Project Report

Team 233

Hongnian Yu, Dong Liang, Tianlei Sun, Jian Zhu
California Institute of Technology
Department of Electrical Engineering

1 Team Member & Work Split

- **Group members:** Hongnian Yu, Dong Liang, Tianlei Sun, Jian Zhu
- **Team name:** 233

2 Market Structure

2.1 Introduction

There are already tons of food delivery apps/websites in the current market. However, most of the companys either charge too much or have a minimum order amount for their delivery services. To overcome this problem, we propose the concept of community based delivery service with the hope of providing an alternative for the current food delivery market. Literally, community based delivery means the food will be delivered to certain cluster/group of customers residing in the same community. The community has a broad meaning in the scenario – it could refer to a certian campus, campus, etc. For example, a Caltech student who is out for dinner can bring food to other student in Caltech. The basic asuumption is that student from a community will eventually go back to the same community. In this way, we can dramastically cut down the delivery labor cost.

2.2 Current Competitors

The market of food delivery service has existed for decades. There are lots of successful companys such as UberEat, Grubhub, Postmates, etc. The major difference between us and our competitors is that they are not community based. The community based method is likely to greatly reduce the delivery fee, but it may not be able to provide as many choices as the regular method and the restaurant with no community members inside will be unavailable to the customer. Thus, as mentioned in the previous section, we are not aiming to redefine the current market; instead, we simply want to provide an alternative solution.

3 App Development

3.1 Architecture

The general architecture of our app is shown in Figure[1]. There are two types of users in this app – driver and customer. Driver can make the post to indicate which restaurant he can deliver. Then this information is posted to our database and the corresponding restaurant will be shown in the main page. Customer can then make an order follow a common procedure. The order/delivery status can be found in the history section, which contains information for both the current order/delivery and previous order/delivery.

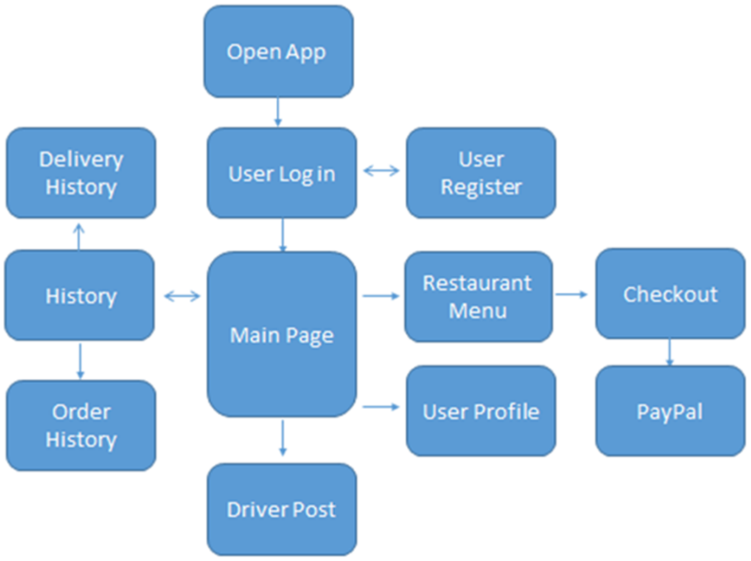

Figure 1: Me2 Architecture

3.2 Android Techniques

- **RecyclerView**
  RecyclerView widget in android studio is a more advanced and flexible verision of regular ListView. Similar to ListView, RecyclerView is used to store and display a list of objects in the screen, but it is more in term of memory usage and speed. For instance, ListView will encounter severe performance degradation when the list contains thousands of object. On the one hand,the app would end up using a lot of memory and storage, potentially making the app slow and crash-prone. On the other hand, the app created UI widgets each time a new object scrolled onto the screen and destroyed the widgets when it scrolled off, that would also cause the app to run slowly[1]. RecyclerView avoid this problem by using a dynamic view structure that can reuses cells while scrolling up/down and decouple the object list from its container. In our app design, we use RecylcerView to display the available restaurants and order history, which could potentially be a long list.

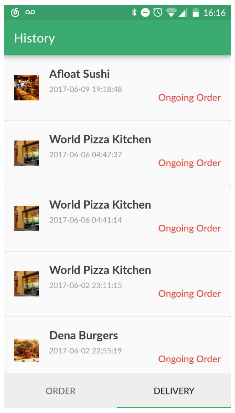

Figure 2: A screen shot of RecyclerView usage in our app

- **AutoCompleteTextView**
  One critical technique we used is AutoComplete Text. One common issue in searching happens when customer type the name of the restaurant to the search bar. Because customer may be clearly remember the name of the restaurant, typos often occur. Therefore, our solution to this kind of issue is AutoComplete Text. As shown in the Figure[3], when user type the name of a restaurant, AutoComplete Text package will automatically correct the name of the restaurant. The corrected name input can be used as a key to find the corresponding restaurant ID in the backend.

