Santa Clara University
Scholar Commons

Computer Engineering Senior Theses | Engineering Senior Theses

6-8-2015

OmniSplit: a mobile food ordering platform for restaurant staff and patrons

Andres de Artola
Santa Clara University

Jordan Buschman
Santa Clara University

Ashley Sehatti
Santa Clara University

Follow this and additional works at: https://scholarcommons.scu.edu/cseng_senior

Part of the Computer Engineering Commons

Recommended Citation

de Artola, Andres; Buschman, Jordan; and Sehatti, Ashley, "OmniSplit: a mobile food ordering platform for restaurant staff and patrons" (2015). *Computer Engineering Senior Theses*. 47.
https://scholarcommons.scu.edu/cseng_senior/47

This Thesis is brought to you for free and open access by the Engineering Senior Theses at Scholar Commons. It has been accepted for inclusion in Computer Engineering Senior Theses by an authorized administrator of Scholar Commons. For more information, please contact rscroggin@scu.edu.

SANTA CLARA UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

Date: June 8, 2015

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

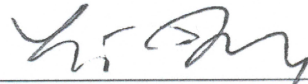
Andres de Artola
Jordan Buschman
Ashley Sehatti

ENTITLED

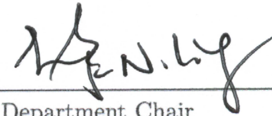
OmniSplit: A Mobile Food Ordering Platform for Restaurant Staff and Patrons

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREES OF

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING
BACHELOR OF SCIENCE IN WEB DESIGN AND ENGINEERING



Thesis Advisor



Department Chair

OmniSplit: A Mobile Food Ordering Platform for Restaurant Staff and Patrons

by

Andres de Artola
Jordan Buschman
Ashley Sehatti

Submitted in partial fulfillment of the requirements
for the degrees of
Bachelor of Science in Computer Science and Engineering
Bachelor of Science in Web Design and Engineering
School of Engineering
Santa Clara University

Santa Clara, California
June 8, 2015

OmniSplit: A Mobile Food Ordering Platform for Restaurant Staff and Patrons

Andres de Artola
Jordan Buschman
Ashley Sehatti

Department of Computer Engineering
Santa Clara University
June 8, 2015

ABSTRACT

The takeout industry has benefitted greatly from smartphone technology, but the dine-in experience has lagged behind. There are several major issues with the current dining experience, including how to split the check and how to address the issue of poor customer feedback. Some mobile and desktop apps have tired to address individual issues of the restaurant dining process, but there is no single platform that attempts to fix the experience as a whole.

We begin by outlining the idea behind OmniSplit, a small to medium business solution that seeks to address many of these issues simultaneously. OmniSplit combines online food payments, individual item ratings, and splitting the check all in one system. We implemented a web application for restaurant staff members and an iOS app for restaurant customers. With the web application, restaurant staff can customize their menu and respond to incoming orders and payments. The platform can be expanded on in the future to include additional features, such as the ability to add images to a menu. The iOS app allows users to see the menus of restaurants hosted on OmniSplit, and gives them the option to order food in a group with their friends. At the end of ordering, users can leave ratings for the food they ate. We discuss what we can add in the future to expanding our system, including the ability to tip, ability to refund payments, and a way to view restaurant analytics. Finally, we discuss the societal impact of our system and conclusions we learned during implementation.

Table of Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Current Solutions . . . 2
- 1.3 Solution . . . 2

2 Requirements **4**
- 2.0.1 iOS App . . . 4
- 2.0.2 Web Application . . . 5
- 2.1 Non-Functional Requirements . . . 5
- 2.2 Design Constraints . . . 6

3 Design **7**
- 3.1 Conceptual Models . . . 7
- 3.2 Initial Architectural Diagram . . . 7

4 Implementation **12**
- 4.1 Final Architectural Diagram . . . 12
- 4.2 Technologies Used and Design Rationale . . . 13
  - 4.2.1 iOS App . . . 13
  - 4.2.2 Web Application . . . 14
  - 4.2.3 Server . . . 14
  - 4.2.4 Databases . . . 15
- 4.3 Use Cases . . . 15
  - 4.3.1 iOS App . . . 15
  - 4.3.2 Web Application . . . 18
- 4.4 Databases . . . 20
  - 4.4.1 MongoDB . . . 20
  - 4.4.2 Parse . . . 21
- 4.5 Security . . . 22
- 4.6 Development Timeline . . . 22
- 4.7 Testing . . . 23
  - 4.7.1 Alpha Testing . . . 23
  - 4.7.2 Beta Testing . . . 23

5 Societal Issues **27**
- 5.1 Ethical . . . 27
- 5.2 Social . . . 27
- 5.3 Political . . . 27
- 5.4 Economic . . . 28
- 5.5 Health and Safety . . . 28
- 5.6 Manufacturability . . . 28
- 5.7 Sustainability . . . 29
- 5.8 Environmental Impact . . . 29

5.9 Usability 29
5.10 Lifelong Learning 29
5.11 Compassion 29

6 Conclusion **30**
6.1 Report Summary 30
6.2 Lessons Learned 30
6.2.1 Planning 31
6.2.2 Scope Creep 31
6.2.3 Use of Github 31
6.3 Moving Forward 31
6.3.1 Analytics 31
6.3.2 Tipping and Refunding Payments 32
6.3.3 QR Codes 32
6.3.4 Geofencing 32
6.3.5 Search Functionality 32

7 Appendix **33**
7.1 Additional Diagrams 33

List of Figures

3.1 Initial iOS App Ordering Page 8
3.2 Initial iOS App Payment Splitting Page 9
3.3 Initial Web App Incoming Orders Page 10
3.4 Initial Web App Menu Customization Screen 10
3.5 Initial Architectural Diagram 11

4.1 Final Architectural Diagram 12
4.2 Active Parse Database 21
4.3 Gantt Chart for Fall Quarter 2014 24
4.4 Gantt Chart for Winter Quarter 2015 25
4.5 Gantt Chart for Spring Quarter 2015 26

7.1 iOS State Diagram 33
7.2 Web State Diagram 34

Chapter 1

Introduction

1.1 Motivation

Imagine a group of five friends eating at a restaurant. In this hypothetical order, three of the five friends decide to share an appetizer. One person keeps ordering drink after drink while everyone else is drinking water. Some people order appetizers and all of the entres end up costing different amounts. To make things even more complicated, the appetizers are only shared between certain people. Ultimately, no one is quite sure how to pay when the bill comes. The easiest way would be to split the bill five ways, but nobody would be paying for what they actually ordered. So the group opts to calculate how much each person owes and for each person to pay that amount. Many people no longer carry cash, so the group ends up writing the last four digits of each credit card along with the amount to charge each person on the back of the bill.

Now that everyone has figured out how to pay the bill, the group leaves and a new problem arises. No one leaves a review on Yelp or takes the restaurants survey because their experiences were satisfactory and leaving a review would be too time consuming. People typically only go out of their way to review restaurants when they have had either extremely excellent or poor dining experiences.

Even in this short example, we can see some of the flaws of the current dining experience. Whereas takeout has benefited greatly from mobile ordering and payment solutions, dine-in restaurants have been slow to adopt these technologies, causing the system to become outdated and inefficient. In the past few years, many companies have emerged that seek to enhance the in-house dining experience. Some of the most popular food-related services include Venmo for micro-payments and Yelp for restaurant reviews.

1.2 Current Solutions

As stated previously, Venmo is an application that allows for transactions between users. It requires users to initially enter their credit card or bank information and saves it so that money can be sent to users directly, rather than having to deposit cash and pay people back. A downside to Venmo is that transfers to bank accounts do not go through quickly and often someone has to front the money for the entire bill if they are using it to pay for their portion of the meal at a restaurant.

