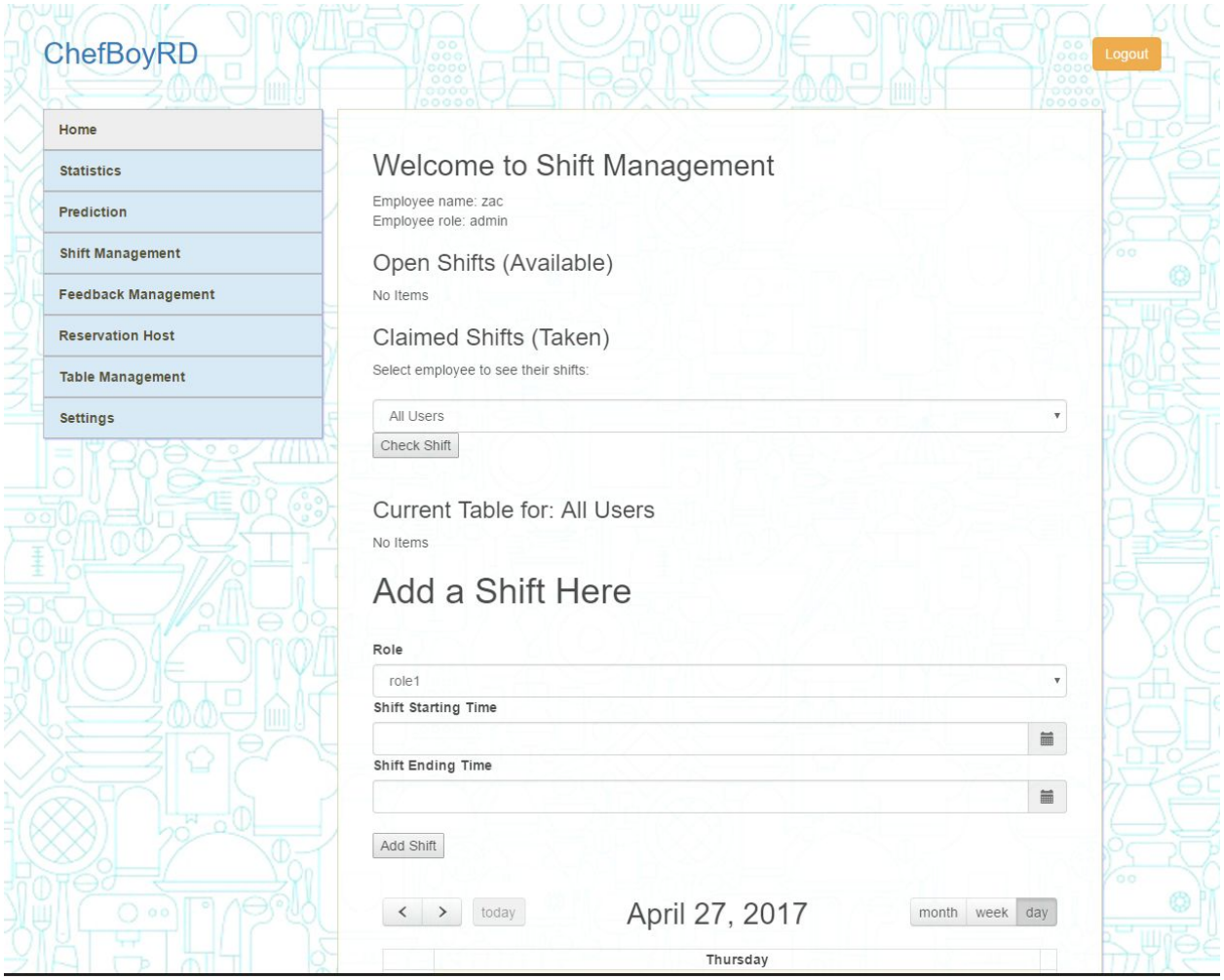
Logout Page



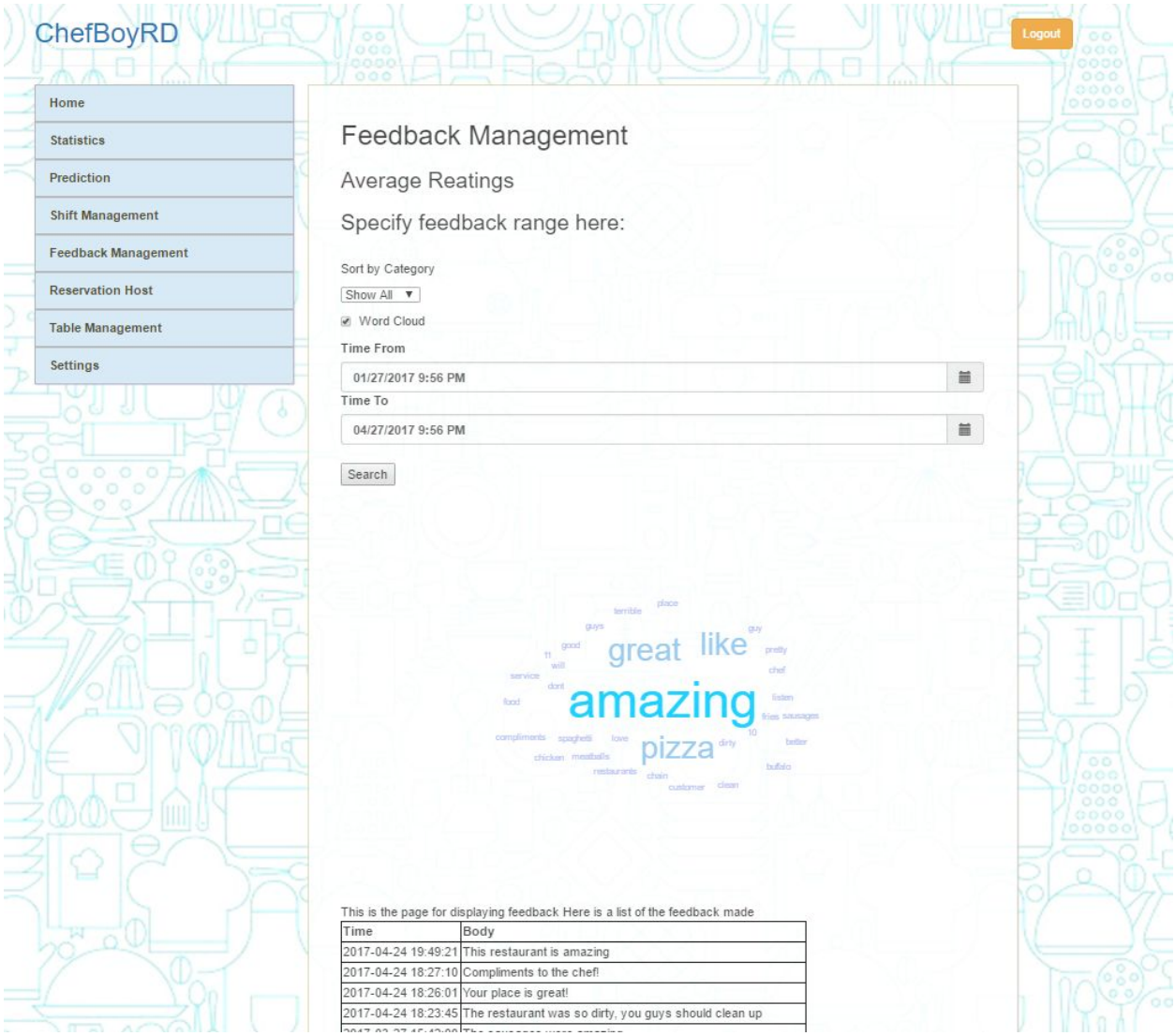
Statistics Dashboard



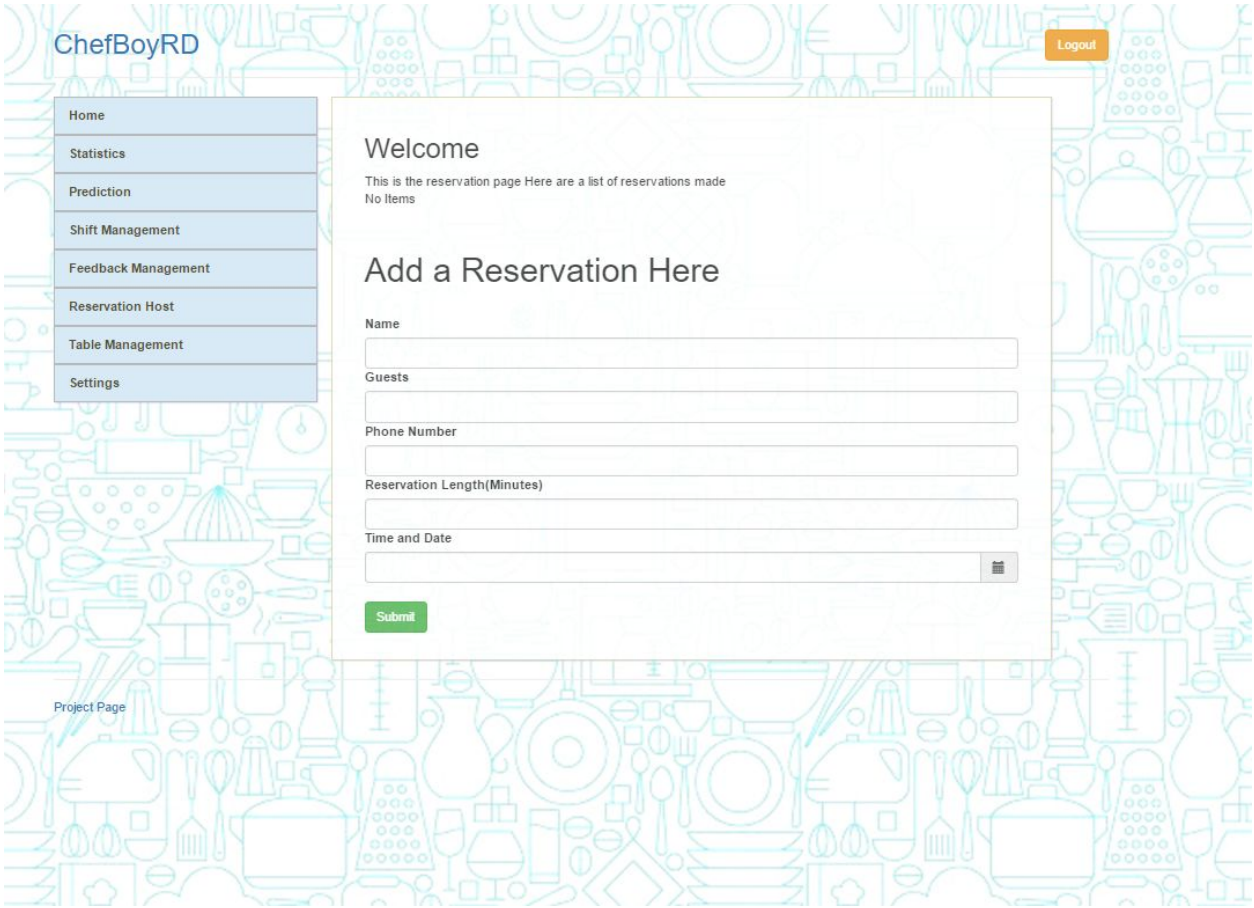
Prediction View



Shift Management



Feedback Management



Reservations (Host Interface)