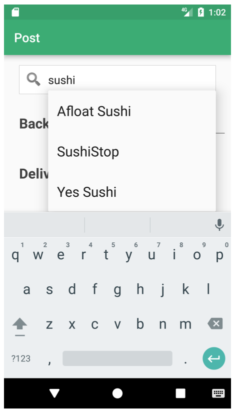

Figure 3: AutoComplete Text in Restaurant Searching

First, based on the location of user, me2 will load all the local restaurant names and corresponding IDs. After the Android App loaded these information, a HashMap is created to store the information. AutoComplete Text package can ensure that the name entered is correct. With the correct name, me2 can simply match the corresponding ID with the name as the key. Then, the ID received from HashMap, along with other information user inputted, are passed to the next activity and uploaded to the backend.

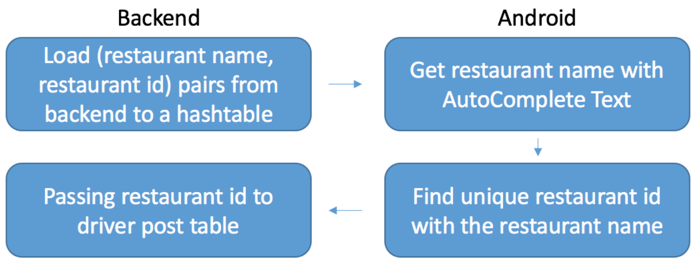
Backend
Android
Load (restaurant name, restaurant id) pairs from backend to a hashtable
Get restaurant name with AutoComplete Text
Find unique restaurant id with the restaurant name
Passing restaurant id to driver post table

Figure 4: Hashing Restaurant ID with Restaurant Name

However, one potential problem happens when the restaurant name is not unique. For example, there are many In-N-Out restaurants in Pasadena. In this case, the restaurant name might not be unique. Our solution is to use ”Resturant Name” + ”Restaurant Street” as the key of the HashMap. By doing so, AutoComplete Text will provide details about the location of restaurants for users to choose. And the ”new” restaurant name is still unique to the HashMap.

- **PayPal Android SDK**
  We adopted PayPal as our payment method after a customer confirmed his order. PayPal has provided Android SDK that makes it easier to accept app payments. Before that we have created a *client_id* required for the test environment. Also we set up a personal Sandbox account email and password to test the Android integration against the PayPal Sandbox. When customer clicks the ”pay with PayPal” button in the checkout page, he will be directed to the SDK’s UI, with total cost of the order also listed on the first page, as shown in Figure[5]. Meanwhile, the PayPalService class embedded in the SDK will be called so that it gets ready for PayPal server communication and service authentication. Then we can use our Sandbox account email to log in to our testing PayPal account and details of the account like balance will be shown. After clicking the ”Pay” button, PaymentActivity embedded in the SDK will be activated for initiating the transaction and returning the payment result. If the payment is successful, customer will be directed back to the restaurant menu page.

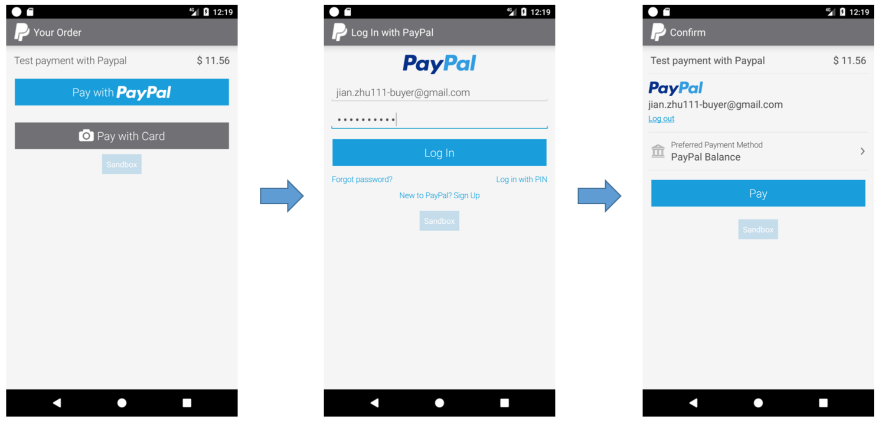

Figure 5: PayPal Android SDK Integration

Currently we do not support payment with credit card yet, and only single payment is used here within PayPal, which means each time customer makes an order, he has to input his account email and password. This could be troublesome and no good for user experience. However, PayPal SDK has a second use case, allowing user to log in to PayPal just one time and consent to future payments. Therefore we may consider adding this feature to make the order process more convenient.