Yelp is currently the most widely used application for reviewing restaurants. It allows users to rate food locations as well as comment on a restaurants service. Unfortunately, Yelp does not have the option to review individual dishes, but rather the restaurant as a whole. This makes it difficult to know a restaurants true rating because many restaurants have good dishes and not so good dishes. Yelp is specifically used for ratings as well as information about restaurants and that is unfortunately where its usefulness ends.

Other applicable mobile ordering services include Tapingo and Eat24. Tapingo allows users to order remotely for take out, while Eat24 lets users have their order be delivered to them. These two options are great for take out and just ordering food, however they do not allow for an in-house dining experience. These services are unable to accommodate the issue of splitting checks among friends while eating at a restaurant.

1.3 Solution

Obviously, each of the current solutions solve a unique issue present in the dining industry, but each service has a significant downside and none solve the entire issue the industry is facing. In response, we decided to develop OmniSplit, a service aimed at small to medium sized restaurants which seeks to do just that. OmniSplit combines customer payments, food ratings, and online ordering all into one system. There are two parts to the system: an iPhone app and a web application. The iPhone app allows users to see what restaurants in the area have signed up with OmniSplit and order from those restaurants using a variety of payment methods. When one first launches the app, they see a list of restaurants in the area. When a group of users enter a listed restaurant, they can then enter a table number to allow them to form a group. The group orders through the mobile app, and once everyone has submitted their order, a waiter or waitress who will bring the food to the table. After eating, the group can then split the check in any way they please.

In order for a restaurant to benefit from OmniSplit, a restaurant owner first has to register their restaurant on the OmniSplit website. Doing so gives the restaurant owner access to a wide variety of online tools and puts their restaurant publicly up on the OmniSplit mobile app. Tools include an editor to customize their iPhone menu and and a live feed of events including customer orders and received payments.

In addition to providing an end-to-end solution to restaurants, OmniSplit offers several benefits over the current existing solutions. First, the menu customizer allows restaurants to design and style their listed menu to match with the look and feel of their restaurant. This allows restaurants to make a customized menu without having to hire someone to design a pricey restaurant app. Second, our app will combine the functionality of services like Venmo with mobile ordering solutions like Tapingo. Individuals can order food as a group, but each individual can control what they do and dont want to pay for. Each user will get a detailed receipt showing what they paid for, and the restaurant will get a summary of all the orders. In the future, our platform can be easily expanded to allow for refunds to either an individual or the ordering group as a whole. Finally, after everyone has paid for their food, the user will be encouraged to rate each food item they purchased with a 1-to-5 star rating. This not only gives restaurants a better idea of how well food items are recieved by customers, but also allows our system to rank restaurants more accurately that Yelp, which only allows users to rate the restaurant as a whole.

Chapter 2

Requirements

We decided that for our system, we would design a web app for restaurant staff members and an iOS app for restaurant customers. In order for our solution to meet the needs of both restaurant owners and consumers, we have constructed a list of requirements that both components of our system must meet. First are the functional requirements, which define what the system must do. Next are the non-functional requirements, which define the manner in which the functional requirements must be met. Lastly are the design constraints, which limit the way that the application is to be designed and implemented.

2.0.1 iOS App

The mobile app user (a restaurant customer) must be able to:

- Create an Orderly account.
- Log in/out of their Orderly account.
- Browse a selected restaurants menu.
- Enter a table's identification code to join an ordering group.
- Add food selections from a menu to an order.
- View entire ordering groups order.
- Pay for specific food items in an order.
- Split the cost of items among some or all other users in an ordering group.
- Leave a review after finishing order placement.

2.0.2 Web Application

Additionally, we have concluded that a web application user (a restaurant worker) must be able to:

- Create an Orderly account.
- Log in/out of their Orderly account.
- Create a restaurant and menu.
- Customize their restaurants menu.
- View various analytical measures about the restaurants customers and orders.
- View orders waiting in the queue and completed orders.
- Mark orders as completed.

2.1 Non-Functional Requirements

- The iOS app should work on all recent iPhones, from the iPhone 5 through iPhone 6/6 plus.
- The mobile app should work on all most browsers, namely all updated versions of Safari, Firefox, and Chrome.
- The back-end system (databases, file storage, and servers) should be easy to scale up in case of rapid user base growth.
- Restaurant customers should be able to pay with most major credit cards (if on an iPhone 6/6 plus).
- Our implemented payment method should not have high transaction fees.
- The web application should be a single-page web app, and going to different tabs should not require navigating to a new page.
- Both the iOS app and web application should be:
  - User friendly – easy and intuitive to use.
  - Secure – no data should be at high risk for being stolen.
  - Responsive – users should not have to wait a long time for data to be processed.

2.2 Design Constraints

- The total cost of the system must not exceed our budget ($1,072.79). This was the amount we recieved in grant money from the university.
- The project must be completed by the deadline (Week 10, Spring Quarter 2015).

Chapter 3

Design

3.1 Conceptual Models

A conceptual model shows what the graphical user interface for a system will eventually look like. Figure 3.1 and fig. 3.2 are the conceptual models for the Orderly iOS app. We envisioned that after after logging in, all of the views for the iOS app will have a static navigation bar on top of the window and a static tab bar on the bottom of the window. The top navigation bar will have an icon for to view items currently in your cart. The bottom tab view bar will contain important menu items, including settings, selected payment method, and more. Figure 3.1 shows what we initially envisioned a restaurant's menu would look like. From this screen, users can add menu items to their cart to prepare for checkout. Figure 3.2 shows a conceptual model for the checkout screen, which a user will arrive at when they have finished ordering food and are ready to pay. On this view, users should be able to either pay for what they ordered or enter a dollar amount that they want to pay.

Figure 3.3 and fig. 3.4 are the conceptual models for the web application. The web application will be a single page web app, so all of the tabs can be clicked on without having to load a new page. Figure 3.3 shows the orders tab, which will have a real-time feed of completed and pending orders for the restaurant. On this menu, restaurant staff can see orders that have been placed and mark them as completed. Figure 3.4 shows the menu customization tab. On this tab, restaurant staff should be able to customize the menu that users see when they go to the restaurants menu on the iOS app.

3.2 Initial Architectural Diagram

An architectural diagram is meant to show the high-level structure of a program or system. Figure 3.5 shows our initial architectural diagram for OmniSplit. Pictured are the

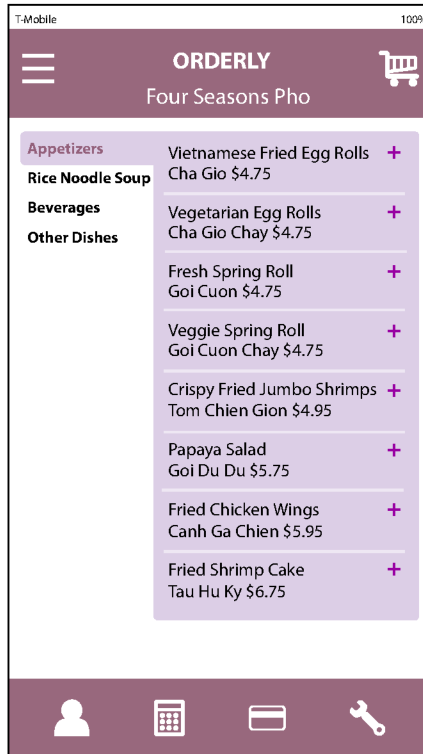


Figure 3.1: Mock-up for the iOS app payment ordering page.

three components that we planned on implementing for our system – the web app, the iOS app, and the back-end system (servers, file storage, and database) – and how the systems components interact with each other.

We planned for the iOS app to run natively on iOS devices, and the web app to run in a web browser of a computer or a mobile device (such as an iPad). Both of these components run code client-side and are not planned to run on the server. At their cores, both the iOS app and the web app will were designed with a model-view-controller (MVC) architecture. The view is what users see on their screen and what responds to change in the system. The model provides the structure of the system and can be updated by the controller. The controller is the logic that determines how the view and model interact and are updated. iOS apps are typically programmed using a MVC architecture, and websites can use MVC architectures to create complex single-page web applications like that of OmniSplit.