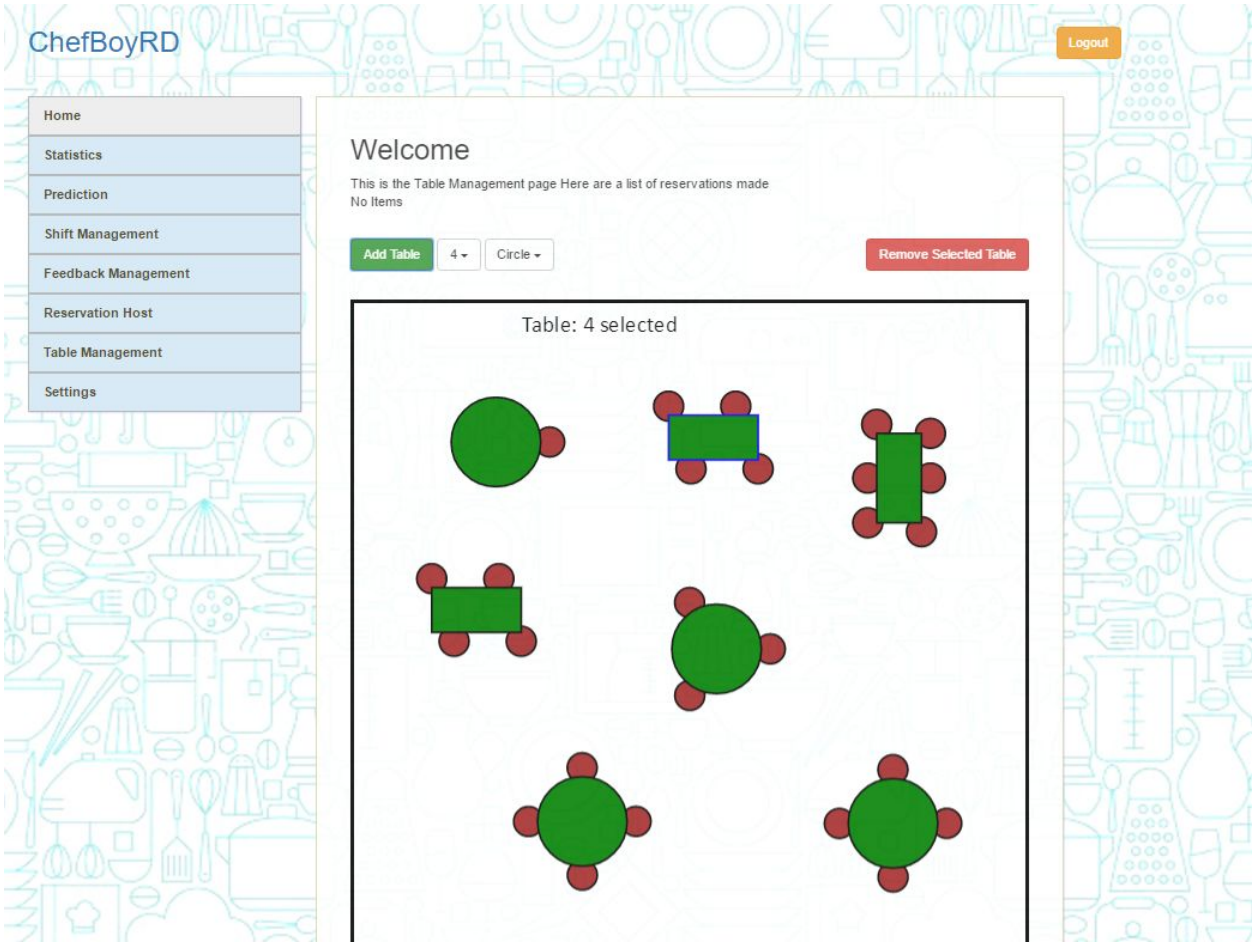


Table Management Interface

b. Implementation

The interface is implemented using an HTML templating system called Jinja2 [7] along with the Bootstrap [9] CSS framework. Jinja works well with the underlying web framework, Flask [8], that our application is built upon. Using Jinja allows us to create small UI components from pure HTML and CSS. Then we can make other templates which include the UI components and render information for the user.

