Find our GitHub repository at:
https://github.com/wongkristen/SE_RestaurantAutomation

Easy Eats

Report 2 – FULL

March 12, 2017

By Group #11: Group #11:Tejas Bhoir, Elizabeth Caronia, Mithulesh Kurale, Brett Lechner, Raj Patel, Prithvirajan Venkateswaran, Kristen Wong

Table of Contents

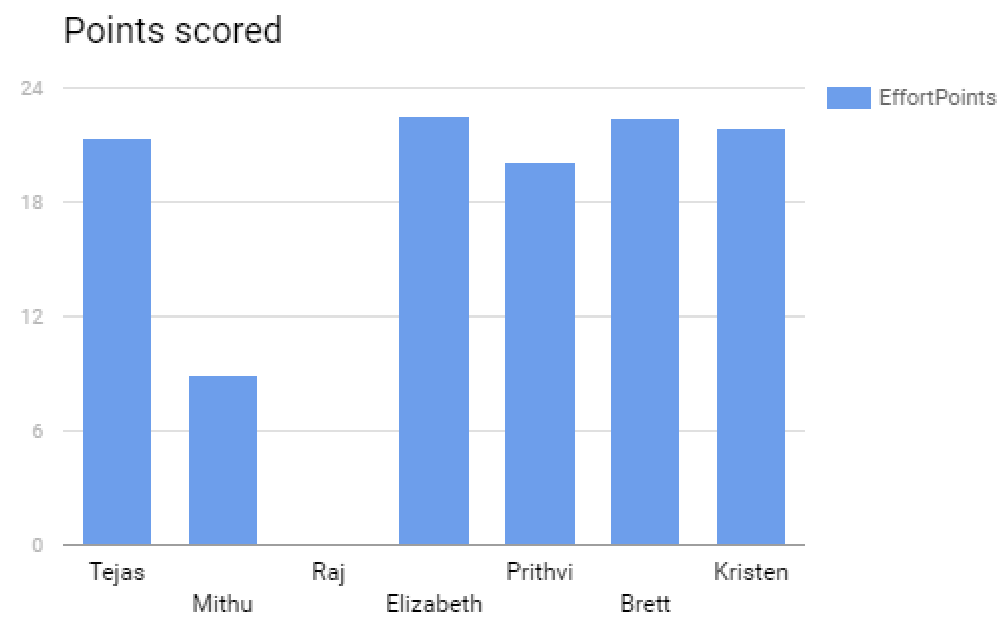
I. Individual Contributions

a. Responsibility Matrix

Responsibility Allocation

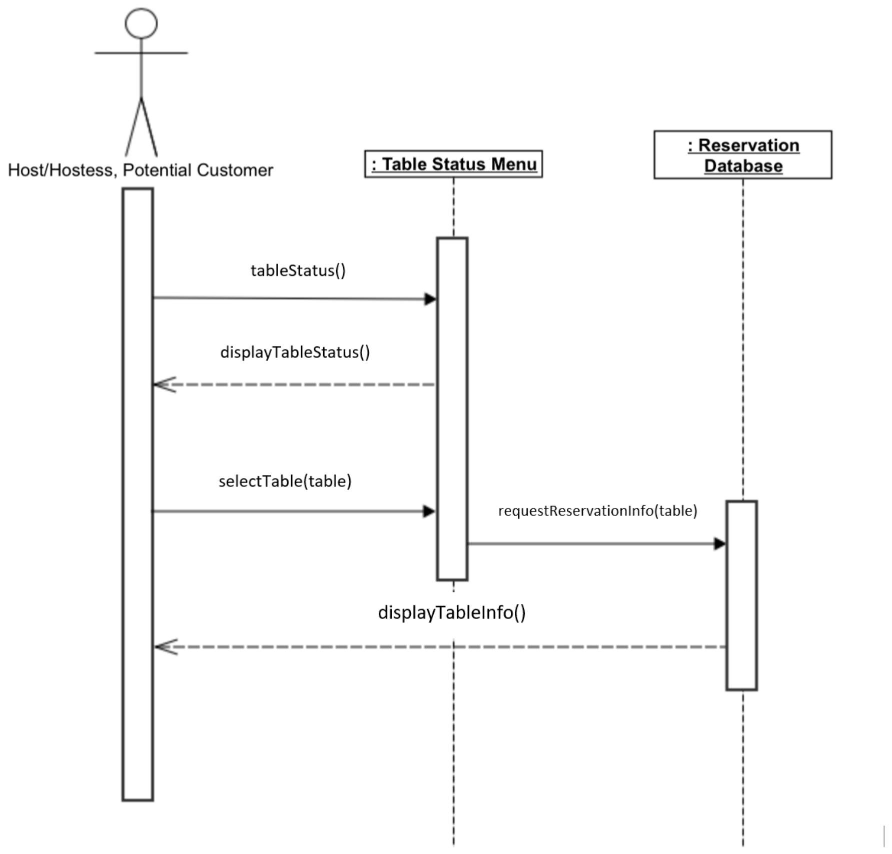
	Tejas	Mithu	Raj	Elizabeth	Prithvi	Brett	Kristen
TOC 2pts				100%			
Rep1: CSR 10pts	16%	16%		16%	16%	16%	16%
Rep1: SysReq 6 pts	5%			30%	5%	5%	30%
Rep1: FRS 30 pts	20%	10%			20%	40%	10%
Rep1: UI Spec 15pts	40%	10%		35%	15%		
Rep1: Domain 25 pts	20%	10%			20%	20%	20%
Rep1: Plan Work 5pts				100%			
Individual Contributions Breakdown 10pts				75%		25%	
Responsibility Chart 3pts				100%			
Sec1: Create User Cases 10pts				10%			
Sec1: Sequence Diagrams 10pts	15%				40%		45%
Sec1: Describe Design 5pts							100%
Project Management 6pts	16.6%	16.6%	0.1 %	16.6%	16.6%	16.6%	16.6%

b. Responsibility Allocation Chart



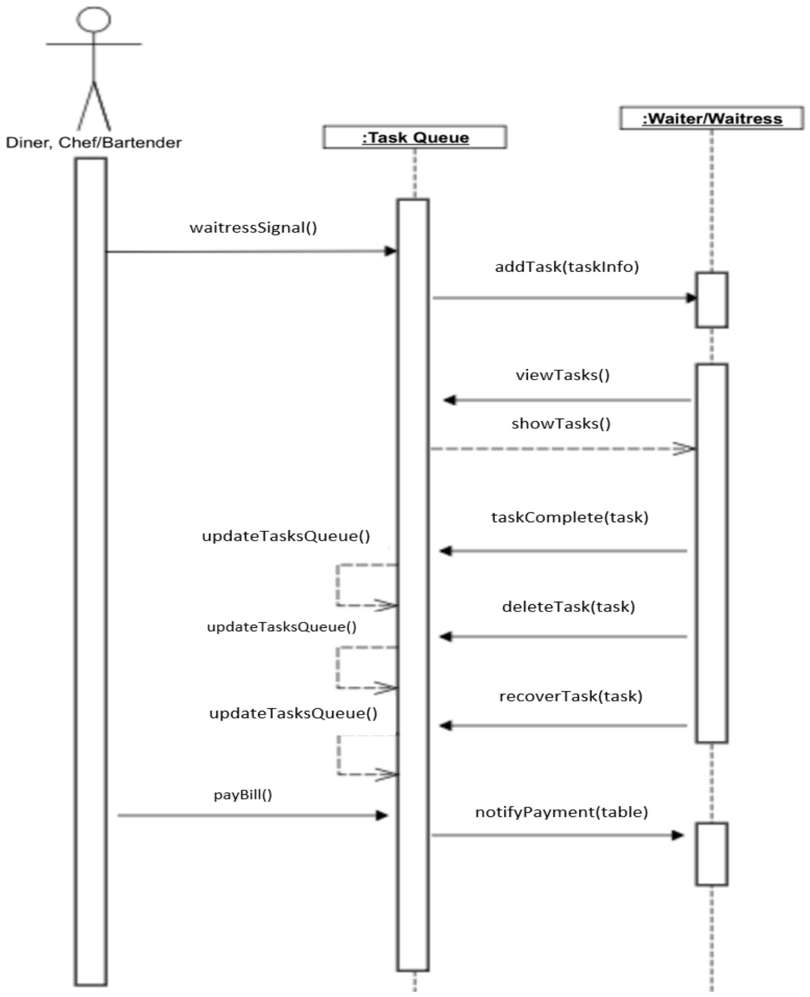
II. Interaction Diagrams

a. Use Case 1 - TableStatus



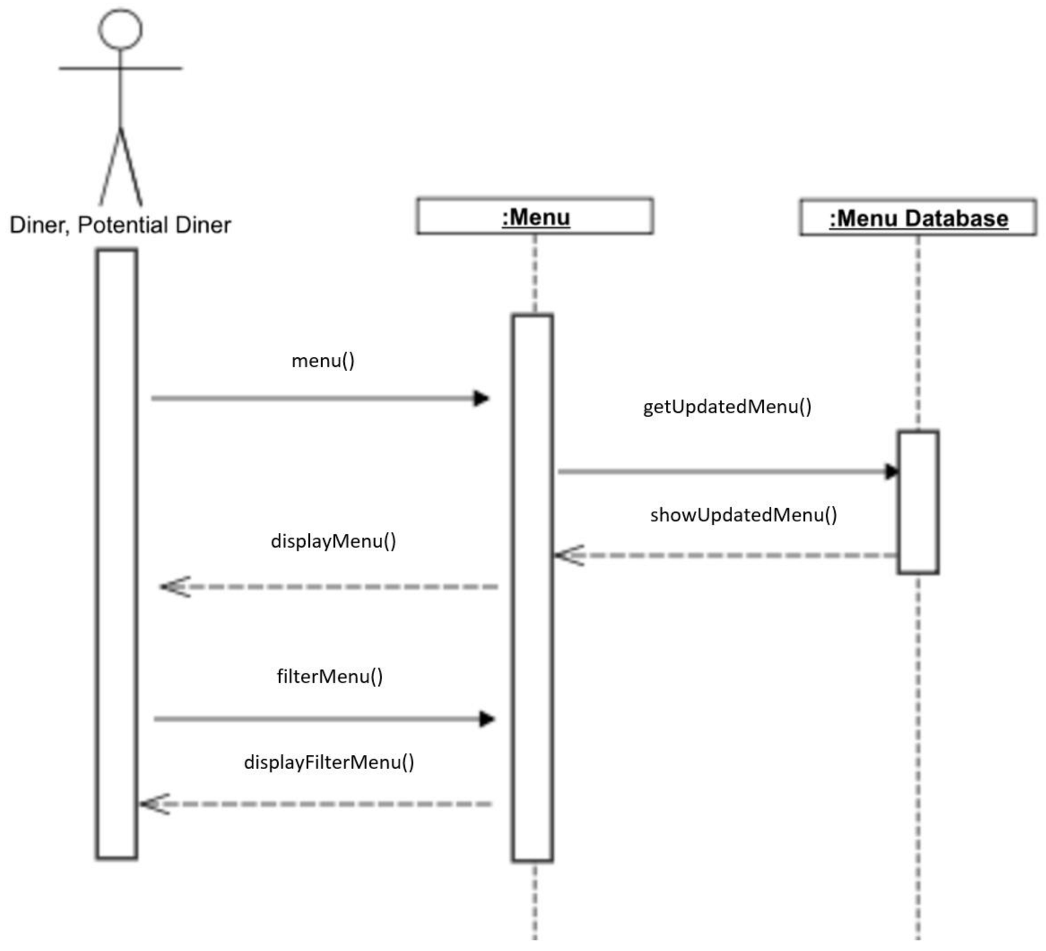
A hostess or potential customer can open up the table status interface through their smartphone or computer. Here they will see a color coded layout of the restaurant and with all of the tables and their statuses. The user can then select a specific table by tapping or clicking the screen. The app will then return a more detailed description of that table's status, including future reservations and estimated wait time..