We planned on having two servers: a web server and an analytics server. Our web server is supposed to allow for interaction between the different components of the system and for clients to access back-end components. Through the web server, clients should also able to communicate with each other. The analytics servers function is to analyze the database

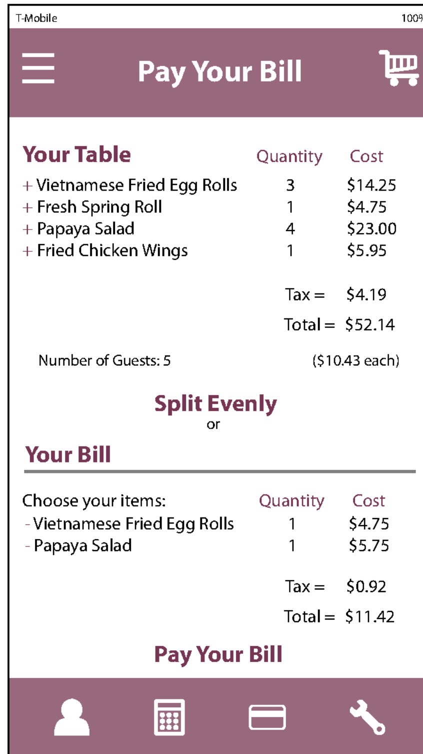


Figure 3.2: Mock-up for the iOS app payment splitting page. Here, one can split food items among everyone in their ordering group or among specific users.

and provide an interface for any components that want to analyze some of the database information. Both servers should be able to interact with our database, which we designed to contain text data such as login info (usernames, passwords), and info for analytics (order info, visited restaurants, ratings, etc.). The web server should be able to additionally access our file storage, which contains all of our larger assets (menu themes, user uploaded pictures, etc.). Both the web app and the iOS app can should be able to request permission from our web server to so they can indirectly interact with the file storage.

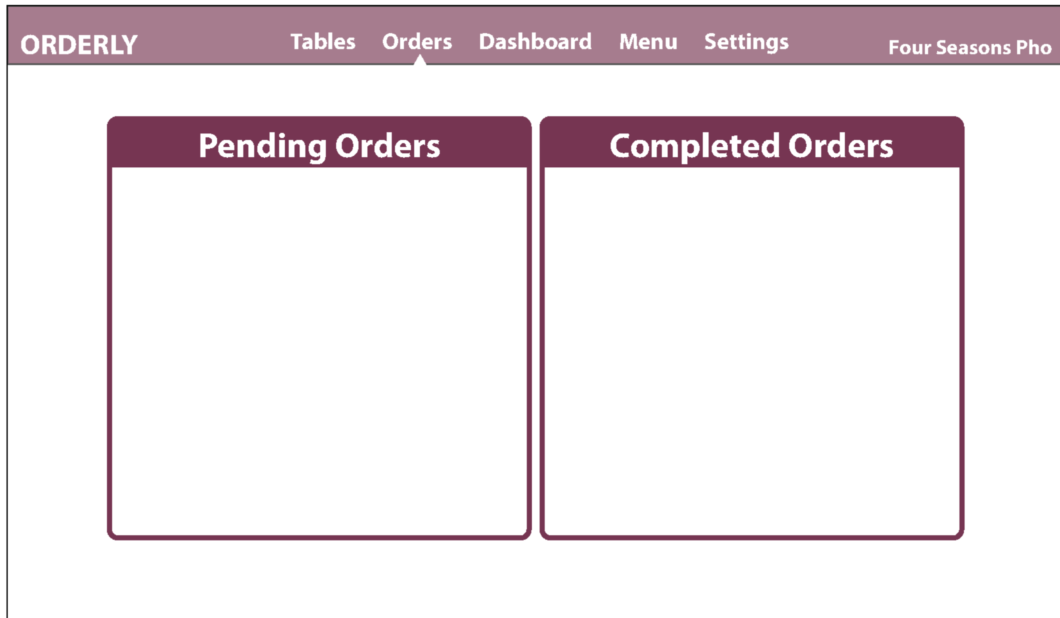


Figure 3.3: Mock-up for the web app incoming orders page. One can see orders coming and and mark them as completed.

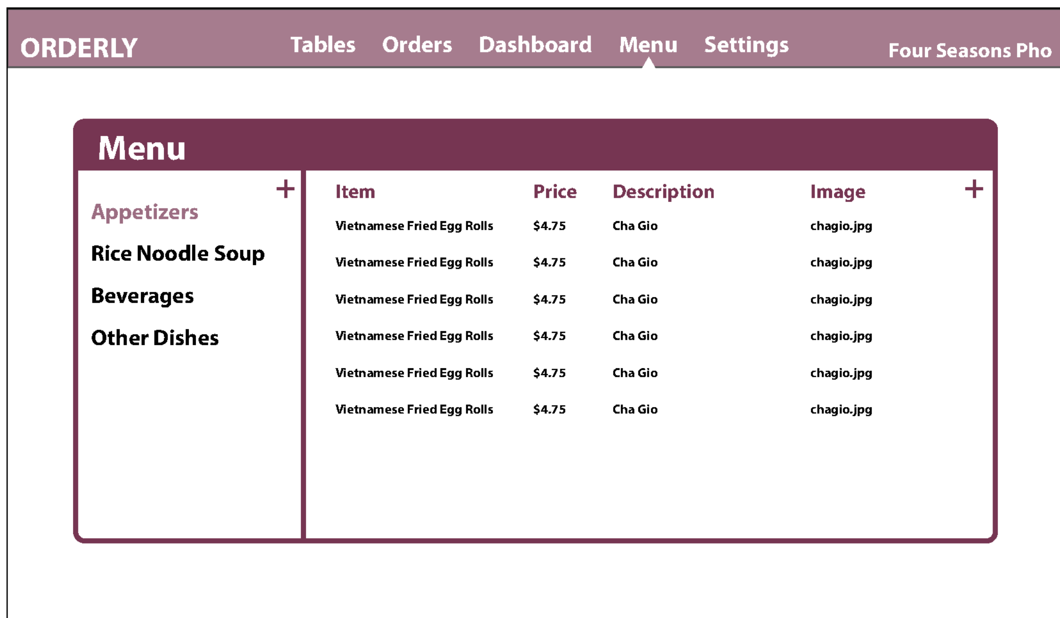


Figure 3.4: Mock-up for the web app menu customization screen. You can change the order and listing of food items here.

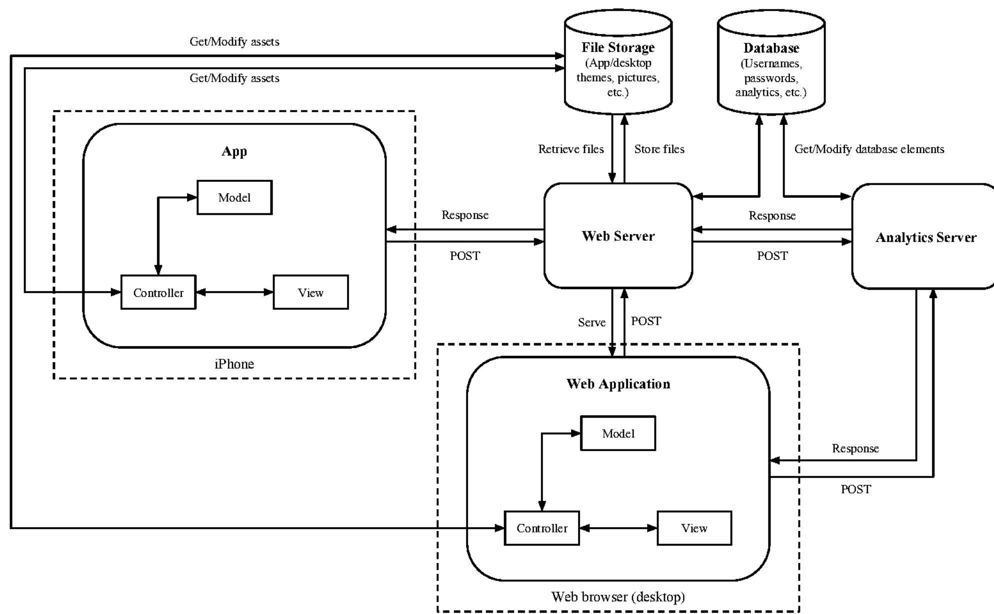


Figure 3.5: Original architectural diagram, showing the high-level layout of the OmniSplit system. Pictured are the web app (bottom), iOS app (left), and data stores (top right).