By having this modular system it allows us to create an easily extensible application where new features and UI components can be added almost effortlessly and previous work can be extended upon with little to no overhead. Our current implementation uses a "root" default template which all other templates inherit from in order to keep the entire style of the site

consistent through each user interface. From this default template all of templates are extended and built upon one another.

The twitter bootstrap significantly simplifies the styling process for our interface as well and allows us to focus on the pure design of the interface rather than the technicalities of UI design such as device scaling and resizing.

12. Design of Tests

a. Test Cases

The following are our test cases for the main functions of our use cases. A full documentation of each of our unit tests can be found here:
http://blanco.io/ChefBoyRD/autodoc/chefboyrd.tests.html

Test-case Identifier: TC-1 Use Case Tested: FD-UC1, FD-UC2 Pass/fail criteria: The test passes if the day's order data is successfully passed to the database and stored Input data: Current day's order data	
Test Procedure:	Expected Result:
Step 1. Current day's data is sent to the database	System database updates itself with the new data

Test-case Identifier: TC-2 Use Case Tested: FD-UC1 Pass/fail criteria: The test passes if the prediction algorithm produces a prediction of the next day's food usage using order data from the database Input data: Order data from the database	
Test Procedure:	Expected Result:
Step 1. System database performs prediction algorithm on the order data	Prediction algorithm makes a prediction about what food will be used the next day

Test-case Identifier: TC-3 Use Case Tested: FB-UC1 Pass/fail criteria: The test passes if the feedback database receives and stores feedback data, and reports any significant data trends Input data: New customer feedback and existing feedback from the database

Test Procedure:	Expected Result:
Step 1. Send feedback to the feedback database	Database stores the data and updates database
Step 2. Send a chain of extremely positive feedback regarding an aspect of the service (a particular employee, dish etc)	Database alerts user to the service/employee/dish that received the positive feedback
Step 3. Send a chain of extremely negative feedback regarding an aspect of the service (a particular employee, dish etc)	Database alerts user to the service/employee/dish that received the negative feedback

Test-case Identifier: TC-4 Use Case Tested: FB-UC2 Pass/fail criteria: The test passes if the feedback database receives and stores feedback data, and sorts data based on content Input data: New customer feedback and existing feedback from the database	
Test Procedure:	Expected Result:
Step 1. Send feedback to the feedback database	Database analyzes the feedback and sorts based on content

Test-case Identifier: TC-5 Use Case Tested: TR-UC1 Pass/fail criteria: The test passes if the system correctly determines if a reservation is available for a customer, and makes a reservation if it is available Input data: Customer reservation containing date, time, and party size	
Test Procedure:	Expected Result:
Step 1. Customer requests a reservation, giving the time and party size (assume there is a spot open)	System checks the table schedule and updates to include the new reservation, sends confirmation

Step 2. Customer requests a reservation, giving a time and party size (assume there are no tables free to accommodate the size)	System checks the table schedule and offers an alternate time

Test-case Identifier: TC-6 Use Case Tested: TR-UC2 Pass/fail criteria: The test passes if the system correctly identifies if there is a table for the arriving customer party Input data: Customer party size	
Test Procedure:	Expected Result:
Step 1. Customer party arrives, party size is sent to the system (assume there is a table open)	System checks the table schedule and updates to reflect the new party seating
Step 2. Customer party arrives, party size is sent to the system (assume there are no tables free to accommodate the size)	System checks the table schedule and places customer party into a queue to wait for a table to open

Test-case Identifier: TC-7 Use Case Tested: MS-UC Pass/fail criteria: The test passes if the employee portal successfully provides the employee shift schedule and updates as necessary, and allows the manager to manually change/override schedules Input data: Employee shift data	
Test Procedure:	Expected Result:
Step 1. Employee accesses the employee portal	Portal accurately displays current employee shift schedule
Step 2. Employee makes changes to his/her schedule	Portal updates to reflect these changes
Step 3. Manager accesses the employee portal and creates/edits employee schedules	Portal updates to reflect these changes, overriding any prior employee changes

Test-case Identifier: TC-8 Pass/fail criteria: The test passes when a user can successfully login and logout of the application and only access resources which belong to its specified role Input data: Employee shift data	
Test Procedure:	Expected Result:
Step 1. Create an employee user in the Users table with a role of admin	Employee record is present in the table with same name and role
Step 2. Make a POST request to the login page with a bad password	The page returns an HTTP 401 code and the user remains logged out and cannot access any of the authorized
Step 3. Make a POST request to the login page with the correct username and password	Page response is an HTTP 200 code and use is redirected to the home page
Step 4. Request all pages requiring an admin or manager access	Page responses result in HTTP 200 and pages are displayed correctly
Step 5. Request the logout page	User becomes logged out and gets an HTTP 200 response
Step 6. Request other pages requiring the user to be logged in	HTTP 401 codes should be returned for all pages which require authorization and /or login
Step 7. Login as a non-admin user and request pages which require admin and non-admin access	Pages requiring admin role return HTTP 401 while pages which require login (but not admin role) still return HTTP 200.

b. Unit Testing

Our plan for unit testing includes having at least one functional test for each method in every single class of our project. Unit tests will provide at minimum the following inputs for each function in order to minimize realized errors when integrating systems together.