b. Use Case 4 - Waitress Signal



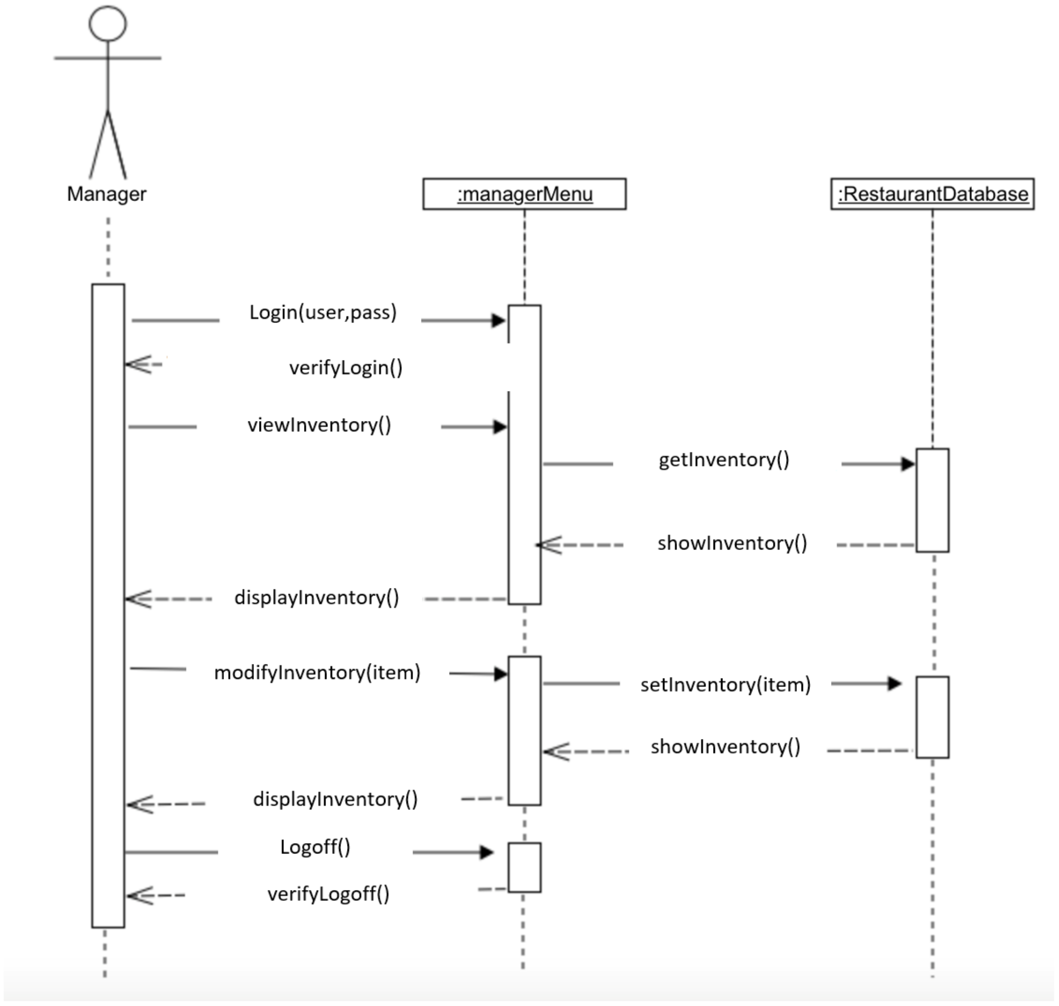
Diners, chefs, and bartenders are able to submit task requests to waiters and waitresses. When a task is submitted, it is added to the queue and the waiter or waitress is notified. The waiter/waitress is able to view their entire task queue on their interface. When a task is completed, the waiter/waitress can mark the task as completed and the queue will be updated. The waiter/waitress also has the options to deleted tasks and recover deleted tasks. Finally, once a table has paid for their meal, this notification will be added to the task queue and the waiter will be notified.

c. Use Case 8 - Menu



A diner or potential diner can view the restaurant's menu from a webpage. When a diner opens the menu application, it will display the most updated version of the restaurant's menu. From here, the user can select options to filter the menu items and the menu app will display appropriate items based on the user's selection.

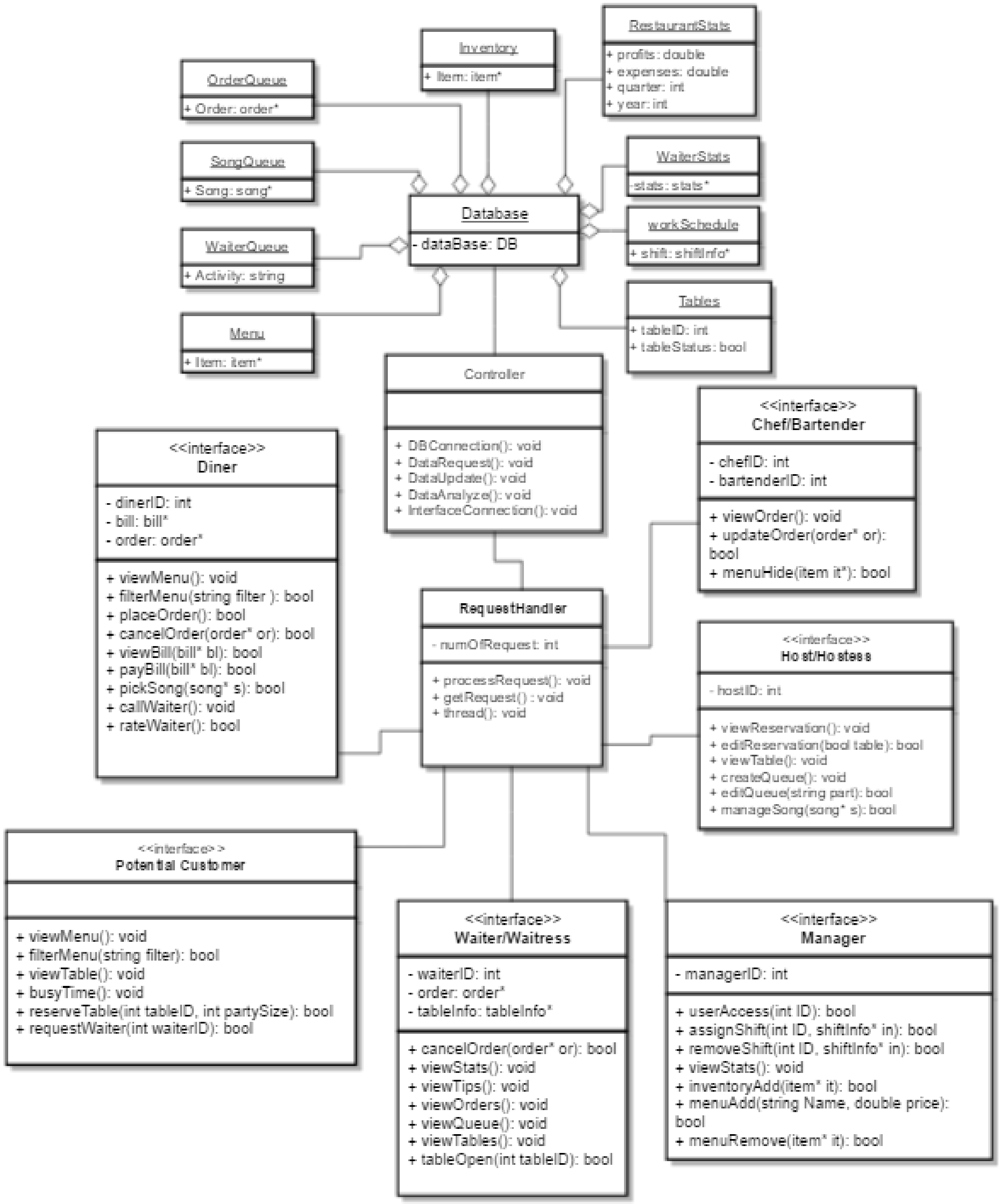
d. Use Case 15 - TrackRestaurant



The manager has the the authority to view and track the restaurant inventory with this application. When the manager first opens the trackRestaurant interface, he or she if required to log onto the system. Once his or her login information if verified, the manager can select to either view the inventory or change inventory data. If the manager selects to view the restaurant's inventory, the app will retrieve inventory data from the database and display it to the manager. If the manager chooses the change inventory data option, he or she can enter his or her changes. These changes will be sent to the database and the updated inventory will be displayed to the manager. When the manager is done using the trackRestaurant interface, he or she can log off of the system to ensure that the information is secure.

III. Class Diagram and Interface Specification

a. Class Diagram



i. Class Descriptions

Inventory - The Inventory holds the quantity of items in the restaurant. The manager is able to view and update the inventory as needed.

OrderQueue - holds the queue of the orders placed for the chef, bartender, and for the waiters to view. Chef/Bartender can update the queue for orders ready.

SongQueue - holds the queue of song requests made by the diners in the restaurant. The hostess has ‘management’ of this queue.

WaiterQueue - holds the queue of activities for the waiter to perform, whether a table is requesting them or the chef/bartender is calling orders.

Menu - The Menu contain all of the restuarant’s menu information, including the names of menu items and their descriptions.

Table - The Table is a class that shows the specific table in the restaurant. It shows the number of seats and their availability status. The waiter is able to view the availability status of the table.

Schedule - The Schedule is a class that displays which employees are working and the time table of the shift that they are working. The manager creates the schedule.

RestaurantStats - holds values for profits/expenses, and can show available records for the quarter and the year.

WaiterStats - holds the information diners submit about their experience with specific waiters.

Database - The Database stores all the information created and used within the application. This information includes the restaurant menu, restaurant inventory, schedules, and business stats.

Controller - The Controller communicates with the Request Handler and the Database. It shares information between the Request Handler and Database while sending requests.

RequestHandler - The RequestHandler receives all requests from the different interfaces. It then processes this information and sends specific instructions/information to the respective locations.

Diner - The Diner class allows customers seated at tables to view menu items and place orders. Additionally, Diners will be able to notify waiters/waitresses if they need assistance and pay their bill.

PotentialCustomer - Potential Customers have the ability to view the restaurant menu as well as table statuses and wait times before they have arrived at the restaurant. They are also able to place orders online or schedule reservations.

Waiter/Waitress - The Waiter class contains the functions for the waiter to receive notifications from customers and chefs/bartenders.

Manager - The Manager class contains all of the interface functions for the manager of the restaurant to carry out his/her necessary duties and oversee the restaurant. The Manager has the ability track the restaurant's inventory, manage the employees' schedules and performance, and view the restaurant's stats.

Host/Hostess - The Hosts/Hostesses are able to view and update the statuses of tables and the queue of reservations.

Chef/Bartender - The Chef/Bartender is responsible for maintaining the order and also handling the requests that is given by the waiter.

b. Data Types and Operation Signatures

i. Diner Interface:

This class is so that a customer can interact with the system to place or cancel orders.

Attributes:

- *dinerID: int*
  ID to represent the diner at a table.
- *Bill: bill**
  Struct containing information pertaining to a diner/table's bill.
- *Order: order**
  Struct containing information pertaining to a diner's order.

Methods:

+ *viewMenu(): void*
  Used to display viewable menu on the diner's interface.

+ *filterMenu(string filter): bool*
  Used to filter the viewable menu based on a specific filter requirement.

+ *placeOrder(): bool*
  Used to confirm a diner's order and send the order to the chef/bartender's queue.

+ *cancelOrder(order* or): bool*
  Used for the diner to cancel an order if it is in queue and not being made. Needs a confirmation from the waiter to be accepted.

+ *viewBill(bill* bl): void*
  Used for the diner to at any time view their current bill.

+ *payBill(bill* bl): bool*
  Used for the diner to pay their bill.
+ *pickSong(song* s): bool*
  Allows a diner to select a song to be put in queue on the SongQueue.
+ *callWaiter(): void*
  Allows a diner to simply call their waiter to their table.
+ *rateWaiter(): bool*
  After paying the bill, diner is asked to submit a short review of their waiter.

ii. Potential Customer Interface:

Interface available for anybody to access and view menu, see table availability, and to reserve a table.

Method:

+ viewMenu(): void
  Allows a user to view the menu on their device.
+ filterMenu(string filter): bool
  Allows a user to filter the viewable menu
+ viewTable(): void
  Allows a user to view the table availability chart to see how busy or not the restaurant is.
+ busyTime(): void
  Gives a user data regarding previously recorded busy days and times during the week
+ reserveTable(int tableID, int partySize): bool
  A user returns information used to reserve a table for a given size and time. Reservations are forfeited after a pre-disclosed time limit.
+ requestWaiter(int waiterID): bool
  If a user wishes to select a specific waiter for any reason, they can select them from this method.

iii. Waiter Interface:

The interface used by the waiter to service customers and communicate with the employees quickly.

Attributes:

- *waiterID:int*
  ID associated with the waiter.
- *Order: order**
  Struct defined with the information regarding a diner’s order.
- *tableInfo: tableinfo**
  Struct defined with the information regarding the waiter’s current tables.