Chapter 4

Implementation

We began testing different methods of implementation during winter break and started implementing our design in winter quarter 2015 (full implementation timeline is included in section 4.6).

4.1 Final Architectural Diagram

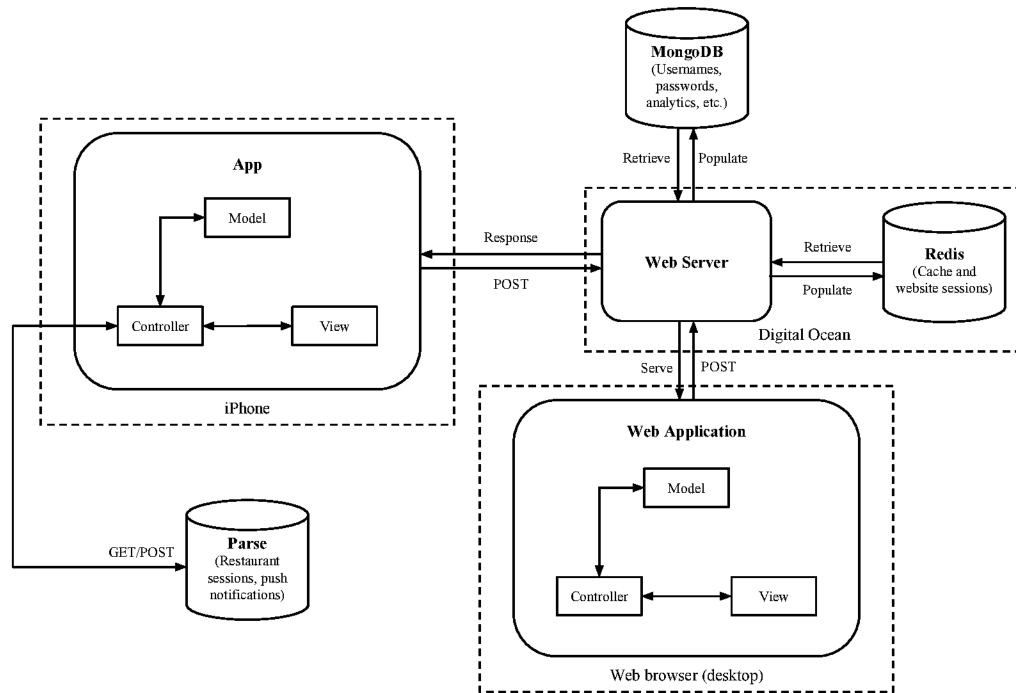


Figure 4.1: Architectural diagram of the final system, showing the high-level layout of the OmniSplit system. Changes include the addition of a Parse database and a redis web cache.

Figure 4.1 shows the design of our final system. The majority of our implementation followed the initial design. both the web app and the iOS app are running model-view-

controller architectures, we are using a MongoDB database for our main datastore, and components can communicate with the database through our web server. We programmed the iOS app with Objective-C, and used node.js to program the web server.

However, there are several key differences between our initial design and final implementation. Whereas the UI for the web app is virtually the same as in the mock-ups, the UI for the web app was completely changed. Additionally, we removed our file storage for images, as we decided not to allow for addition of images to the menu in our implementation.

As we did not implement analytics, we did not purchace a second server for analytics, but rather stuck with one server. However, we installed a redis cache to the web server, which allowed for us to cache key-value pairs on the server itself rather than having to go to the remote MongoDB database. This was crucial for caching login information, as having to query the database every time a user was accessing a user-restricted file ended up taking too long. In the final implementation. Using redis greatly reduced the amount of time it took to serve user-restricted files.

Finally, we added a Parse database, which allowed for the storage of key-value pairs. We had to keep the iOS apps that are in a group order synchronized, and we did so by using push notifications. These push notifications managed by the Parse database (covered more in depth in section 4.4.2).

4.2 Technologies Used and Design Rationale

We used a variety of technologies to make the system fully functional.

4.2.1 iOS App

- **Objective-C:** There are two languages that can be used to program for iOS – Objective-C and Swift. At the time of our design phase, Swift was very new and there was not a lot of documentation, and we all had prior experience with Objective-C, so it was more natural for us to design with Objective-C.

- **Stripe:** Stripe is the library we used for handling credit cards. There are several different platforms for mobile credit card payments, but we chose Stripe because of the low transaction rate (approximately 2.9% per transaction). Furthermore, Stripe offers several benefits over other similar solutions, such as the ability to refund entire purchases (including the transaction fee) for free and support for Apple Pay if we

decide to add it in the future.

- **CocoaPods:** CocoaPods is a dependency manager for iOS projects. Instead of installing third-party libraries (such as the Parse library for in-app purchases) manually, CocoaPods allows for us to install dependencies and keep them up to date with ease.

4.2.2 Web Application

- **HTML, CSS, and JavaScript:** These are the basic technologies necessary to make a web application. HTML provides the layout of the page, CSS provides the styling, and JavaScript provides the functionality.

- **EJS Templates:** EJS is a template engine for node.js, which is what we programmed the back-end of our system in. EJS templates allow us to modify web templates before they even reach the user, meaning we can customize web pages based on which user is logged in.

- **AngularJS:** AngularJS is a front-end JavaScript framework, and allows for data binding between the view and the model. When the view is changed, the model is instantly updated. AngularJS allowed us to make a single-page web app.

- **BootStrap:** We used this framework mainly for scaffolding, which defines how objects on a page are laid out and respond to resizing. This allowed us to focus on more important aspects of design.

4.2.3 Server

- **DigitalOcean:** We decided to host with DigitalOcean because all of their servers use solid state drives and offer hosting in San Francisco, meaning that the response time for web requests is very quick. In addition, they provide a lot of documentation on how to set up and configure different web servers and provide very low rates (around $5 for a small development accound).

- **Nginx:** Nginx was the HTTP server we decided to use because of its stability and amount of resources on the product. With Nginx, not only could we make our web application open to the public, but we could add lower-level functionality to our website, such as sitewide HTTPS.

4.2.4 Databases

- **MongoDB:** MongoDB is one of the most popular NoSQL languages available. It's support comes from its ease of use and the amount of documentation out on the language. Whereas traditional SQL stores values as key-value pairs, MongoDB stores data as JavaScript objects. Although we sacrifice speed using MongoDB versus a traditional SQL-like language, we gain flexibility over what we can store in and retrieve from the database. This was important, because we stored complicated data structures (such as entire menus) that would have been difficult to store in a traditional database. Furthermore, because we were using JavaScript for our back-end language (node.js), storage and retrieval of information with MongoDB requires no conversion.

- **Parse:** Parse is a very popular key-value NoSQL database system. We used Parse to handle syncronization between iOS devices with silent push notifications. Parse was useful for us because we did not have to worry about difficult concepts such as scalability when we were using it mainly for one purpose.

4.3 Use Cases

Use cases describe the different ways that users can interract with a system. Below are the different use cases for the iOS app and the web application. As stated previously, the iOS app is designed to be interracted with by restaurant cutomers, and the web application is designed to be interracted with by restaurant staff members.

4.3.1 iOS App

Actor Restaurant Customer

Goal View a restaurants menu

Pre-Condition Actor is viewing a restaurants landing page.