- Check for at least 2 cases of expected input and expected output where the function works properly

- Write at least one test case where the method should accept and invalid input which should raise some kind of error.
- Check edge cases for mathematical functions where boundaries like -1, 0, +1 (and similar) where methods might be expected to go wrong.

By providing these types of tests for every method we can ensure that the method blocks are functional and provide us with satisfactory knowledge that our system shall function correctly when building and integrating our system.

c. Test Coverage

Several of our tests involve state-based testing. For example, the reservation and seating test cases specifically deal with the state of the table schedule (whether there are vacant tables, if the restaurant can accommodate the customer party, etc).

The algorithms we use are tested with sample data to ensure their accuracy. For example, in test case TC-3, we send multiple samples of feedback to the database to test if the algorithm can catch the positive and negative trends we will send. In addition, we will send sample data to the prediction algorithm and feedback sorting algorithm to make sure they process the data and produce their results accordingly.

Overall, the tests cover all aspects of the use cases. Each scenario of each algorithm is tested as well. In addition, since the tests deal with receiving and displaying information from the databases, the tests cover connectivity between the databases and application. Additional tests will be designed and implemented as the need arises.

On top of this we can also utilize code-coverage tools like Coveralls (https://coveralls.io) when building our test system which can help us spot logical areas which we may have missed when designing our unit tests.

d. Integration Testing Strategy

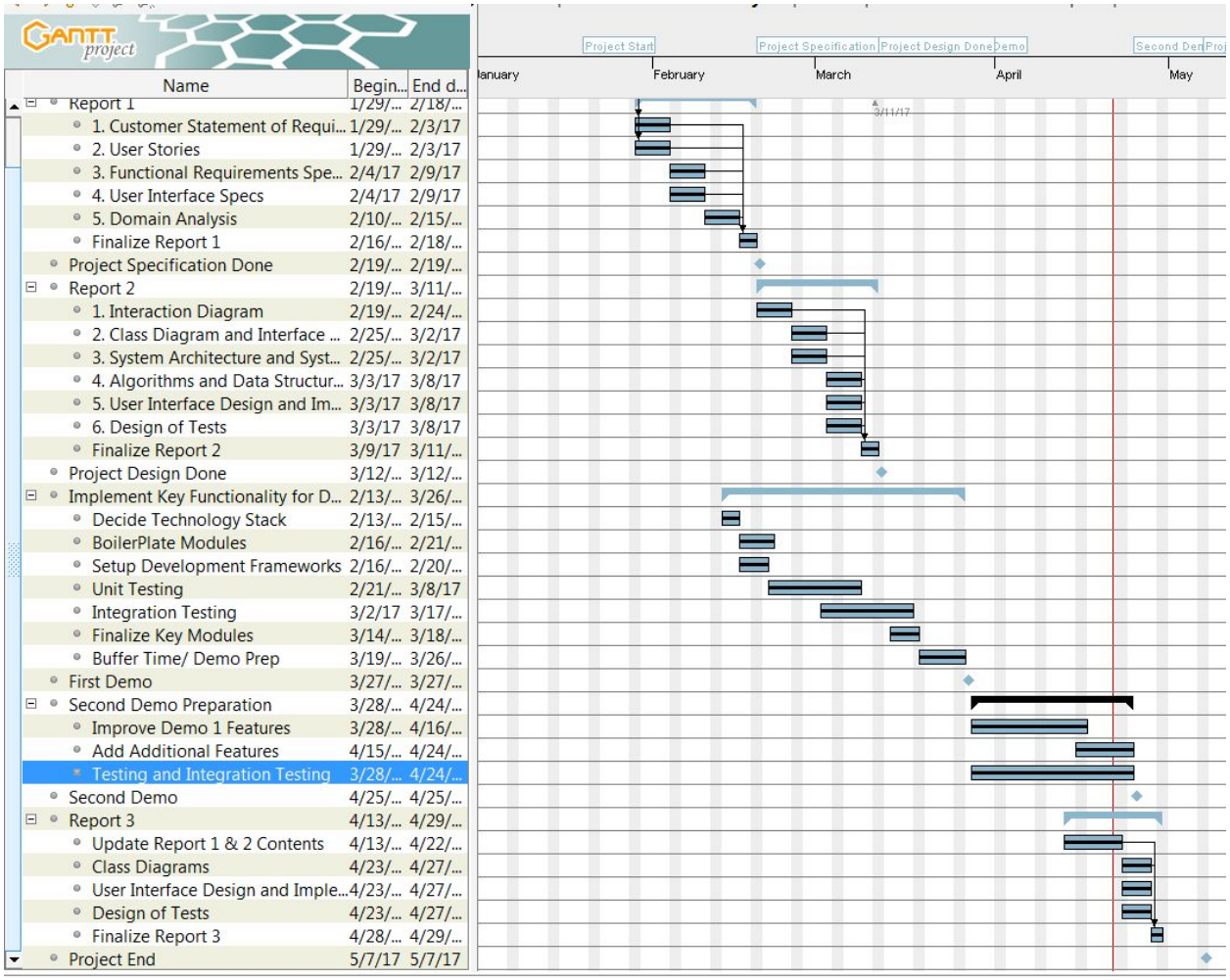
We decided to use bottom-up testing to test our project. Since there are well-defined parts of each component of our project, it would be simple to test each part individually, then test them as they were integrated with each other. For example, the customer feedback portion of our application is composed of several parts: the customer sending feedback to the system, the database and the algorithm. We test to make sure the feedback is received and stored, then we test the algorithm using the data received.