- **Google Distance Matrix API**
  Another feature we have developed is distance measurement, because customers may choose a restaurant based on how far it is, and how long it takes the driver to bring food back. Thus we used the Google Distance Matrix API which provides travel distance and time based on the recommended route between start and end points. To use this API, we first have to get an API key for identification and authentication. Then we can call the Google distance service by entering the origin and destination location.

  However, in the final version of our application, we did not put this feature into use because we do not know when the driver will finish eating at the restaurant, simply using this distance estimate becomes somehow insufficient. Moreover, we have already asked driver to input his return time when making a post, so we could just use this for customer's choice instead.

3.3 Back-end

The back-end is the machine that runs a site. The user doesnt see it or directly interact with it as with client-side technology, but its always running in the background, delivering smooth functionality, a desktop-like experience, and information from the database right into the application. The back-end of our application handles requests from the users and returns information from the database, such as what restaurants are available, what food you can order, who will be assigned to deliver the food, etc. It plays an important role in making the application functional.

3.3.1 Back-end Structure

In order to make it work, we built our back-end based on Apache2 web server and MySQL database. Apache is the most commonly used Web server on Linux systems, while MySQL is one of the most popular relational database management system. We instantiate the Web server on Ubuntu 14.04 system within Amazon EC2, and instantiate the database on Amazon RDS. The structure of our application's back-end is the same as depicted below.

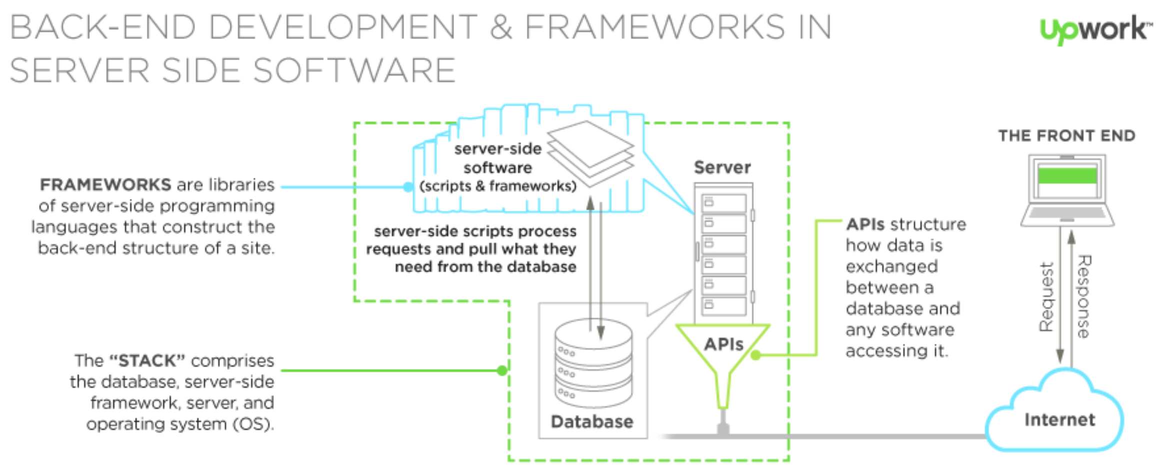
BACK-END DEVELOPMENT & FRAMEWORKS IN SERVER SIDE SOFTWARE
upwork
FRAMEWORKS are libraries of server-side programming languages that construct the back-end structure of a site.
server-side software (scripts & frameworks)
server-side scripts process requests and pull what they need from the database
Server
APIs structure how data is exchanged between a database and any software accessing it.
THE FRONT END
Request
Response
The "STACK" comprises the database, server-side framework, server, and operating system (OS).
Database
APIs
Internet

Figure 6: Back-end Structure

3.3.2 Database implementation

Since we used relational database to store all the information, we applied the knowledge learned in CS 121 to design the structure of all the tables. We have created six tables to store all the data, which are *user, restaurant, food, post, orders, orderDetail*.