Post-Condition Actor is viewing the restaurants menu.

Scenario

1. Start app to get to the restaurant listing page.
2. Select the desired restaurant from the restaurant listing page.

Exceptions

- Phone is not connected to the server.

Actor Restaurant Customer

Goal Enter or change table code

Pre-Condition Actor is viewing a restaurant's landing page and has table identification code.

Post-Condition Actor will join an ordering group and be moved to the restaurant's menu.

Scenario

1. Select "Enter table code" button.
2. Type in table identification code.
3. Press continue.

Exceptions

- Table code is entered incorrectly.

Actor Restaurant Customer

Goal Add food item to cart

Pre-Condition Actor is viewing the menu and has joined an ordering group.

Post-Condition Actor has added a food item to the cart.

Scenario

1. Select plus button next to the desired food item.
2. Select the minus button if the food item is no longer desired.

Exceptions

- App crashes or phone dies.

Actor Restaurant Customer

Goal Pay for order

Pre-Condition Actor has ordered food and the restaurant has delivered the food to the table.

Post-Condition Actor has paid for the order.

Scenario

1. Select split button if you want to split the food with somebody else.
2. Select users from the alert box to share the cost of the food item.
3. Verify order is correct.
4. Press OK.
5. Select “Pay” button.

Exceptions

- Food items were erroneously added to the cart or a user splits with the wrong person.

Actor Restaurant Customer

Goal Leave rating

Pre-Condition Actor has just paid for the order and has the app still open.

Post-Condition Actor has rated their order.

Scenario

1. Actor may select star level (1-5 stars) to indicate quality of food.
2. If desired, no rating needs to be left for some or all of the food items.
3. Select “Submit” button.

Exceptions

- User ordered a food item that they did not eat.

4.3.2 Web Application

Actor Restaurant Staff

Goal Log in

Pre-Condition Actor has an OmniSplit restaurant account.

Post-Condition Actor has logged in.

Scenario

1. Navigate to the OmniSplit login page (www.omnisplit.com).
2. Enter registered email and password.
3. Press "Log In".

Exceptions

- Incorrect username or password entered.

Actor Restaurant Staff

Goal View pending/previous orders

Pre-Condition Actor is logged in.

Post-Condition Actor is viewing sortable list of pending and previous orders.

Scenario

1. On the menu bar, select the "Orders" tab.

Exceptions

- User waits too long and get logged out by the server.

Actor Restaurant Staff

Goal Mark pending orders as completed

Pre-Condition Actor is logged in and viewing list of pending/completed orders.

Post-Condition Actor has marked an order as completed.

Scenario

1. Locate the desired order under the pending orders column.
2. Either click the button below the column or drag it to the “completed” column to mark it as complete.

Exceptions

- User waits too long and gets logged out by the server.

Actor Restaurant Staff

Goal Edit menu

Pre-Condition Actor is logged in.

Post-Condition Actor has edited the menu.

Scenario

1. From the top, select the “Menu” tab.
2. Press the “Add new category” button to add a new section to the menu.
3. Click on a category on the iOS render and press “Add new food item to this category” to add a new food item to the category.

Exceptions

- User waits too long and gets logged out by the server.
- User enters duplicate information.

Actor Restaurant Staff

Goal Edit web app settings

Pre-Condition Actor is logged in.

Post-Condition Actor has edited their account settings.

Scenario

1. From the top, select the "Settings" tab.
2. From this screen, enter new information (if necessary) to the desired settings fields.
3. Press the "Submit" button for the section that you edited to save the information.
4. If not satisfied, press the "Reset" button to revert settings back to their previous saved state.

Exceptions

- User waits too long and gets logged out by the server.
- User enters an invalid string that doesn't render on an iOS device.

4.4 Databases

As stated in previously, we have two databases – a Parse database and an iOS database. We will cover how both of these were implemented.

4.4.1 MongoDB

We said before that MongoDB was used as our primary datastore. We have 3 collections, or different kinds of information stored in our database. These include menus, restaurants, and users. On our web server, we defined how the collections are to be defined with models. The code snippets in listings 4.1-4.3 show the implementation for our models in node.js.

```
1   var User = new Schema({
2    email: { type: String, required: true, unique: true },
3    password: { type: String, required: true },
4    restaurant: { type: Schema.Types.ObjectId, ref: 'Restaurant' }
5  });
```

Listing 4.1: User Model – /models/user.js

```
1  var Restaurant = new Schema({
2    name: { type: String, required: true },
3    backgroundImage: String,
4    theme: String,
5    description: { type: String, default: '' },
6    menu: { type: Schema.Types.ObjectId, ref: 'Menu' },
7    address: {
8      addressLine1: { type: String, default: '500 El Camino Real' },
9      addressLine2: { type: String, default: '' },
10     city: { type: String, default: 'Santa Clara' },
11     state: { type: String, default: 'CA' },
12     zip: { type: Number, default: 95050 },
13     _id: false
14   },
```

```
15 });
```

Listing 4.2: Restaurant Model – /models/restaurant.js

```
1  var Menu = new Schema({
2    group: [{
3      name: { type: String, required: true },
4      description: { type: String, default: "" },
5      _id: false,
6      item: [{
7        name: { type: String, required: true },
8        description: { type: String, default: "" },
9        image: String,
10       price: { type: Number, required: true },
11       _id: false,
12       step: [{
13         text: { type: String, required: true },
14         required: { type: Boolean, required: true },
15         maxOptions: { type: Number, required: true },
16         _id: false,
17         option: [{
18           text: String,
19           priceModifier: { type: Number, default: 0 },
20           _id: false,
21         }]
22       }]
23     }]
24   }]
25 });
```

Listing 4.3: Menu Model – /models/menu.js

When a restaurant account is created, 3 entries are created under the same id – one for each collection. This way, it is easy to reference a restaurant from a user's id, and a menu from a restaurant's id. In listing 4.1, we can see that a user consists of an email, password, and a reference to a restaurant. In listing 4.2, we can see that the model contains all essential information about one's restaurant, including name, description, address, and a reference to a menu. In listing 4.3, we can see that a menu consists of an array of groups. Each group defines a section of the menu, such as appetizers or drinks. Each group has an array of items, which describe each food item on the menu. All of the tables have the ability to store data that is not currently supported in the UI, such as images for menu items.

4.4.2 Parse

+ Row | - Row | + Col | Security | More ▾

objectId String	channel String	userId String	createdAt Date	currentOrder Array	submitted Boolean	updatedAt Date
1Soq136zNL	a5535edc0bb8…	b1800617230	Jun 08, 2015,…	[{"count":1,"name":"French Fries"}]	false	Jun 08, 2015, 02:41

Figure 4.2: A look at the Parse database in action. Pictured is an iOS user who currently has an order of french fries in their cart.

It was essential that we dedicated a large amount of time working on data syncronization for the iOS apps. In order for users to be able to interract with other members of their

ordering group in real time, we needed to have some form of syncronization between users in the ordering group. Without syncronization, a major component of our original design would be missing.

To manage syncronization, we used silent push notifications. These behave like normal push notifications, but the user is not notified of their arrival. Rather, they are silently handled by the iOS app. In our iOS app, whenever a user in an ordering group would perform an action (i.e. adding/removing items to and from an order), an entry for the user in our Parse database would be updated. Figure 4.2 shows an example entry in the Parse database. An entry in the database includes data such as the user's ID, the group they're in, and their current order.

Once a user updates their entry in the Parse database, a silent push notification is sent out to all other members in the group. This in turn signals all group members to update their app.

4.5 Security

One major concern that we had was how we were going to handle security in OmniSplit. We took several measures to secure the site. First, all passwords in our databased are salted and hashed, meaning that even if the database's security is compromised, it would take a prohibitively long time to crack the passwords. Furthermore, there is site-wide HTTPS, meaning all connections between the user and our website are encrypted. This prevents anyone from snooping on network traffic and stealing user logins. Finally, once logged in, a secure session cookie will be stored on a user's browser. This cookie is required to access secure pages, such as the menu customization screen.