During project development the developers will write tests to go along with every feature as it is pushed into the main codebase. In order for merges to be able to occur we have set up a continuous integration system using Travis-CI (https://travis-ci.org) and GitHub

(https://github.com) which will run the tests with new code changes and requires passing all tests before new changes can be merged in. This ensures that each feature will perform and function without breaking other features that had already been developed.

13. History of Work, Current Status, and Future Work

a. History of Work



This is overall how our milestones evolved. Our deadlines did not change much from the previous parts. We followed the project deadlines with a day or two as a buffer to make any last changes/edits.

There were little to no deviations from this plan of work, with the plans of work in our previous reports.

January 17 - January 29:

The spring semester started January 17. We formed our project groups on January 20th, and met intermittently before and after class. We talked about different project ideas and it came down to three ideas: a home network security monitor, restaurant automation, or stock market fantasy league. From a majority, we decided to pursue a restaurant automation idea. Our team brainstormed unique ideas that we could implement that would improve existing restaurant automation technology.

We organized our ideas and submitted our project proposal.

January 30 - February 5:

After submitting our project proposal, we immediately started working on part 1 of report 1. We specified the problem that our target customer, small-medium sized restaurant owners faced. We completed the customer statement of requirements, glossary of terms, and User Stories. In the perspective of the customer, this specifies the problem that our software will aim to solve. And from the feedback we received, we included more detail about how our systems will function.

January 17 - January 29:

The spring semester started January 17. We formed our project groups on January 20th, and met intermittently before and after class. We talked about different project ideas and it came down to three ideas: a home network security monitor, restaurant automation, or stock market fantasy league. From a majority, we decided to pursue a restaurant automation idea. Our team brainstormed unique ideas that we could implement that would improve existing restaurant automation technology.
We organized our ideas and submitted our project proposal.

January 30 - February 5:

After submitting our project proposal, we immediately started working on part 1 of report 1. We specified the problem that our target customer, small-medium sized restaurant owners faced. We completed the customer statement of requirements, glossary of terms, and User Stories. In the perspective of the customer, this specifies the problem that our software will aim to solve. And from the feedback we received, we included more detail about how our systems will function.

February 6 - February 12:

After submitting part one of the project. Most of our team came together and met to discuss the plan for the coming week. We tried to approach this week with a different strategy. We assigned individuals to lead specific projects, where they would delegate work to our other teammates, but in the end had to finalize their portion of the report. We applied this strategy to the Functional Requirements Specification, and User Interface Specification, and finished part 2 of the report behind schedule. We also made edits to our report 1 part 1, where we included more detail about problems our system is addressing on top of our primary problems. We did not receive feedback this week.

We also evaluated some of the previous projects dealing with Restaurant Automation. We could not find a project that we wished to implement. Some of the other projects were implemented with a computer/mobile application which meant it would be incompatible with our desired system. One project we found, GravyXpress, was implemented as a web app, but when we tried to test it for functionality, it seemed like the project was not fully functional/incomplete. We decided to borrow some of it's specifications and re-implement them our own way.

February 13 - February 19:

After submitting part 2 of the report, we started to prioritize the implementation,as there was only one section left for report 1. We added a use case to section 3, to include an employee portal system that we agreed would be necessary. We made some other changes to report 1 and finalized some details. Due to time constraints we had to submit our report, but some of the group members felt that some parts of report 1 could've been elaborated on or improved.

Our group met during this week and talked about how we would approach the implementation. We decided that Python would be the preferable language to work with, as a majority of us were familiar with it and if not, it would be simpler to learn. We decided to implement with Flask in the back-end, SQLite as the database, and AngularJS as the front end.

February 20 - February 26:

During this week we officially started the implementation process. A basic MVC framework using Flask was set up so that we had something to build on top of. The VPS was setup with a basic flask server with an index page. A basic SMS example with Twilio was set up. This week we started diving into our projects.
We also completed part 1 of report 2, the interaction diagrams.