Methods:

+ *cancelOrder(order* or): bool*

  Used to notify a waiter of a order cancel request, which they will resolve and update the order.
+ *viewStats(): void*

  Allows the waiter to see their reviews and stats on their interface.
+ *viewTips(): void*

  Allows the waiter to see their tips and payments on their interface.
+ *viewOrders(): void*

  Allows the waiter to see the order queue for their tables, to prepare anything ahead of time.
+ *viewQueue(): void*

  Allows the waiter to view a queue of certain things that may have been notified of. This is a backup system of backlog for the waiter
+ *viewTables(): void*

  Allows the waiter to view their tables and look where they need to go.
+ *tableOpen(int tableID): bool*

  Used to notify the host/hostess of a table where diners have left, and the table has been cleared and is ready for a new party.

iv. Manager Interface:

The interface used by the manager to manage various restaurant systems, namely work schedules, restaurant stats, and menu/inventory.

Attributes:

- *managerID: int*

  Reference number for the manager.

Methods:

+ *userAccess(int ID): bool*

  Manager can choose what ID's can access which sections of the databases.
+ *assignShift(int ID, shiftInfo* in): bool*

  Allows the manager to assign an employee a shift to work.
+ *removeShift(int ID, shiftInfo* in): bool*

  Allows the manager to remove an employee from shift.
+ *viewStats(): void*

  When accessed by a manager the viewStats will show them the waiter stats, as well as overall restaurant stats.
+ *inventoryAdd(item *it): bool*

  Allows the manager to increase amounts of an item used for orders when inventory comes in, and decrease for any reason.
+ *menuAdd(string Name, double price): bool*

Allows the manager to put a new order on the menu to be made available.

+ *menuRemove(item* it): bool*

Allows a manager to remove an order from the menu or to temporarily hide one, similar to the chef.

v. Host/Hostess Interface:

Attributes:

- *hostID: int*

The ID number representing the Host/Hostess.

Methods:

+ *viewReservation(): void*

Allows the host/hostess to view the reservations that have been made.

+ *editReservation(bool table): bool*

Allows the host/hostess an ability to edit the reservations for any reason.

+ *viewTable(): void*

Allows the host/hostess to view the table chart.

+ *createQueue(): void*

The host/hostess can create a wait for table queue.

+ *editQueue(string part): bool*

The host/hostess can edit the reservations that have been made.

+ *manageSong(song* s): bool*

The host/hostess can manage the song queue.

vi. Bartender / Chef Interface:

An interface for the chef and bartender interact with orders from the system.

Attributes:

- *chefID: int*

ID number for the chef.

- *bartenderID: int*

ID number for the bartender.

Methods:

+ *viewOrder(): void*

Used to display order queue on the chef/bartender's interface.

+ *updateOrder(order* or): bool*

Used to update status of the order represented by order* or.

+ *menuHide(item it*): bool*

Used to hide a menu item if the chef runs out of something needed to make the order.

vii. Request Handler:

Attributes:

- *numOfRequest: int*

  A queue set to store the number of requests at a given time frame.

Methods:

+ *processRequest(): void*

  Send the next available request to the controller when the controller is done with it's current request.
+ *getRequest(): void*

  Used to retrieve the request from a user's interface.
+ *thread(): void*

  Used by threads to process multiple requests efficiently and effectively.

viii. Controller:

Methods:

+ *DBConnection(): void*

  Used to establish a connection to the database.
+ *DataRequest(): void*

  Used by a request to pull information from the database to be used on a user's interface.
+ *DataUpdate(): void*

  Used to update data on the database and interfaces.
+ *DataAnalyze(): void*

  Used to pull records on stats stored from the system.
+ *InterfaceConnections(): void*

  Used to establish a connection with a user's interface

ix. Database:

The database consists of different varying objects including any information related to stored data, or interfacing data

Attributes:

- *Database:dataBase: DB*

  Identifier used to represent the actual database.
- *OrderQueue:Order: order**

  Struct defined to contain information regarding a diner's order in queue.
- *SongQueue:Song: song**

  Struct representing a song selected by the host or diner to be used in the queue.
- *WaiterQueue:activity: string*

  A queue representing any notifications the waiter previously received, now shown as a checklist.
- *Menu:Item: item**

The data pertaining to items on the viewable menu.

- *Tables:tableID: int*

  ID number representing the tables on the table diagram.
- *Tables:tableStatus: bool*

  A identifier to mark whether a table is taken or available.
- *workSchedule:shift:shiftInfo**

  Struct defined with the information regarding employees and their work schedules.
- *WaiterStats:stats:stats**

  Contains data pertaining to diner's review of their waiters.
- *RestaurantStats:profits: double*

  Holds the data regarding all income.
- *RestaurantStats:expenses: double*

  Holds the data regarding all outcome.
- *RestaurantStats:quarter: int*

  Contains data available for income and outcome through the quarter.
- *RestaurantStats:year: int*

  Contains data pertaining to income and outcome through the year.
- *Inventory:Item:item**

  Struct describing an item in the inventory.

c. Traceability Matrix

	Inventory	OrderQueue	SongQueue	WaiterQueue	Menu	RestaurantStats	WaiterStats	workSchedule	Tables	Controller	Database	RequestHandler	Diner	PotentialCustomer	Waiter/Waitress	Manager	Host/Hostess
1. Manager						x										x	
2. Order Item		x											x		x		
3. Table Status									x					x			x
4. Host/Hostess									x								x
5. Menu					x								x	x	x	x	x
6. ViewOrders		x													x		
7. PlaceOrder		x											x				
8. Chef																	
9. Check						x							x		x		
10. PayBill						x							x				
11. SongChooser			x										x				
12. Diner													x				
13. DinerExp							x						x		x		
14. OrderStatus		x											x		x		
15. CallWaiter				x									x		x		
16. RestStatus									x					x		x	x
17. cleanTable				x					x						x		x
18. InventoryCount	x																
19. Waiter							x								x		
20. WorkSchedule								x							x	x	x
21. RestInfo						x	x									x	
22. Bartender																	

IV. System Architecture and System Design

a. Architectural Styles

Our system uses a 2-tier client-server model. The client, such as a customer or the manager, will be connected to an integrated server. The server will use its own resources to provide services to the client and act as a database. This model allows the client to not be worried about how the server is implemented or how it performs its services. A waiter or customer can simply request any service they desire and the server will provide the service without any other input from the client. The model's ease of use is very important to a restaurant since a customer should be able to request a service without any knowledge of the server. The client can contact the server through different hardware, such as a computer, tablet, or smartphone.

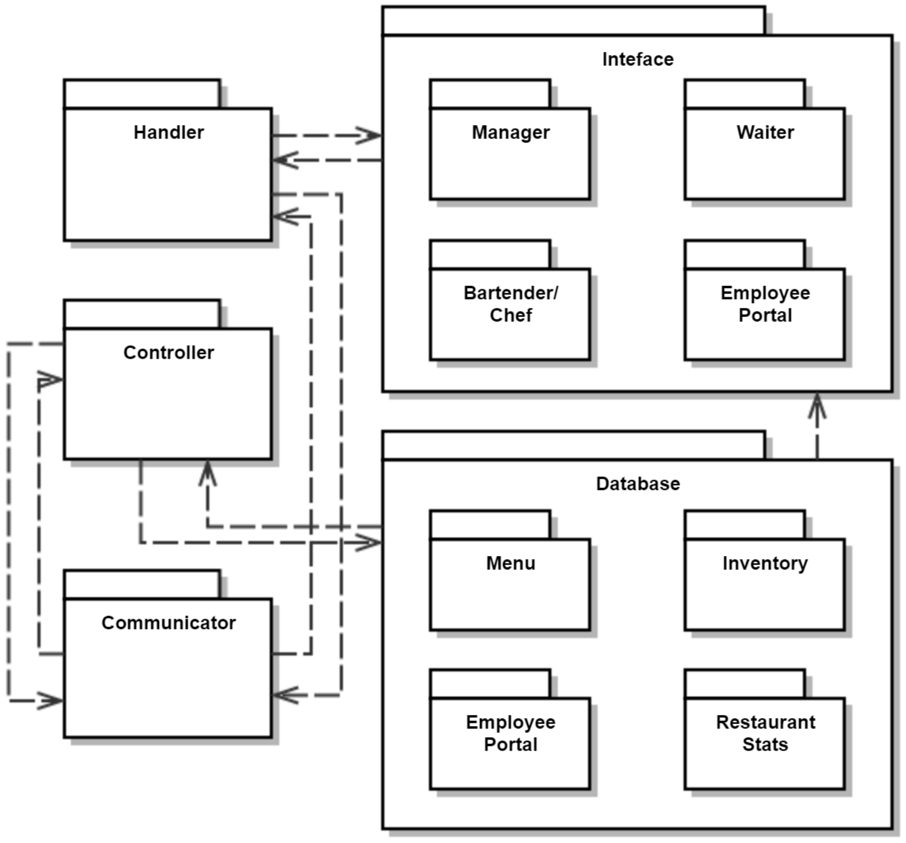
When a client first connects to the server, he or she will have to log in and be verified. Based on their credentials, the client will be given the appropriate permissions to the application. A manager will have administrative privileges to the application, allowing him to edit his employees' shifts or view restaurant statistics. On the other hand, a waiter would have limited access and would only be able to view information related to him, such as the list of orders or the amount of tips received.

The database in the server will be used to store information regarding the restaurant. This includes employee schedules, menu items, inventory, profit and other information. Clients will be able to modify the database based on their level of clearance. A customer will only be able to view certain items in the database such as the menu, while the manager will be able to modify everything. The database will also keep track of transactions between customers and the restaurant.

The clients and server will exchange messages in a request-response messaging pattern. A client will request a service to be done and the server will respond by fulfilling the request and then returning a message to the client. This model best fits the needs of a restaurant, where customers request services and the restaurant responds to them. The simplicity of this model allows maintenance of the system to be easier since the system only handles two-way communication. One flaw of this simple model is its monolithic approach. Most of the work is done on the server. While this may affect the performance of the system, it is best to keep the model relatively simple for reliability and ease of use.

b. Identifying Subsystems

i. UML Package Diagram



c. Mapping Subsystems to Hardware

The system will work on multiple computers. Some computers will be PCs, and others will be run on mobile devices. Since we do not require any type of sensor, all machines should be running a client and server subsystem.

d. Persistent Data Storage

The system needs to persist, saving management information and user accounts after logout or close of the program. Management information such as employee schedules or restaurant statistics need to be stored either locally or on the cloud. A history of changes and system backups will also need to be kept in case of system failure.

e. Network Protocol

This system will be using a MongoDB noSQL database which has the ability to create high performance and flexible databases. NoSQL databases are simpler in design, fitting into our model of a straightforward 2-tier client-server architecture. NoSQL queries are faster than traditional SQL databases. This allows multiple queries to the database to be processed at once. This is important for a restaurant, where many customers will be placing orders at any given time.

f. Global Control Flow

1. Execution orderness - This system will be for the most part a procedure-driven system, meaning it will be linear in its design. The users of the system from the restaurant side will always be required to authenticate themselves in order to see their corresponding interface. Then, the typical usage of the system will follow a linear path in which customers will come in, the host will use his or her interface to find the next available table according to the customer's party size, and the party will be seated when possible. From there, the customers will place orders through the tablets at the table which will push the orders to the chef. Once the food is prepared, the chef's will notify the chefs to take the food to the corresponding table. The one event driven feature in the system will be if the customer chooses to request the waiter come to their table for assistance. The system will be checking for an event where the event in this case is the request made by the customer. The rest of actions will again be procedure driven since the customers will finish by paying the bill through the system and a paid bill will notify the system that a table is ready to be cleaned which will notify the waiters.
2. Time dependency - The majority of the system will be running in real-time. The customer will place an order at some time and at the same time, the system will add this order to the chef's queue. The system will be based

around periodic actions from customers such as requesting a table, placing an order, and paying the bill as well as the employees working around their schedules in real-time. The one event driven response in the system will be the notification to the waiter when a customer requests their assistance or service.

3. Concurrency - The system will use multiple threads in order to handle all of the requests made. Multiple customers will be placing orders at the same time so the chef's queue will have to be updated with each of these orders at the same time. The solution is to have multiple threads to update the queue with each having an order to carry. There will have to be synchronization because the queue is a first-come-first-serve system so each order will be added according to the time it is placed. Also, there will be a thread running at the same time to maintain an updated inventory count. This will have to be synchronized because as soon as an ingredient is exhausted, the menu will have to be properly adjusted to remove any dishes that use this ingredient.

g. Hardware Requirements

EasyEats will most effectively increase the productivity of a restaurant if customers can carry the system on an android mobile device, and the employees have their choice of computer. Users can therefore use a smartphone, tablet or desktop computer.