4.6 Development Timeline

In order to manage our time properly and meet our deadlines, we created a Gantt Chart for each quarter of development. These charts show deliverables and their deadlines, as well as who is assigned to what task throughout the development process. Figure 4.3 shows our Gantt Chart for the fall quarter and winter break, fig. 4.4 shows our Gantt Chart for the winter quarter and spring break, and fig. 4.5 shows our Gantt Chart for the spring quarter. We had planned to do the majority of research, planning, and prototyping fall quarter, the majority of design winter quarter, and the majority of write-ups and testing

spring quarter. We stuck closely to the development timeline for fall and winter quarter, but continued development through spring quarter as we had not finished all of the functional requirements at the end of winter quarter.

4.7 Testing

Throughout the course of our project, we did continuous testing in order to make sure the product functioned as planned. We planned to use both alpha testing and beta testing but we ended up focusing more on alpha testing.

4.7.1 Alpha Testing

Alpha testing occurred throughout the development process. As we developed the application and website, we had to make sure that the two platforms were synchronized and that everything worked as expected. Any bug reports or feature requests made during this process were logged using GitHub and assigned a severity. By doing this, we kept track of what still had to be done to complete the system in addition to being able to prioritize tasks. It also was helpful when multiple people were working on the same platform but different features: one person could report a bug in the others section so that person could replicate it and fix it themselves.

4.7.2 Beta Testing

We had planned to have a beta phase of testing once implementation of the product was done, but we did not finish the system in time for that. Our plan was to have a local restaurant sign up to use OmniSplit and distribute the application to a few of our friends, then have them order and eat at the restaurant using our product. This would have allowed us to discover how users like our product in a real restaurant environment, how stable our product is, and how user friendly it is. We also would have been able to see if there are any important features that are missing from our current implementation.

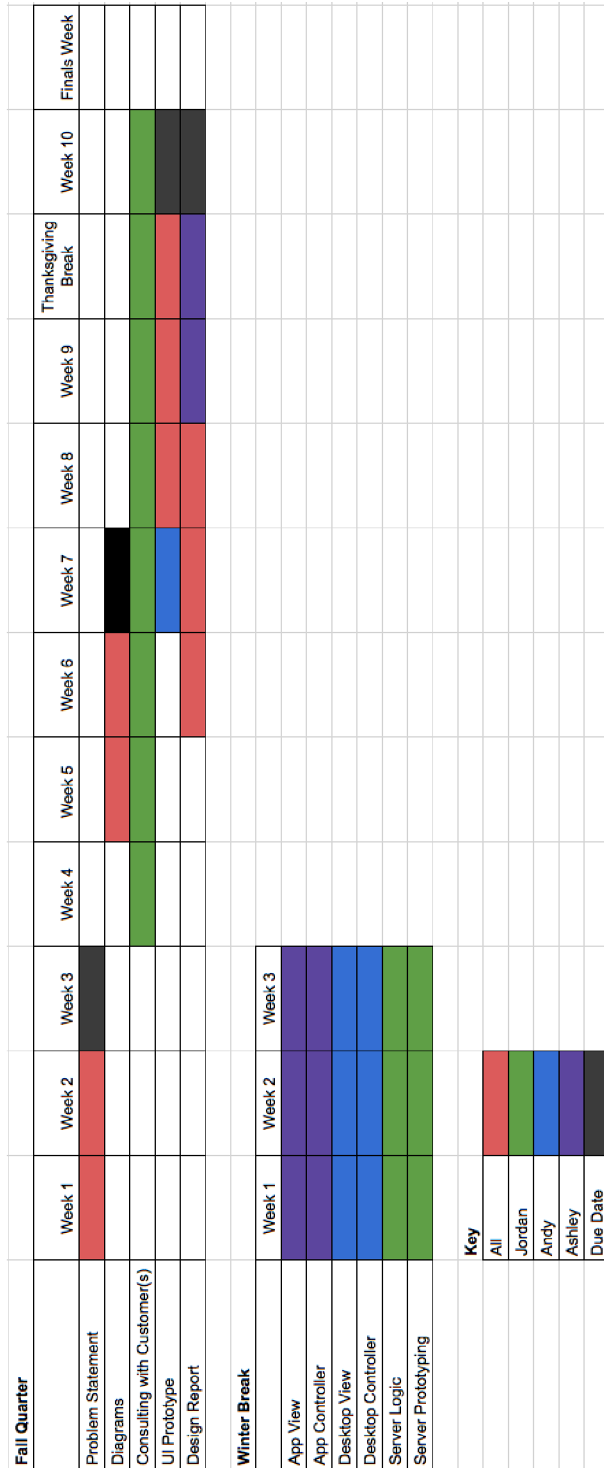


Figure 4.3: Gantt chart for Fall Quarter 2014. Development during this time mainly includes research, planning, and prototyping.

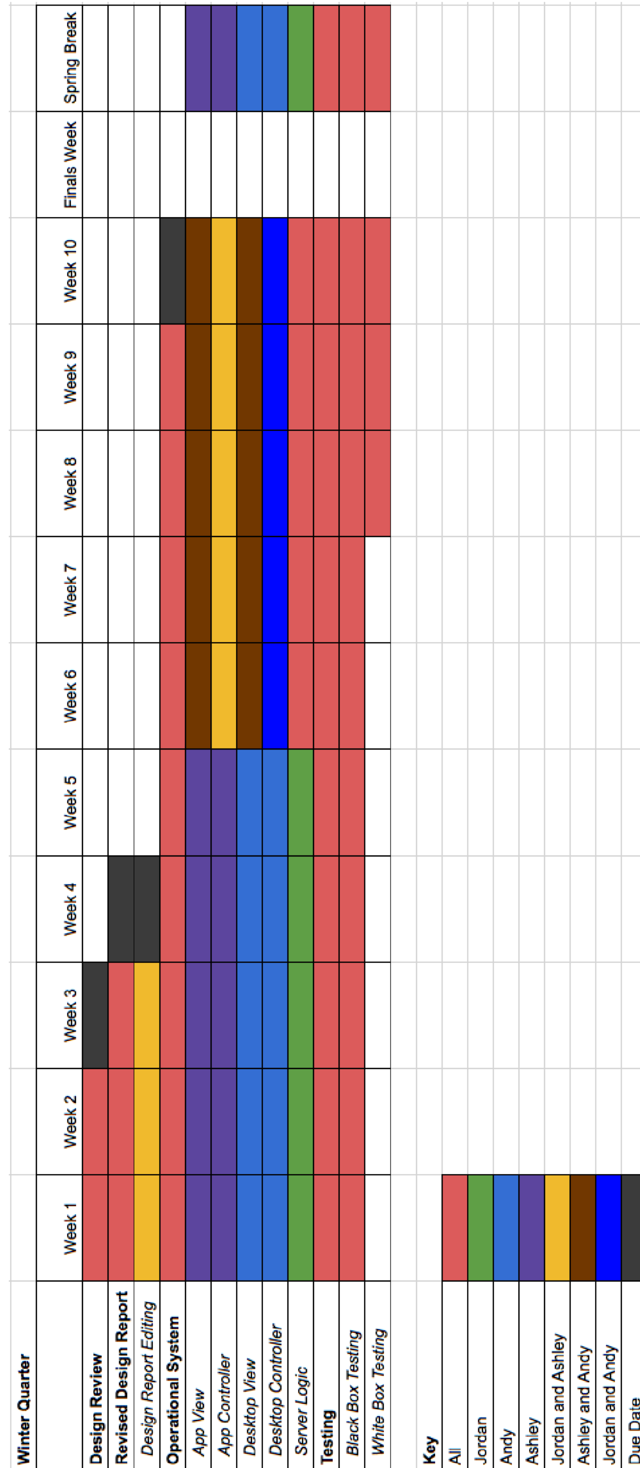


Figure 4.4: Gantt chart for Winter Quarter 2015. Most of the progress during this quarter is made on design and testing.