February 27 - March 5:

Most of our group members used this time to familiarize ourselves with the framework. A majority of us were new to this framework. Basic user authentication was added at this point. This will allow different restaurant staff to have different features available to them. We completed our class diagrams and System Architecture section.

March 6 - March 12:

Worked on design of tests, describing algorithms and data structures. Made some changes to our UI Design, and we referenced back to report 1 about our design. At this point the UI has not been fully implemented yet so we only made some changes to the UI that were dependant on features we were implementing.

We also narrowed down who will be working on what feature of the project, and included that in the report.

March 13 - March 27:

Focused on implementation during this time.

Completed the Dashboard, Prediction, Table Management, Feedback, Reservation features for the demo. We created tests for these features as well as integration testing. Integration was very simple because the features did not rely on each other, and the framework is set-up so features can be added on easily.

March 28 - April 23:

For our second demo we plan on improving the features of our first demo and adding a few more features.

There were some errors in the grading of our Report 2, so after clarifying feedback from our . We were a bit lost on how we could improve the quality of our report, as we received a low holistic grading.

After receiving feedback on our demo, we worked on implementing our final features of the project. We added a shift management feature, improved the graphics of the website, improved the responsiveness of the front-end design.

We also fixed some bugs from our first demo, and improved code readability.

April 24 - May 2:

Final changes to Report 3, adding to the document based on the feedback we received. Fixing diagrams and analysis to include what was implemented for the final demo. Ensuring that data generation functions are working for the project.

b. Current Status

Currently implemented features:

Statistics

- Manager can pull statistics by hour, day of the week for ingredients, tabs, meals, performance, and revenue.

Prediction

- Based on ingredient usage history, software can predict how many ingredients need to be used in a future date range.
- Software can communicate this prediction data by using an Amazon Alexa Skill

Feedback

- Customer can text in feedback and that feedback is categorized for ease of analysis
- Counts the frequency of significant words sent in feedback.
    - Responsive word cloud to display.
- Manager can sort through responses based on certain predetermined categories over a customizable time range.

Reservation

- Customer can make a reservation online
- Host/Hostess can make reservations through their interface.

Table Management

- Host/Hostess interface to manage who is at which table at a given time.

Shift Management

- General Employees can keep track of their shifts using this interface.
    - Add, Drop shifts.

For all features, the website looks very good on mobile.

c. Future Work

- Export interface for all information in database (statistics, reservations, feedback)
- Advanced natural language processing for feedback
    - Utilize relevant phrases based on proximity instead of keywords.
- Allow sorting of feedback through applying one or more filters to the responses instead of only a limited selection of combinations.
- Implement an in-store survey system that allows users to provide numerical feedback on different aspects of their experience, and a UI to view these responses.
- Push Notifications
    - For notable data trends in statistics and feedback
    - For employees about when a shift is covered or not.
- Add a restaurant layout to the table GUI
- Clean up design so the features are completely modular - each feature can act standalone. (have an API so it can be added-on to another system)
- Easy-login management for restaurant staff
    - 4- number pin for quick login
- Predict waiting time for customers in queue.
- Customizable dashboards utilizing multiple types of graphs
- Display overall statistics rather than just graphs for dashboard interface
- Create charts on the client side - less network bandwidth and server strain
    - Improves upon customizability
- More specific Alexa commands
- Improve upon role authentication and authorization
    - Use a hierarchical role-based structure so that endpoints need only specify a single subgroups and all supergroups may still have access
    - i.e. If I am an admin (root group) I should be able to access any page regardless of authorization
    - As a manager I should be able to access tables, shift management, and etc.. even

14. References

- [1] - "Statistics on Food Waste in the United States", https://www.nrdc.org/issues/food-waste
- [Fig 0] ChefBoyRD Icon “Chef Boy” https://img.clipartfest.com/91a3a64a56686f2093016774a3f0ad63_boy-chef-pizza-chef-boy-clipart_236-236.jpeg
- [Fig 1a-1] Administrative Panel - “ UI for admin interface” “https://codecanyon.net/item/restaurant-order-mobile-app-android-ios/7668912
- [Fig 1a-2] Word Cloud - “Word Cloud“ http://theartofdoing.com/wp-content/uploads/2013/03/Chang2-copy.jpg
- [Fig 1a-3] Table View - “Table Management interface for POS software” http://www.vitabyte.com/wp-content/uploads/2012/11/Aldelo-Table-Management.png
- [Fig 1a-4] Reservation Form - “Online Reservation form” https://tableagent.com/static/img/screenreservation.png
- [Fig 1a-5] Employee Schedule - “Employee Schedule and Payroll“ https://s-media-cache-ak0.pinimg.com/736x/aa/11/bc/aa11bcdaafbaf8cc6927f6807ff848cd.jpg
- [2] - Excel or FluidUI (https://www.fluidui.com/) used to create charts/graphs, screen mockups.
- [3] - Interaction Diagrams drawn using SequenceDiagram (http://sequencediagram.org)
- [4] - Class Diagram symbols used following: Russ Miles and Kim Hamilton: *Learning UML 2.0* Reilly Media, Inc. 2006.
- [5] - UML Class Diagrams done using UMLlet Software (http://www.umlet.com/)
- [6] - Plan of Work done using GanttProject (http://www.ganttproject.biz/)
- [7] - Jinja2 Templating System for Python (http://jinja.pocoo.org/docs/2.9/)
- [8] - Flask Web Framework for Python (http://flask.pocoo.org/docs/0.12/)
- [9] - Bootstrap CSS Framework (http://getbootstrap.com/)
- [10] - Unit test and integration testing: Travis-CI (https://travis-ci.org)
- [11] -Version control and project management: Github (https://github.com)
- [12] - Coveralls (https://coveralls.io)
- [13] - Reference Restaurant Data Model (http://www.databaseanswers.org/data_models/restaurant_bookings/index.htm)
- [14] - List of Python Libraries Used:
    - File with list of libraries: https://github.com/ZacBlanco/ChefBoyRD/blob/master/requirements.txt
    - Requests, https://pypi.python.org/pypi/requests
    - Flask, https://pypi.python.org/pypi/flask