	Samsung Galaxy S5	Samsung Galaxy Tab Pro 10.1	HP Elite Windows Desktop Computer
Component	Minimum Requirements	Minimum Requirements	Minimum Requirements
Platform	Android 4.4.2, KitKat	Android 4.4	Windows 8
Processor	2500 MHz Quad-core	2300 MHz Quad-Core	3 Ghz Core 2 Duo
RAM	2 GB	2 GB	4 GB
Hard Drive Space	32 GB	32 GB	160 GB
Network	LTE 150/50 Mbit/s	LTE 150/50 Mbit/s	WLAN 802.11 b/g/n
Screen Size	5.1''	10.1''	23''
Resolution	1080 x 1920	2560 x 1600	Intel HD Basic Graphics

V. Algorithms and Data Structures

a. Algorithms:

None applicable.

b. Data Structures:

In this system, a few different queues will be implemented. The waiter has a queue that lists an action the waiter should take on next. The queue is relatively complex because certain actions may take priority over others, such as when a customer calls for the waiter versus when drinks are ready to take to a table. This queue is designed to streamline the waiter's work, and give them a list to view in case they forget about anything. They simply remove what they've done from the queue simply and easily from their interface so they don't waste unnecessary time running through menus to access the option.

The hostess will have a priority queue to recall what tables are available for seating potential customers. At the beginning of the day, the queue contains all tables in the restaurant. Priority is organized in reverse order of size, so that a party receives the smallest table currently available for them. When a diner sits at a table, the table is removed from the queue and returned when the diner leaves. The nature of a queue means that tables are returned to the back of the list, so unless the restaurant is at maximum capacity, a diner will not get a just vacated table (no wait to be cleaned). If a table of the exact size of the party is not open, but a larger table is, the potential diner will take the larger table.

The chef/bartender share similar queues but they will be distinct to their position. The queue for them is relatively simple, consisting of no priority application. Once an order is placed, it is put in the queue and the chef/bartender will take care of it once it is next in queue. They can update the queue as necessary, since they are required to notify the waiter of any available order.

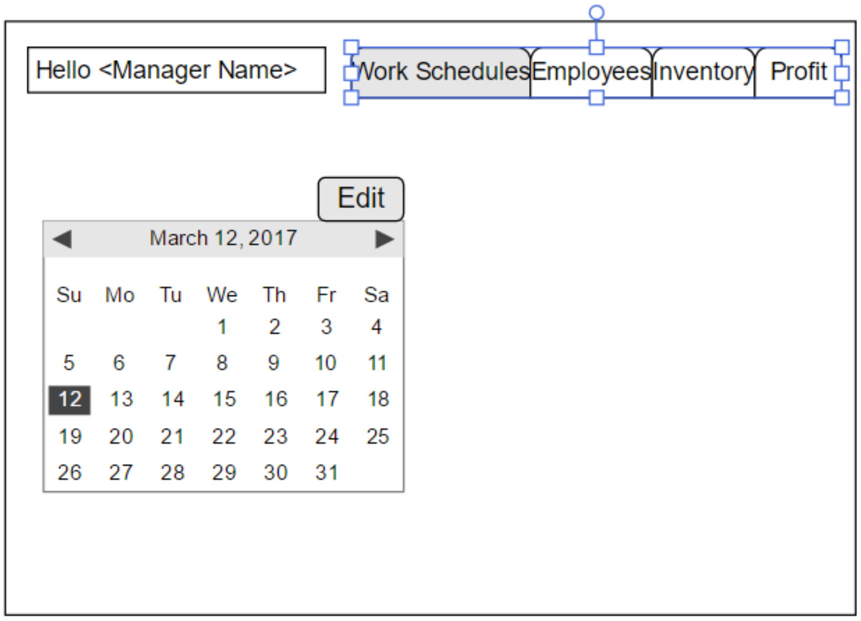
Menu items and inventory will be stored in arrays. Arrays are easily adaptable to certain tasks, and already have many different searching and sorting algorithms that increase performance and responsiveness of the menu system. Customers want the menu to respond quickly to their actions, especially when they filter the menu. The manager, and the chef/bartender all require the menu to be efficiently implemented when they are looking for a menu item to hide and add.

All data structures were chosen for performance over flexibility.

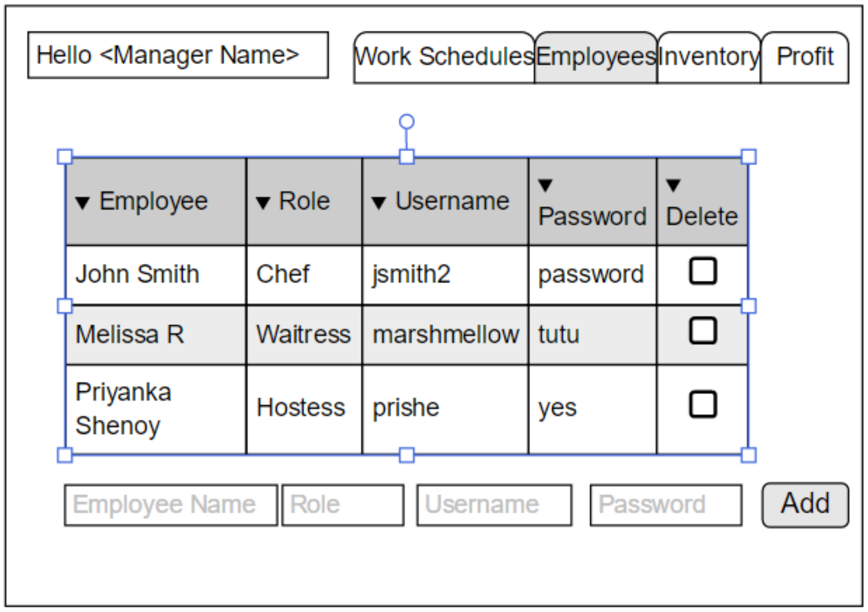
VI. User Interface Design and Implementation:

Our interface designs are subject to change during the future implementation, but we intend to keep the interface very similar to the previous screen drawings done in the previous report. A majority of the changes are going to be primarily aesthetics. We believe ease - of -use is maximized in the designs below.

a. Manager's Interface



Work Schedule Interface

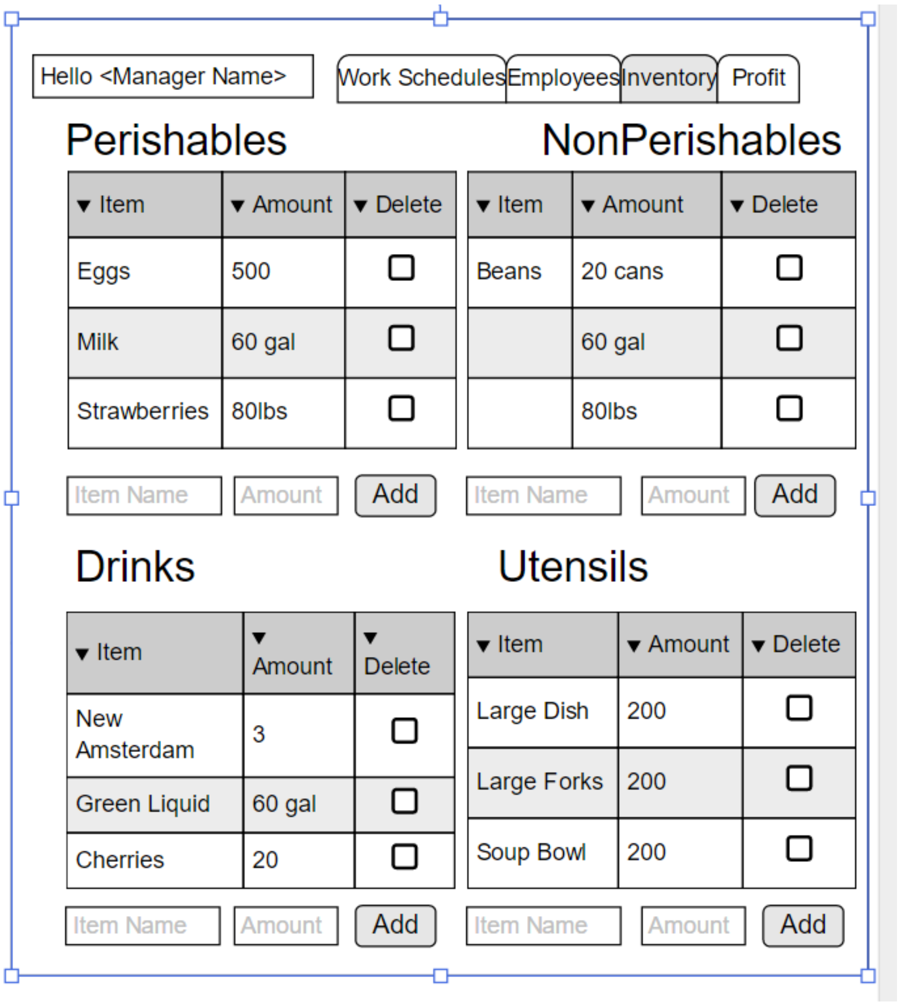

Hello <Manager Name>

Work Schedules | Employees | Inventory | Profit

▼ Employee	▼ Role	▼ Username	▼ Password	▼ Delete
John Smith	Chef	jsmith2	password	☐
Melissa R	Waitress	marshmellow	tutu	☐
Priyanka Shenoy	Hostess	prishe	yes	☐