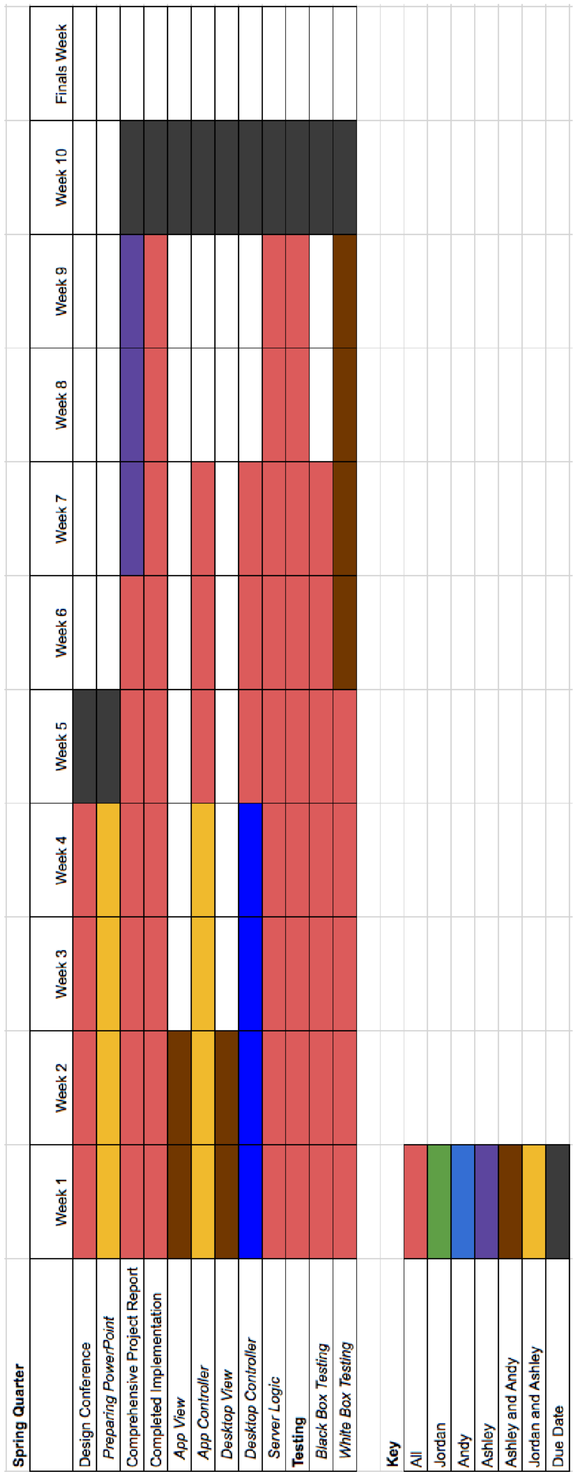


Figure 4.5: Gantt chart for Spring Quarter 2015. The focus of this quarter is on testing and preparing for the Senior Design Conference.

Chapter 5

Societal Issues

OmniSplit was not designed in an ethical vaccuum. Our actions as developers change the world around us, for better or for worse. We tried to develop OmniSplit with societal implications in mind so that we could do the greatest amount of good for the greatest number of people.

5.1 Ethical

As users trusted our system with their credit card information, it was critical that we developed an ethical product. Credit card information is handled by a secure third party, and it is critical that we do not sell user data to outside companies or knowingly leave the system vulnerable to cyber attacks.

5.2 Social

Apps need to be fun to use, or else nobody would end up using them. OmniSplit brings fun, social interraction into an environment that is typically void of fun. Splitting a check the old-fashioned way can be a daunting and time consuming task, and mobile technology can help to remove these issues. With OmniSplit, one can easily and quickly split the check so that they can move on to the actual fun in the dining experience: eating.

5.3 Political

Our project is not political or public. However, products that eliminate jobs end up being the subject of many politically charged debates. As such, we need to ensure that we appeal to waitstaff rather than trying to compete with their service. OmniSplit was developed

to complement waitstaff rather than displace it, and additional features such as ability to request waitstaff and add a tip would help support the current waitstaff.

5.4 Economic

In choosing to develop for iOS rather than Android, we incurred the additional cost of iOS developer licenses. Furthermore, the group member who focused the most on the iOS application was the only member who did not own an iPhone, so we also used our funding to purchase a device to develop on. Had we split up the work differently, we would not have had this cost. However, we attempted to cut costs in other areas such as choosing the most cost effective server hosting company.

5.5 Health and Safety

Our project does not affect our users' health or safety beyond privacy and security - we must make sure that our users' personal information is secure. As we discussed in section 4.5, security is one of our largest concerns, as a system that is insecure and prone to being hacked will greatly hurt our chances at gaining a large user base. As such, we tried to incorporate a variety of security measures into our implementation and practiced good development habits, such as not reusing passwords or posting files that could compromise the project's security online.

5.6 Manufacturability

We designed OmniSplit with some level of expandability in mind. We can always purchace more powerful or more servers to increase our web platform's power. However, some more testing and design would need to be done to allow us to scale this way. Furthermore, database space can always be purchaced to make retrievals faster or give us more space. Our largest manufacturing concern would be having a full-time development team to keep the iOS app up to date. As changes to an iOS app need to be approved by Apple, we need to have quick responses to bugs to ensure that our bug fixes get launched reasonably quickly. The best way to do this is to have several developers, which would take a significant amount of cost in respect to our current budget.

5.7 Sustainability

There is no limit to the lifetime of our product. It can easily be updated to accommodate new needs not originally planned for. Instead of trying to solve one issue, we sought out to develop a platform that could be modified in the future.

5.8 Environmental Impact

Although we did not focus on fixing the environment, our system does help to reduce paper use. OmniSplit is designed to be paper free, so all reciepts are digital.

5.9 Usability

We made every effort to make our product as user-friendly and intuitive as possible. We chose a readable font, placed buttons where they would be easily noticed to keep the user moving forward in the process of ordering and paying for their meal, used titles or headings that would make the purpose of different sections or pages clear to a user, and put features where users would expect them. The end user should have no trouble picking up our product and using it without any instruction.

5.10 Lifelong Learning

In designing and developing our product, we all had to learn new technologies or new ways to use the technologies we already knew to accomplish different tasks. We had to seek out this information for ourselves rather than being given it in our courses.

5.11 Compassion

Waitstaff at restaurants are typically underpaid and overworked. They are expected to make most of their income from tips, but that is rarely a livable wage. Many people do not tip their servers enough because the service was slow, but this is usually due to servers being overworked and restaurants being understaffed. By supplementing waitstaff but not replacing them, our hope is that our product makes service faster and reduces the burden on waitstaff without decreasing the number of servers employed at a restaurant or reducing their tips. A future feature that we would have liked to have implemented in our product would show the server's picture and a caption or fun fact about them on the tip page. This would encourage people to tip their waitstaff better.

Chapter 6

Conclusion

6.1 Report Summary

In chapter 3, we described our vision for the implementation of OmniSplit. Chapter 4 elaborates on how we actually carried this vision out. In the end, we designed two platforms, a web application and a mobile iOS application, that will allow restaurants and customers to better enjoy the dine-in experience. The web application, exclusively for restaurant owners and staff, had most of its core components implemented, including managing their account, customizing their menu, viewing pending and completed orders, and marking orders as completed. The mobile application, for restaurant customers, allows a user to browse restaurants menus, join ordering groups, pay for all or part of a bill, and leave a rating. The decision to implement our design with a web-based application came from the advantages of current web tools. The technologies we used helped to create a clean design and structure of our system. We worked to integrate all the components to create an environment that we believed to be very user friendly.