    - Peewee ORM, https://pypi.python.org/pypi/peewee
    - Twilio, https://pypi.python.org/pypi/twilio
    - Flask-Login, https://pypi.python.org/pypi/flask-login
    - werkzeug, https://pypi.python.org/pypi/werkzeug
    - gunicorn, https://pypi.python.org/pypi/gunicorn
    - matplotlib, https://pypi.python.org/pypi/matplotlib
    - wtforms, https://pypi.python.org/pypi/wtforms
    - Flask-WTF, https://pypi.python.org/pypi/flask-WTF
    - WTForms-Alchemy, https://pypi.python.org/pypi/wtforms-alchemy
    - Flask-Table, https://pypi.python.org/pypi/flask-table
    - Python-Dateutil, https://pypi.python.org/pypi/python-dateutil
    - phonenumbers, https://pypi.python.org/pypi/phonenumbers
    - scipy, https://pypi.python.org/pypi/scipy
    - numpy, https://pypi.python.org/pypi/numpy
    - scikit-learn, https://pypi.python.org/pypi/scikit-learn
    - hashids, https://pypi.python.org/pypi/hashids
- [15] - Other packages, libraries used for our front-end are directly mentioned in this directory:
    - https://github.com/ZacBlanco/ChefBoyRD/blob/master/chefboyrd/views/templates/head.html

Project Management

a. Merging the Contributions from Individual Team Members

We faced many issues when coordinating the contributions from all team members. At first we all split up section among the group members to contribute to. However, this resulted in low individual accountability, we would rely on each other to take ownership of the report section.

Instead, now we assign a section of the report to one or two individuals, who will be completely responsible for making sure all the requirements of the section are met. This way, if there is some specific information needed for the report, that only a couple team members know, the person responsible for the section will reach out to others.

We used Google Docs, so that we may work simultaneously and stuck to a specific formatting so that we would not create more work for ourselves by continuously formatting report sections. We used Google Drawings for some of the diagrams, but not using a dedicated software also limits us in the tools that we can use. We found that using UMLet is useful because it's a standalone software that is simple to use and does not require installation. It can also export in many formats. In the case we have specific formatting we need to adhere to, we have one person start the formatting and send the file to the other contributors.

After the initial hurdles, we figured out what works for our team and our coordination improved.

For project management of implementation, we used github to track the progress of different features. Whenever one of us would reach a milestone, we would push to the master branch and merge our contributions. Due to how we had subteams and we worked on different features, there were rarely merge conflicts.

There was a case where we all did not communicate and individually fixed the same error three times. This was an indication that communication between each team member could have been improved.

d. Breakdown of Responsibilities

Subteam A - Richard and Zac
Subteam B - Ben, Seo Bo, and Jarod
Subteam C - Brandon, Jeffrey

Module/Class*	Team Member Responsible
PredictionView	Zac
PredictionController	Richard
OrdersController	Subteam A
AdministrativePortalView	Subteam A + B
FeedbackForm	Subteam B
FeedbackReview	Subteam B
FeedbackResponseAnalyzer	Jarod
SmsInterface	Seobo
HostView	Subteam C
ReservationController	Subteam C
TableGUIController	Subteam C
EmployeePortalView	Subteam C
DatabaseHandler	Subteam A + B + C
AuthHandler	Zac
AlexaInterface	Subteam A

*some of these classes require implementing models, most of which are exclusive to a class, so the do not need to be mentioned here. See Class diagrams for in-depth information.

Tests will exist for all classes within the project. The author of each class is responsible for writing the tests as well. Unit and integration tests will be written to ensure that the project can function in pieces and as a whole together. When merging code into the master branch of the codebase we will at that point, ensure that tests have been written and pass before merging.

Builds and testing will be performed by Travis-CI. It is a tool which integrates with Github to run unit tests and integration tests as new modules are added to project branches. It helps determine the commits, and thus code changes, which result in failing unit and integration tests. This will enable us to pinpoint issues swiftly and with high accuracy. It will also help prevent buggy code from entering the codebase.

Integration into the main branch will be done by each group member as they finalize their modules/classes. It will be coordinated by Zac.