Employee Name | Role | Username | Password | Add

Employees Interface



Hello <Manager Name>

Work Schedules | Employees | Inventory | Profit

Perishables

▾ Item	▾ Amount	▾ Delete
Eggs	500	☐
Milk	60 gal	☐
Strawberries	80lbs	☐

Item Name | Amount | Add

NonPerishables

▾ Item	▾ Amount	▾ Delete
Beans	20 cans	☐
	60 gal	☐
	80lbs	☐

Item Name | Amount | Add

Drinks

▾ Item	▾ Amount	▾ Delete
New Amsterdam	3	☐
Green Liquid	60 gal	☐
Cherries	20	☐

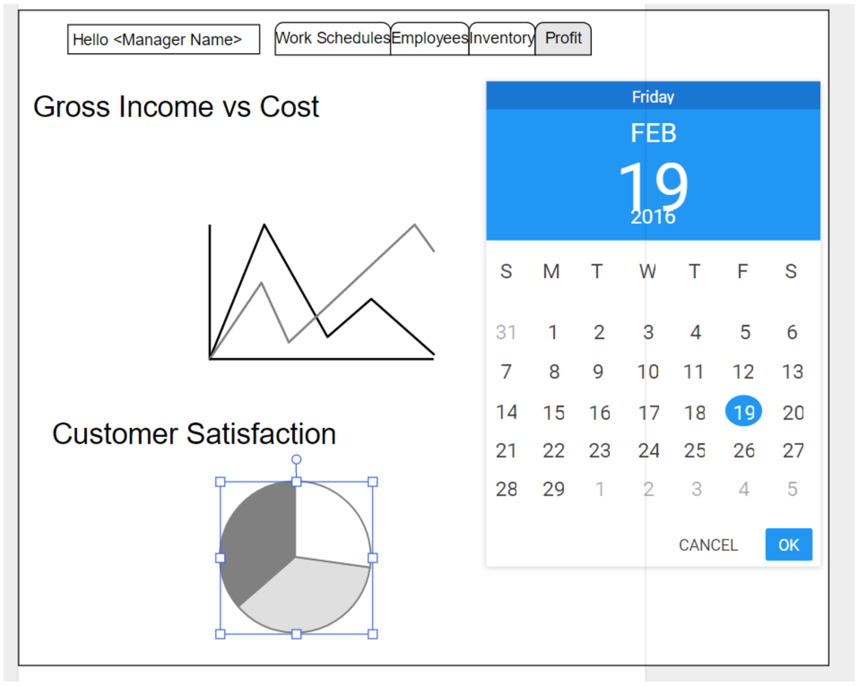
Item Name | Amount | Add

Utensils

▾ Item	▾ Amount	▾ Delete
Large Dish	200	☐
Large Forks	200	☐
Soup Bowl	200	☐

Item Name | Amount | Add

Inventory Interface



Hello <Manager Name>

Work Schedules | Employees | Inventory | Profit

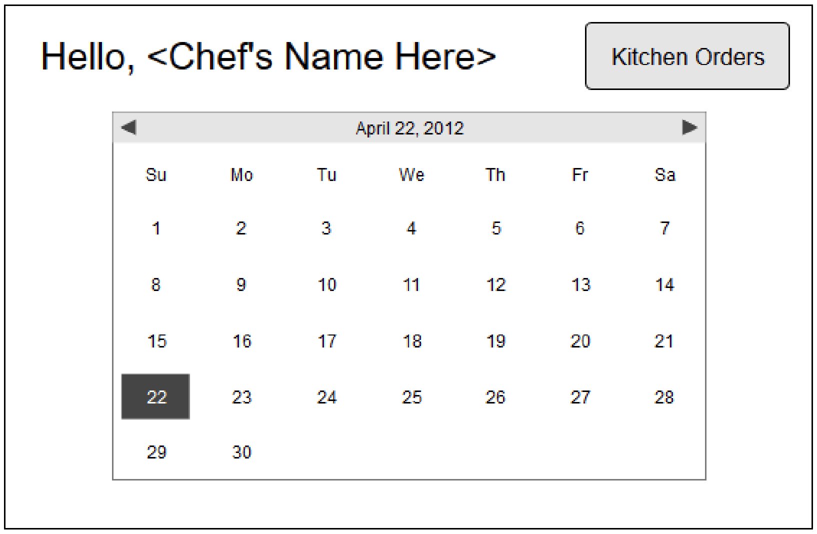
Gross Income vs Cost

Customer Satisfaction

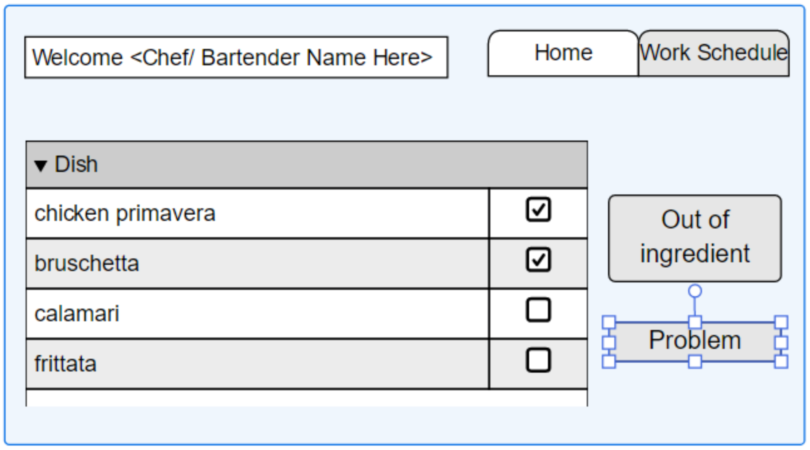
Profit Interface

The manager's interface homepage has remained the same, however, new sub-interfaces have been added in order to meet the requirements of the manager actor. The homepage is very simple in that it gives the manager the current day and a calendar to view future schedules. At the top right are tabs to select from various functions that the manager can perform. For the employee page, the manager has to go through minimal clicks in order to add or delete employees from the database. The same applies for the inventory tab and this section also allows for the manager to edit the inventory item counts. Lastly, the profits menu is very graphics oriented and displays all relevant information for the restaurant's profit/loss in visual representations for any given day.

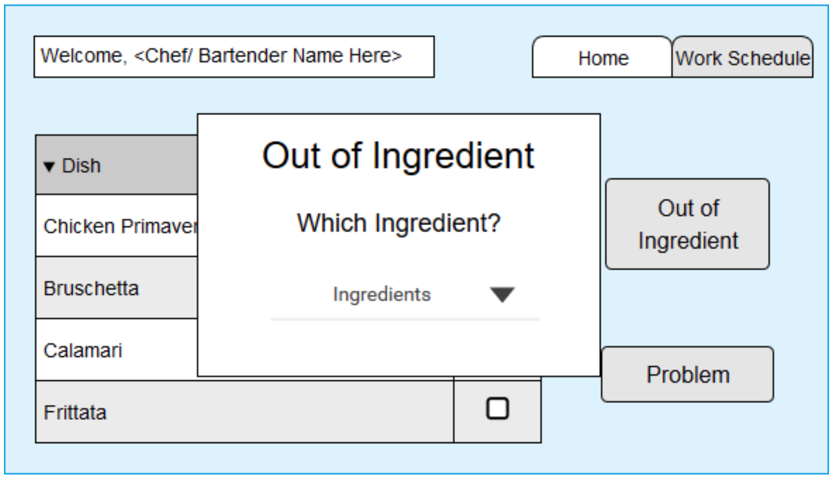
b. Chef Interface



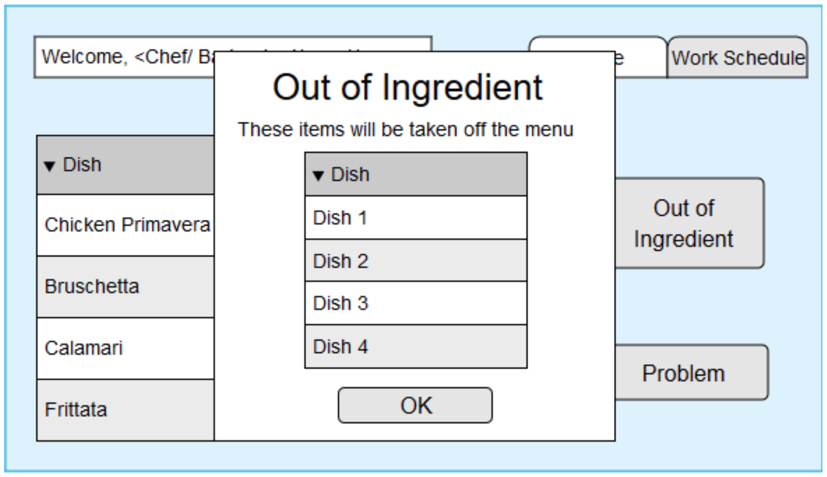
Work Schedule Interface



Home (Orders) Interface



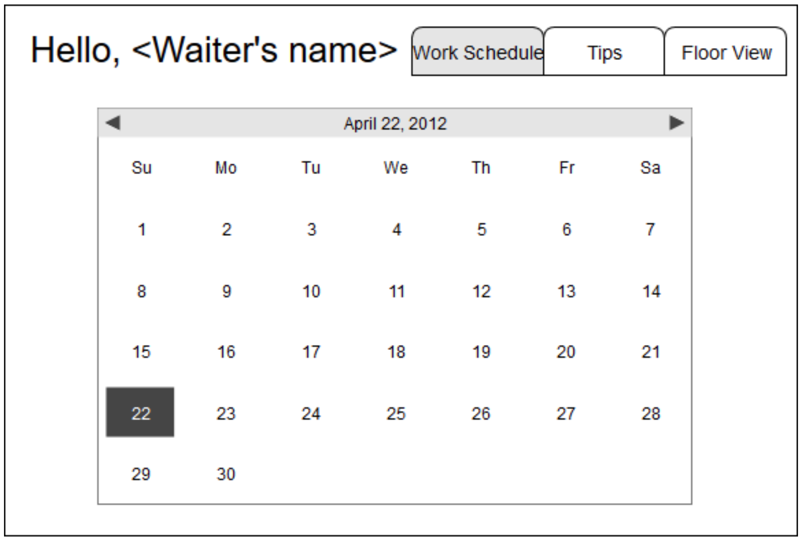
Out of Ingredient Pop-up



Out of Ingredient Confirmation Interface

The design of the chef interface has remained mainly the same. The chef will be busy making orders so we have prioritized minimizing the number of clicks required to do an action. Dishes will appear in a list along with a checkbox. The chef only has to press the checkbox to notify the waiter that the order is ready. The "Out of ingredient" screen is now a pop-out box instead of a new screen.

c. Waiter/Waitress Interface



1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

Work Schedule Interface

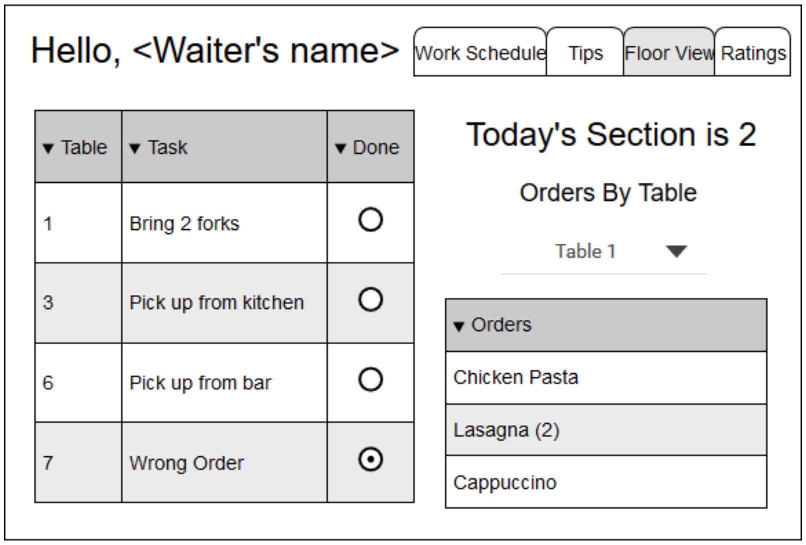
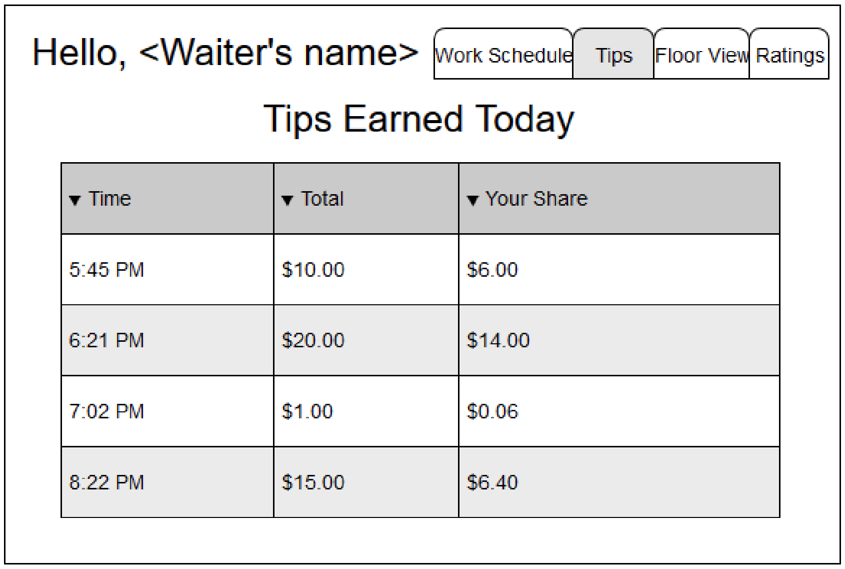


Table	Task	Done
1	Bring 2 forks	○
3	Pick up from kitchen	○
6	Pick up from bar	○
7	Wrong Order	⦿

Orders
Chicken Pasta
Lasagna (2)
Cappuccino

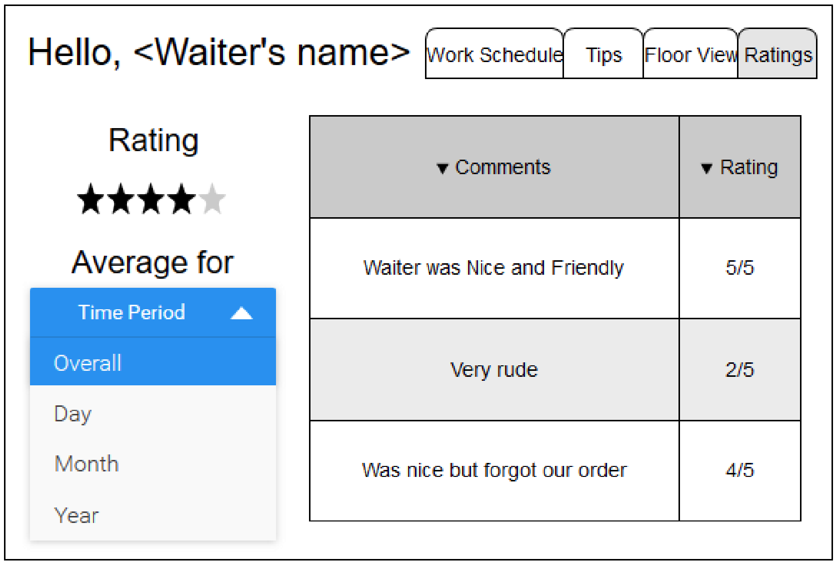
Floor View Interface



Tips Earned Today

Time	Total	Your Share
5:45 PM	$10.00	$6.00
6:21 PM	$20.00	$14.00
7:02 PM	$1.00	$0.06
8:22 PM	$15.00	$6.40