Even though the components were working sufficiently for a demo, we had to cut several features out from both components. Restaurant customers do not have accounts, and the ratings they leave do not get submitted to the server. We had to cut out analytics completely from our implementation, as we ended up focusing the remaining weeks of our development on existing modules rather than adding an entirely new (but incomplete) one.

6.2 Lessons Learned

OmniSplit was a large and comprehensive project, and we learned a lot about programming and development along the way.

6.2.1 Planning

First and foremost, we discovered that any large projects requires in-depth planning in order to stay on track. Although we mostly stayed on track, we spent a good amount of time re-thinking and re-programming completed modules. By setting stricter deadlines for ourselves and putting more effort into the early planning process (market research, mock-ups, etc.), the development of our project could have been much more streamlined.

6.2.2 Scope Creep

Scope creep is defined as planning to develop too many features that eventually have to be cut so that the deadline can be made. This became a major issue later on in our development process. As our deadline was approaching, we realized that we would not be able to adequately implement some initially planned features. In the end, we decided to dedicate the majority of our efforts towards making sure core functionality (i.e. splitting the check, editing the menu) worked before adding any features that were not essential to the demo (i.e. leaving ratings, marking orders as completed).

6.2.3 Use of Github

As we described in section 4.7.1, we not only used Github as a site to store our code repository, but as a tool for bug tracking. By listing bugs and prioritizing them, we could easily divy up the work and not waste time fixing minor bugs when there are major issues to fix. Some minor issues came up when we used GitHub, such as conflicts when we tried to edit the same file at the same time, but without GitHub we would have had wasted much more time than we did on keeping track of bugs and keeping the code repository up to date.

6.3 Moving Forward

Even though OmniSplit is currently functional, we discussed in section 6.2.2 that we did not get to include everything that we wanted into our final system. In order to take this product to marker, we have identified a number of features that we would need to implement.

6.3.1 Analytics

The most important feature that we had to cut from our final design is is the ability to view analytics about a restaurant. OmniSplit is a comprehensive platform for many different restaurants, and with a large enough userbase, it would be very easy to give a restaurant

key statistics about their customers, including their favorite and least favorite food items, how much they tend to spend, and how many return to eat at the restaurant a second time. Implementing this would allow a restaurant to grow and thrive without relying on gut-feeling management.

6.3.2 Tipping and Refunding Payments

Even though you can make payments to the restaurant, you cannot select a tip for your waiter. This is essential for any platform that wants to complement the current waitstaff. Additionally, the ability to refund payments is necessary in the event that a customer ordered the wrong thing.

6.3.3 QR Codes

One minor thing that we discussed but never got to look into was the ability to integrate QR codes into the iOS app. Instead of entering a table code number, one could just scan a QR code placed at the table to join an ordering group. Not only would this make the app more user friendly, but it would prevent people from being able to order food when they are not in the restaurant.

6.3.4 Geofencing

Geofencing is the ability to draw virtual boundaries around areas by using GPS or other positioning technology. Adding this minor feature would greatly extend our system's functionality. Not only could listed restaurants be narrowed down by location, but the app could know when a user is near an OmniSplit restaurant and an icon to access the app could show up on the lock screen.

6.3.5 Search Functionality

Although not in our functional requirements, it would be very useful to be able to search for restaurants. Additionally, with geofencing and analytics implemented, it would not be too hard for our system to recommend top restaurants a user's area based on his or her eating habits and current location.

Chapter 7

Appendix

7.1 Additional Diagrams

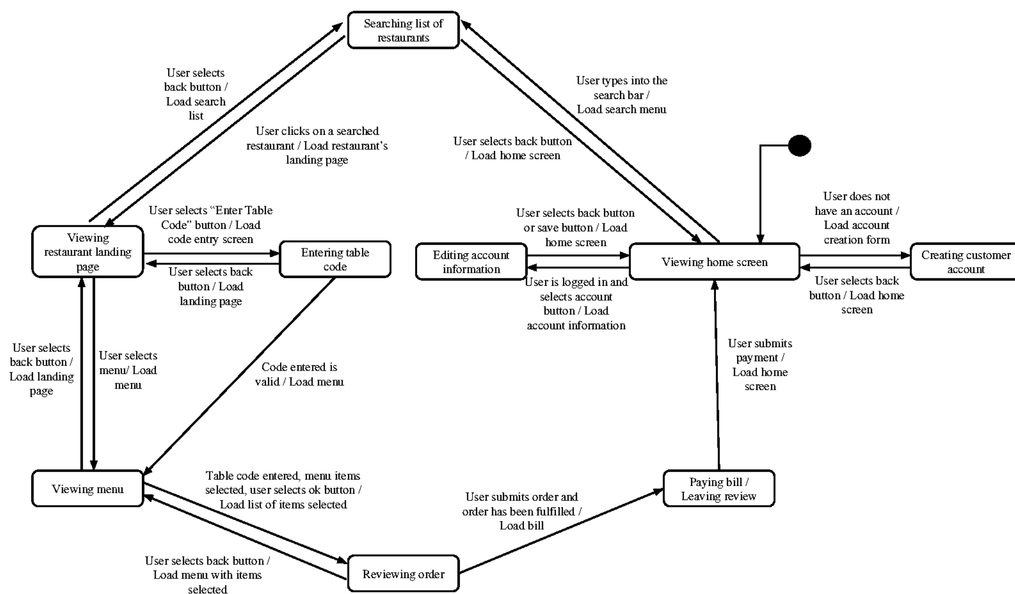


Figure 7.1: Original state diagram for the iOS app. Features not incorporated into the final design include creating/managing an account and searching for restaurants.

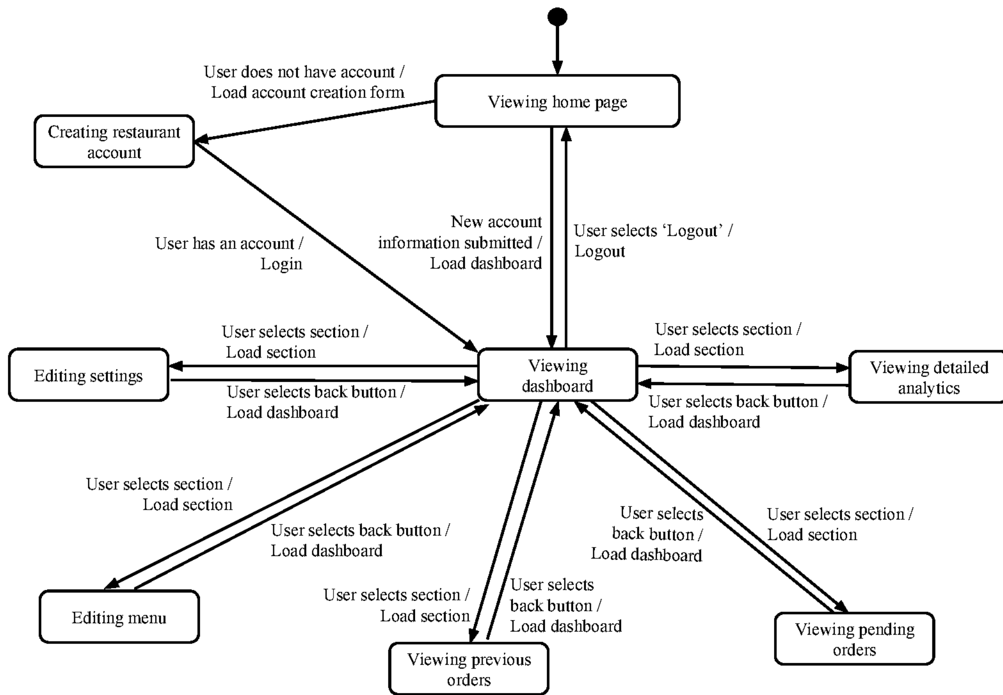


Figure 7.2: Original state diagram for the web app. Features not incorporated into the final design include viewing analytics.