Tips Interface



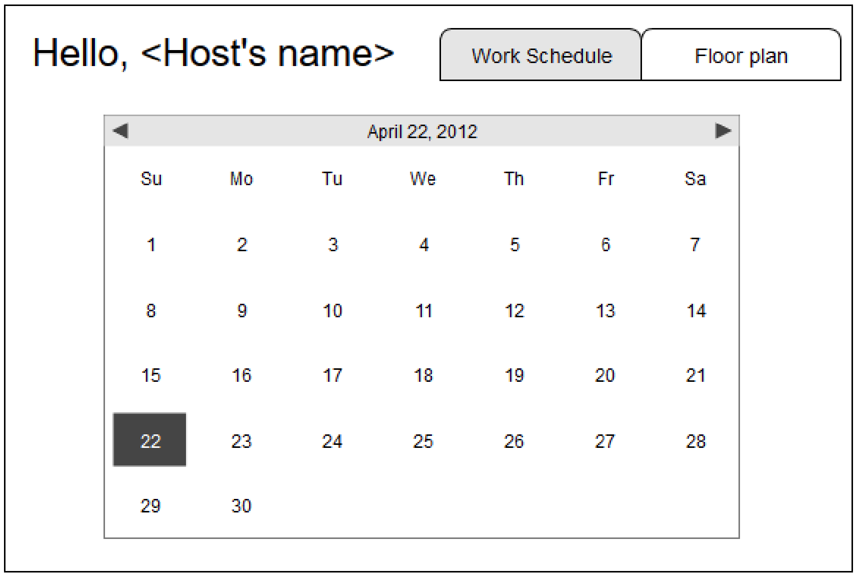
Hello, <Waiter's name>

▾ Comments	▾ Rating
Waiter was Nice and Friendly	5/5
Very rude	2/5
Was nice but forgot our order	4/5

Ratings Interface

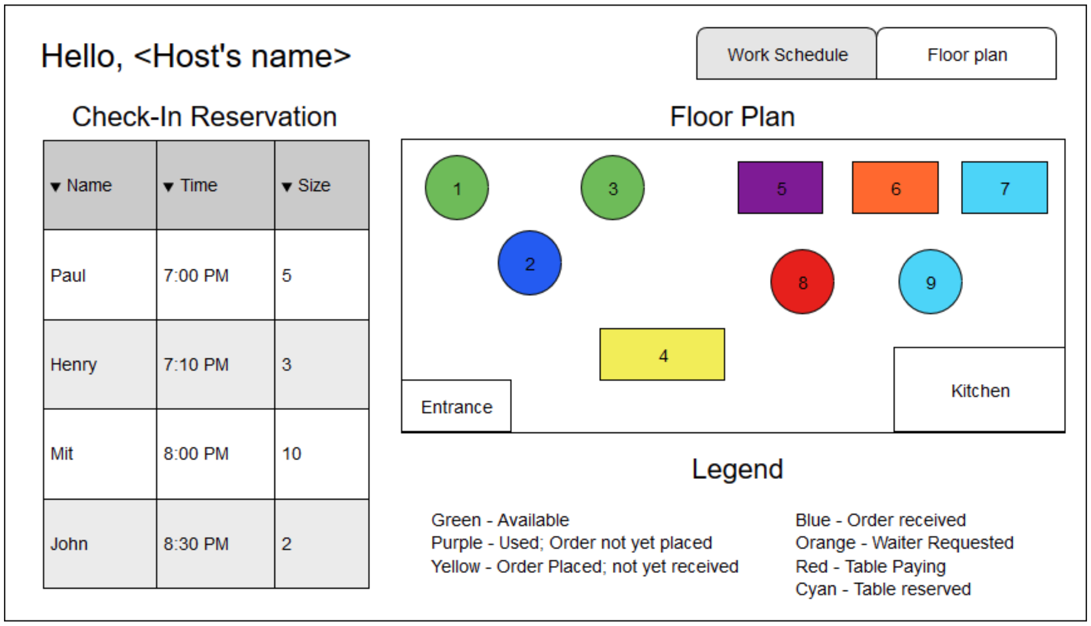
The waiter interface has remained mostly the same. The "Tips" interface now takes up the entire screen, allowing the waiter to view more tips earned. The average rating is now shown by stars. The right side of the screen will be a list of comments received by customers. When the waiter has done a task, he will press the checkbox next to the task, which will make the task disappear.

d. Host/ Hostess Interface



Su	Mo	Tu	We	Th	Fr	Sa
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

Work Schedule Interface


Hello, <Host's name>

Work Schedule | Floor plan

Check-In Reservation

▾ Name	▾ Time	▾ Size
Paul	7:00 PM	5
Henry	7:10 PM	3
Mit	8:00 PM	10
John	8:30 PM	2

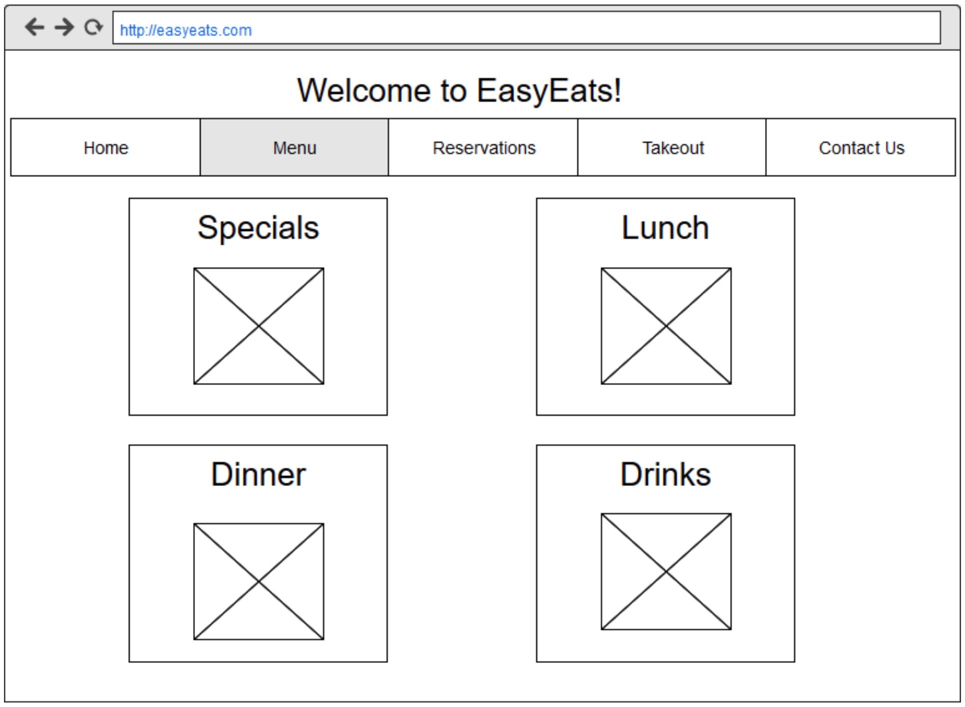
Legend

Green - Available
Purple - Used; Order not yet placed
Yellow - Order Placed; not yet received
Blue - Order received
Orange - Waiter Requested
Red - Table Paying
Cyan - Table reserved

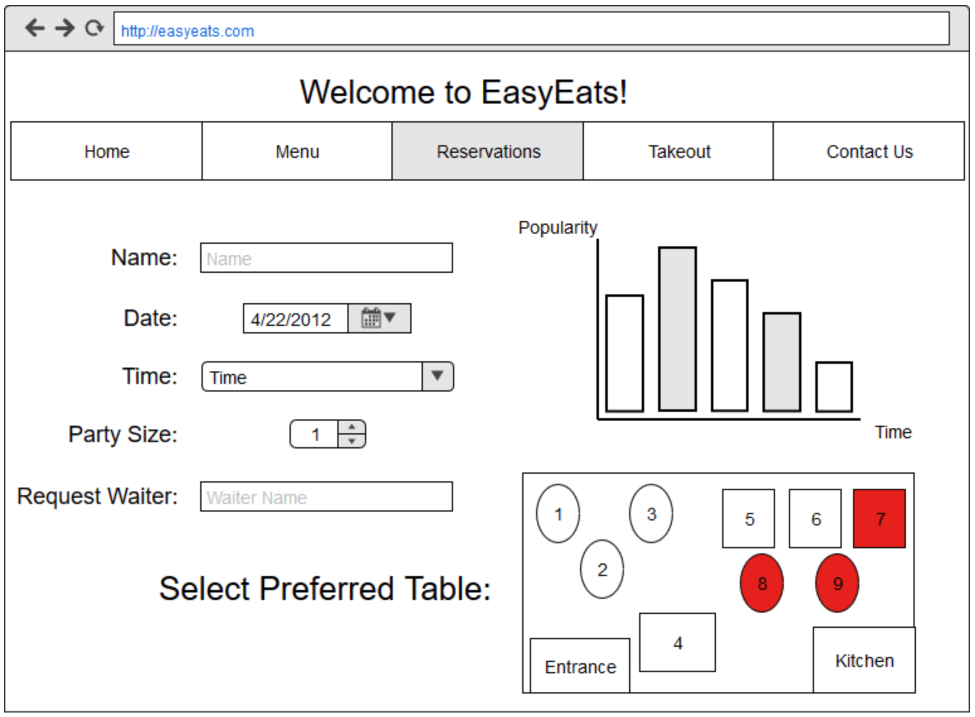
Floor Plan Interface

The host interface has seen no changes. The host will change the status of each table by picking a color. Some statuses, such as paying the bill or waiter requested, will be updated automatically without any input from the host.

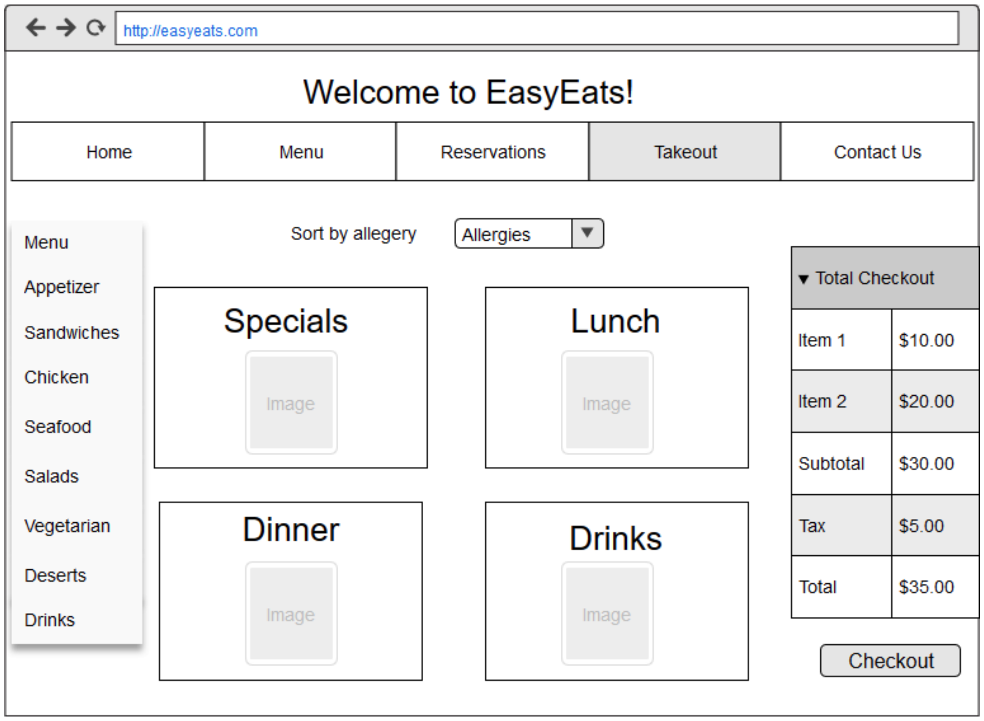
e. Customer Website Interface



Menu Interface

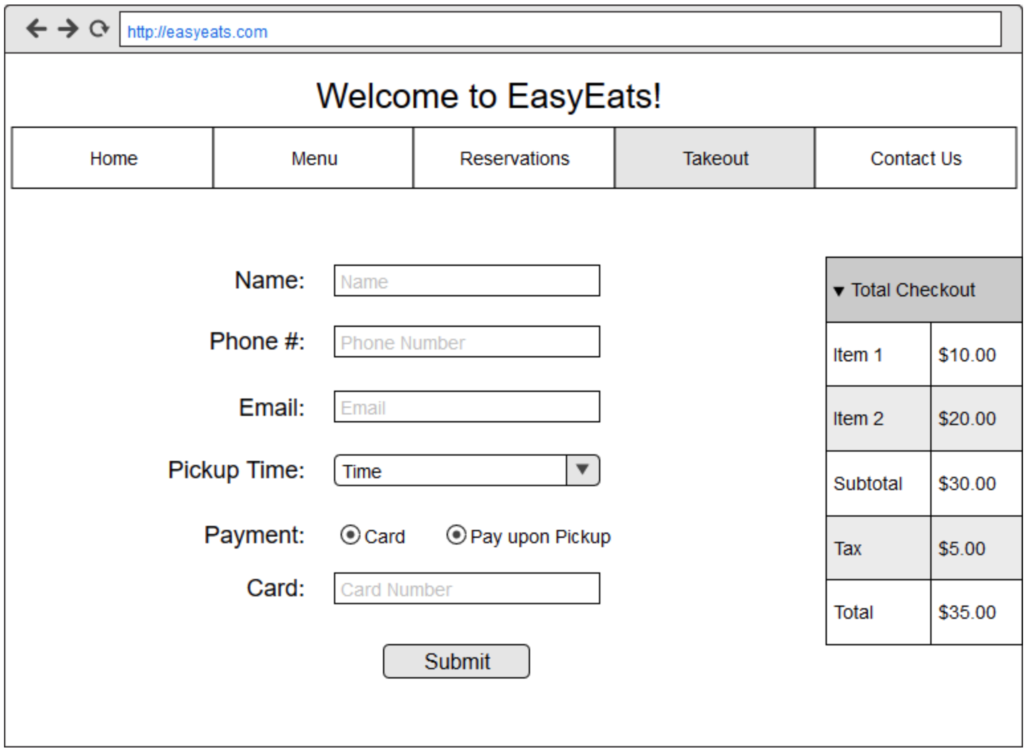


Reservations Interface



Item 1	$10.00
Item 2	$20.00
Subtotal	$30.00
Tax	$5.00
Total	$35.00

Takeout Interface



Name:	Name
Phone #:	Phone Number
Email:	Email
Pickup Time:	Time
Payment:	Card / Pay upon Pickup
Card:	Card Number

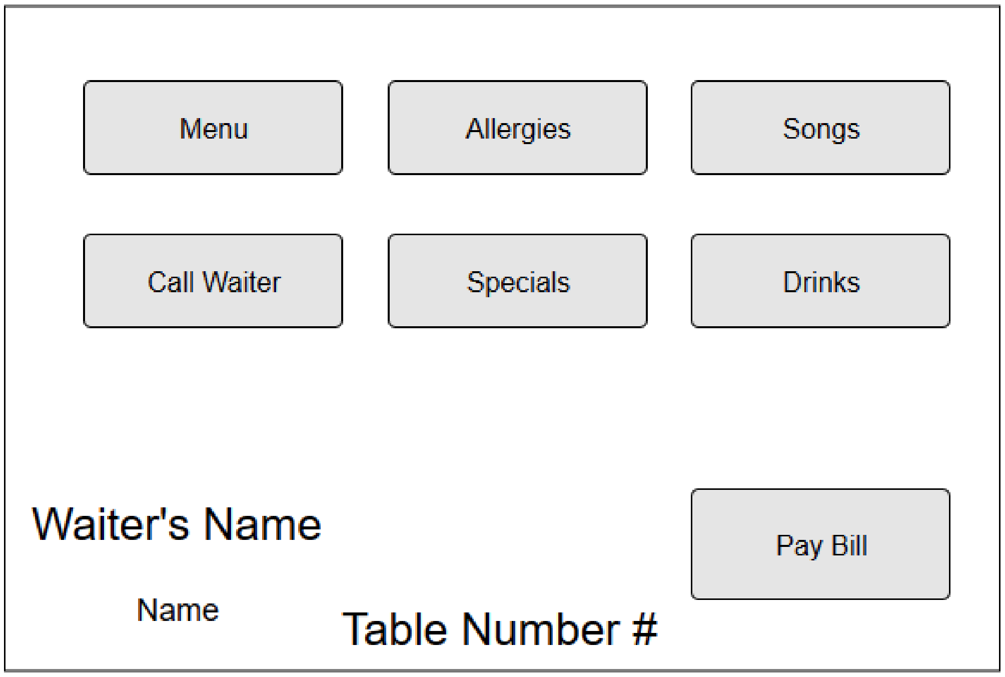
▼ Total Checkout

Item 1	$10.00
Item 2	$20.00
Subtotal	$30.00
Tax	$5.00
Total	$35.00

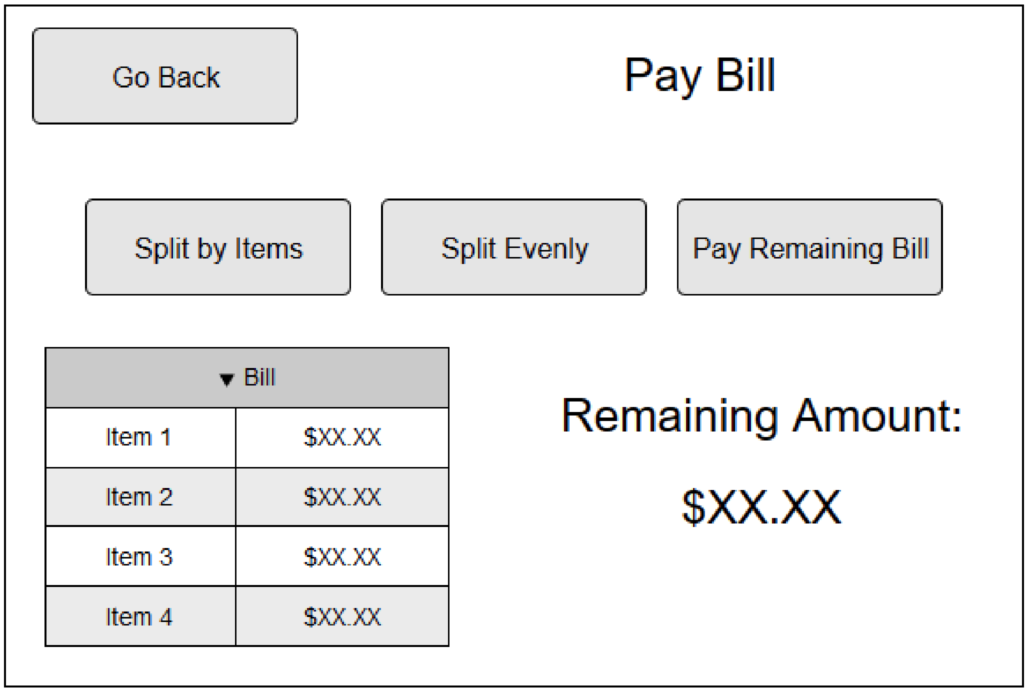
Takeout Customer Information Interface

The website's menu and reservation pages have remained mostly similar to the initial design. A new addition where a customer submits his/her information for the takeout order. This page is similar to other restaurants, where a customer puts down their name, phone number, email, pickup time, and payment method.

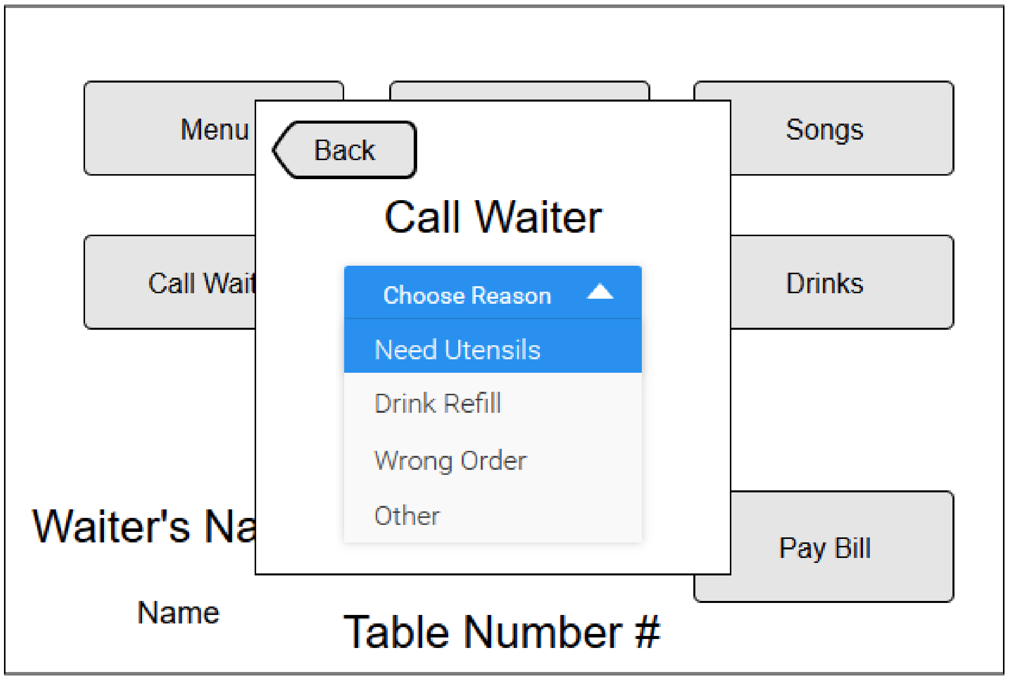
f. Diner Interface



Home Interface



Pay Bill Interface



Call Waiter Interface

The main change in the diner interface is the “Call Waiter” screen. It is now a pop-out box where the customer will select the reason with a dropdown menu.

VII. Design of Tests:

a. Test Cases:

i. Manager:

Test Case Identifier: TC-1
Function Tested: userAccess(int ID) : bool
Pass/Fail Criteria: Test will pass if the user is granted access to the manager interface.

Test Procedure	***Expected Results***
Call function (pass)	The user will be granted access to the manager interface.
Call function (fail)	If an incorrect ID is entered or if the ID already has access, then the function will return false.

Test Case Identifier: TC-2
Function Tested: assignShift(int ID, shiftinfo* in): bool
Pass/Fail Criteria: Test will pass if the shift is assigned and added to the schedule.

Test Procedure	***Expected Results***
Call function (pass)	The shift will be assigned to the correct ID and added to the schedule.
Call function (fail)	If the ID is already assigned that shift then the function will return false.

Test Case Identifier: TC-3
Function Tested: removeShift(int ID, shiftinfo* in): bool
Pass/Fail Criteria: The test will pass if the shift is successfully removed from the system.

Test Procedure	***Expected Results***
Call function (pass)	The shift will be removed from the system.
Call function (fail)	If the employee shift that is being deleted is the only employee on that shift, then the function will return false and the shift will not be deleted.

Test Case Identifier: TC-4
Function Tested: viewStats(): void
Pass/Fail Criteria: The test will pass if the overall restaurant stats are displayed to the user.

Test Procedure	***Expected Results***
Call function (pass)	The user is presented with the overall

	restaurant stats.
Call function (fail)	If there are currently no saved stats in the database, a message will output accordingly. If there is an error connecting to the database, a message will output notifying the user of the error.

Test Case Identifier: TC-5
Function Tested: inventoryAdd(item* it): bool
Pass/Fail Criteria: Test will pass if the item is added to the inventory.

Test Procedure	***Expected Results***
Call function (pass)	The item will be added to the inventory.
Call function (fail)	If an item with the same or similar name is already in the inventory, then the function will return false and the item will not be added.

Test Case Identifier: TC-6
Function Tested: menuAdd(string Name, double Price): bool
Pass/Fail Criteria: Test will pass if the item is added to the menu.

Test Procedure	***Expected Results***
Call function (pass)	The new item will be added to the menu.
Call function (fail)	If an item with the same or similar name already exists on the menu with the same price, the function will return false and the item will not be added to the menu. If an item with the same or similar name with a different price already exists on the menu, the function will change the price of the menu item and return true.

Test Case Identifier: TC-7
Function Tested: menuRemove(item* it): bool
Pass/Fail Criteria: Test will pass if the menu item is removed from the menu.

Test Procedure	***Expected Results***
Call function (pass)	The menu item is removed from the menu.

Call function (fail)	If the item does not exist in the menu, the function will return false and no changes will be made.

ii. Chef/Bartender:

Test Case Identifier: TC-8 **Function Tested:** viewOrder(): void **Pass/Fail Criteria:** The test will pass if the order queue is displayed to the Chef/Bartender.	
Test Procedure	***Expected Results***
Call function (pass)	The order queue is displayed to the user.
Call function (fail)	If there is no order queue saved in the database, a message will output accordingly. If there was an error with connecting to the database, a message will output notifying the user of the error.

Test Case Identifier: TC-9 **Function Tested:** updateOrder(order* or): bool **Pass/Fail Criteria:** Test will pass if the order specified is updated.	
Test Procedure	***Expected Results***
Call function (pass)	The specified order is updated.
Call function (fail)	If the order does not exist in the order queue, then the function will return false and no changes will be made.

Test Case Identifier: TC-10 **Function Tested:** menuHide(item* it): bool **Pass/Fail Criteria:** The test will pass if the specified menu item is hidden.	
Test Procedure	***Expected Results***
Call function (pass)	The menu item specified will be hidden.
Call function (fail)	If the specified menu item does not exist on the menu, the function will return false and no changes will be made.

iii. *Waiter/Waitress:*

Test Case Identifier: TC-11
Function Tested: cancelOrder(order* or): bool
Pass/Fail Criteria: The test case will pass if the order specified is cancelled.

Test Procedure	***Expected Results***
Call function (pass)	The function will cancel the specified order.
Call function (fail)	If the order to be cancelled does not exist in the order queue, then the function will return false and no changes will be made.

Test Case Identifier: TC-12
Function Tested: viewStats(): void
Pass/Fail Criteria: Test will pass if the waiter/waitress's personal stats are displayed to them.

Test Procedure	***Expected Results***
Call function (pass)	The waiter/waitress is presented with their personal stats.
Call function (fail)	If no stats currently exist in the server, then an appropriate message will be outputted to the user. If there was an error with connecting to the server, then a message will be outputted to the user notifying them of the error.

Host/Hostess:

Test Case Identifier: TC-13
Function Tested: tableStatus(tableID* table): void
Pass/Fail Criteria: Test will pass if the statuses of the tables are displayed for the host.

Test Procedure	***Expected Results***
Call function (pass)	The host/hostess is given the status of the specific table entered (available, ready to be cleaned, occupied, or reserved).
Call function (fail)	If the table ID entered was incorrect or there is no information available about the table entered, the function will display a message.

Test Case Identifier: TC-14 **Function Tested:** waitingQueue(): bool **Pass/Fail Criteria:** Test will pass if the the host/hostess successfully adds a new party to the waiting queue.	
Test Procedure	***Expected Results***
Call function (pass)	The host enters the details of the party requesting to join the queue (name and party size) and the details are successfully entered into the queue.
Call function (fail)	If the queue fails to accept the new item because of missing details such as name or party size, an error message will be displayed asking for full details.

Test Case Identifier: TC-15 **Function Tested:** editReservation(reserv* res): bool **Pass/Fail Criteria:** Test will pass if the reservation details are updated.	
Test Procedure	***Expected Results***
Call function (pass)	The host enters the reservation ID to change the details of that specific reservation. Details such as party size, name, or time can be changed. Also, the host may delete the reservation altogether.
Call function (fail)	If the reservation ID does not exist, then the function will display an appropriate error message.

iv. Potential Customer:

Test Case Identifier: TC-16 **Function Tested:** reserveTable(): bool **Pass/Fail Criteria:** Test will pass if table is available to reserve and the reservation is made.	
Test Procedure	***Expected Results***
Call function (pass)	The potential customer views the layout of the tables at the restaurant and chooses the table that can seat the desired party size and after entering details such as name and time, the

	reservation is made and updated in the system. A message will confirm this to the customer.
Call function (fail)	If the desired table is not available at the date and time the reservation was made for, a message will be displayed asking the customer to choose a different time.

Test Case Identifier: TC-17
Function Tested: menu(): void
Pass/Fail Criteria: Test will pass if the potential customer is shown the full menu.

Test Procedure	***Expected Results***
Call function (pass)	The potential customer is shown the full restaurant menu with the proper descriptions and prices for each menu item.
Call function (fail)	The test will fail if the menu fails to load part or all of the information.

v. Diner:

Test Case Identifier: TC-18
Function Tested: placeOrder(): void
Pass/Fail Criteria: Test will pass if the customer's order is sent to the chef.

Test Procedure	***Expected Results***
Call function (pass)	The customer places an order and the order successfully is placed in the queue of orders for the chef.
Call function (fail)	The test fails if after placing the order, the order does not show up in the chef's queue.

Test Case Identifier: TC-19
Function Tested: payBill(): bool
Pass/Fail Criteria: Test will pass if the due amount of the bill has been paid in full.

Test Procedure	***Expected Results***
Call function (pass)	This test applies only to paying with credit or debit. The customer(s) will swipe the card and

	if the payment went through and the bill was paid, a message will display confirming to the customer that the bill was paid.
Call function (fail)	If the card was not correctly read after being swiped or the charges could not go through, then an error message will be displayed explaining the problem.

Test Case Identifier: TC-20
Function Tested: waitressSignal(): void
Pass/Fail Criteria: Test will pass if the waiter/waitress receives a notification from the correct table.

Test Procedure	***Expected Results***
Call function (pass)	The waiter/waitress is shown a notification about the table that needs his/her assistance.
Call function (fail)	The test fails if the waiter/waitress does not receive the signal from the customer or if the notification points to the wrong table.

Controller:

Test Case Identifier: TC-21
Function Tested: DBConnection(): void
Pass/Fail Criteria: Test will pass if a connection to the database is successfully established.

Test Procedure	***Expected Results***
Call function (pass)	The user/class/object is connected to the database.
Call function (fail)	If there is an error and a connection cannot be established, then an appropriate message will output.

Test Case Identifier: TC-22
Function Tested: dataRequest(): void
Pass/Fail Criteria: Test will pass if the user is allowed to pull information from the database.

Test Procedure	***Expected Results***

Call function (pass)	The user will be allowed to pull information from the database.
Call function (fail)	If there was an error with the request or if the request was denied, an appropriate message will output. Reasons for request denial include multiple users requesting at the same time or database update lag.

Test Case Identifier: TC-23
Function Tested: dataUpdate(): void
Pass/Fail Criteria: The test will pass if the correct data is updated.

Test Procedure	***Expected Results***
Call function (pass)	The data is correctly updated.
Call function (fail)	If there is an error with the update or if the update is denied, an appropriate message will output. Errors include updates not being stored correctly or no changes being made during the update. Reasons for denial include too many users trying to access the database at once or database update lag.

Test Case Identifier: TC-24
Function Tested: dataAnalyze(): void
Pass/Fail Criteria: Test will pass if the the requested records are pulled from the database.

Test Procedure	***Expected Results***
Call function (pass)	The records that are requested to be pulled are pulled from the database.
Call function (fail)	If an incorrect record was pulled or no record was pulled the test will fail. The request will be denied if there are too many users making database request or if there is a lag in the database.

Test Case Identifier: TC-25
Function Tested: interfaceConnections(): void
Pass/Fail Criteria: The test will pass is a successful connection is established with the correct user interface.

Test Procedure	*Expected Results*
Call function (pass)	The database will be connected to the correct user interface.
Call function (fail)	If the database does not make a connection, an appropriate message will output. If a connection is made to the incorrect interface, then the test will fail.

vi. *Request Handler:*

Test Case Identifier: TC-26 **Function Tested:** processRequest(): void **Pass/Fail Criteria:** The test will pass if the correct request is sent to the controller at the correct time.	
Test Procedure	*Expected Results*
Call function (pass)	The next request in the queue will be sent to the controller when the controller has completed its current request.
Call function (fail)	If the incorrect request is sent to the controller or if a request is sent before the controller has finished its current request then the test will not pass.

Test Case Identifier: TC-27 **Function Tested:** getRequest(): void **Pass/Fail Criteria:** The test will pass if the correct request is retrieved from the user's interface.	
Test Procedure	*Expected Results*
Call function (pass)	The proper request is retrieved from the correct user interface.
Call function (fail)	If the incorrect request is retrieved the test will not pass. If the function completes and no request has been retrieved, then an appropriate message will output and the test will fail.

b. Test Coverage:

The tests are designed to check the functionality of the more important functions of the program. They check the functions that are necessary in order to run a very basic version of a restaurant and check whether all the proper information or data is being handled and processed correctly. The tests check the most important functions of each of the actors involved in running the program. Part of the testing involves checking to see if the databases correctly update information after some of the functions are called as well as checking if the employee queues are updated each time they are edited or added to.

c. Integration Testing Strategy:

Integration Testing aims to see faults when individual units are combined and tested as a group. After unit test cases pass, we will begin integration testing. We have determined that Big Bang testing is not ideal for our project since it would not necessarily garner enough information about faults. A Hybrid approach might be appropriate but only if more faults are in the middle our project than at one end. We chose a Bottom Up approach because our development begins at a foundational level. Sub teams should not wait until an entire solution, from backend to front end, is developed before integration testing.

Bottom Up testing will allow team members to ensure classes and class attributes interact properly according to previous documentation. When solutions by different sub teams are combined, bottom up testing will ensure that they interact properly. Some examples are provided below. The manager subteam will have to test that the diners are sending payment information to the manager data module. The diner sub team will have to test that when the chef alters the menu, the diner can only order from the current menu. The chef sub team will have to make sure that they can receive diner orders, and furthermore, that the waitress receives the pickup notice.

VIII. Project Management

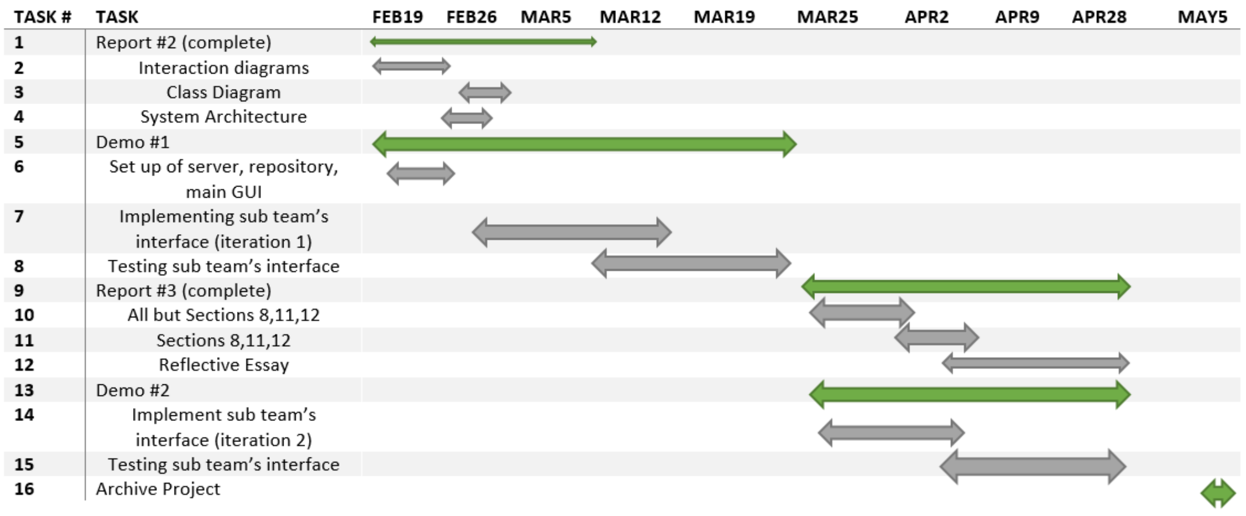
a. Merging the Contributions from Individual Team Members

Every member is contributes report work to any part of a google doc. A team member downloads the final google doc as a word document, edits it, and formats a pdf to turn in. Decisions are either discussed in person during group meetings, or online through GroupMe.

b. Project Coordination and Progress Report

Having a lot of work to complete by weekly deadlines, our team had to develop good time management. Besides being computer engineering students with high course loads, some of us have part time jobs, leadership positions in clubs, or are a commuter. Additionally, we had to self teach application programming skills, since no ECE course at Rutgers has taught us to create our own application.

c. Plan of Work



TASK #	TASK
1	Report #2 (complete)
2	Interaction diagrams
3	Class Diagram
4	System Architecture
5	Demo #1
6	Set up of server, repository, main GUI
7	Implementing sub team's interface (iteration 1)
8	Testing sub team's interface
9	Report #3 (complete)
10	All but Sections 8,11,12
11	Sections 8,11,12
12	Reflective Essay
13	Demo #2
14	Implement sub team's interface (iteration 2)
15	Testing sub team's interface
16	Archive Project

d. Breakdown of Responsibility

Each sub team is responsible for coding one solution cluster:

- Manager
  - Elizabeth:
    - Database driver class
    - Work schedule management
  - Prithvi:
    - Authentication class
    - Interface
- Waitress, Bartender/Chef, Hostess
  - Brett:
    - Database driver class
    - Chef/Bartender Interface
  - Kristen:
    - Host/Hostess Interface
    - Waiter/Waitress Interface
- Potential Customers and Diners
  - Tejas:
    - Database driver class
  - Mithu:
    - Interface
  - Raj:
    - Request Handler

IX. References

Restaurant Automation Group 3 2015

http://www.ece.rutgers.edu/~marsic/Teaching/SE/report2.html

http://www.ece.rutgers.edu/~marsic/Teaching/SE/report1.html

www.gliffy.com

https://moqups.com/

http://www.ziosk.com/