- *user* table stores all the credentials of registered users, with password hashed.
- *restaurant* table stores all the info of the restaurants, which is crawled from Yelp.
- *food* table stores all the food provided by the restaurants with food name, price and category.
- *post* table stores the information of all posts by the driver, including the place driver will return to, post time, restaurant id, driver id and some other info.
- *orders* table stores all the orders with order time, delivery address, driver and customer id, etc.
- *orderDetail* table includes the food name, quantity and price for all the orders.

3.3.3 Server-side Software

The server-side software is all written in PHP. We have written 10 PHP files handling the POST request for registering new accounts, logging in to our application, getting available restaurants, getting order history and delivery history, etc. These PHP files will return the requested information to our front-end and make sure it works as desired.

3.3.4 Push Notification

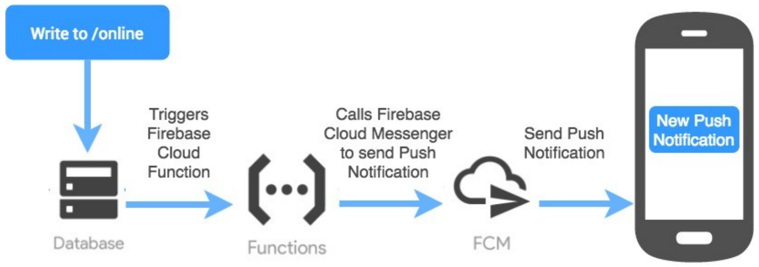
Write to /online
Triggers Firebase Cloud Function
Calls Firebase Cloud Messenger to send Push Notification
Send Push Notification
New Push Notification
Database
Functions
FCM

Figure 7: Push Notification Algorithm

Another feature that we implemented is push notification to drivers when there is a new order. The implementation involves the usage of Firebase messaging service. As indicated in the figure above, when there is an order being placed, the database will be modified and this will trigger Firebase Cloud function in our PHP file. Our server will send a request to Firebase Cloud Server with the details of the message and our notification receiver (the driver who receives the order). Firebase Server will then send a push notification according to the info provided.

3.4 UI & Logo

We named our app as ”me2”, which is the abbreviation for ”me too” and comes from ”Can you bring that for me too?”. This often happens when some of our friends go out for food and we want them to take something back. Besides, we chose green as theme color for our app because it is not so widely used in app design now, and also has some symbolic meaning for growth and energy. As shown in Figure[8], we added a spoon and fork here to illustrate that we are doing something related to food.

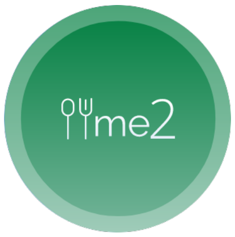

Figure 8: App Logo

For the UI, we also used green as primary color, so the tool bar, status bar, progress bar and most of the buttons have a uniform color. In the early stage, we just created a very simple and coarse version of UI, because we were focusing on developing some of the functions first. After referring to some other apps, we improved some of our UI to make it look better and more refined, like using image buttons in the main page, and adding images in the search bar and text box. Figure[9], Figure[10], and Figure[11] are screen shots taken from our app. We found that sometimes images can be more straightforward and intuitive for understanding.

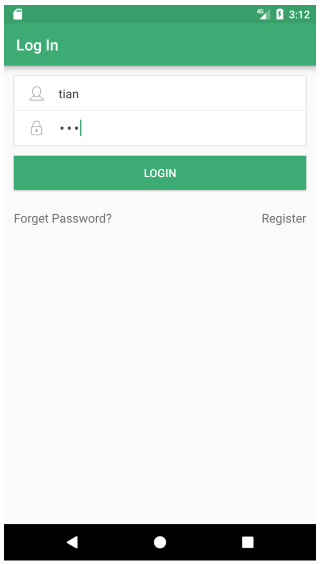

Figure 9: Login Page

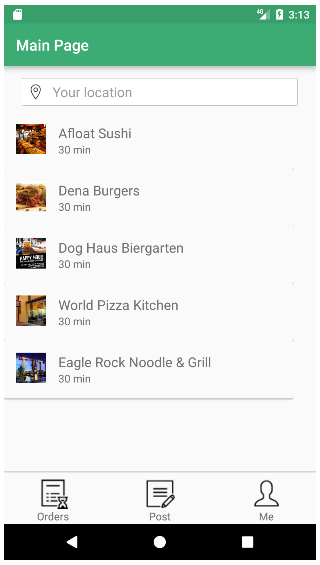

Figure 10: Main Page

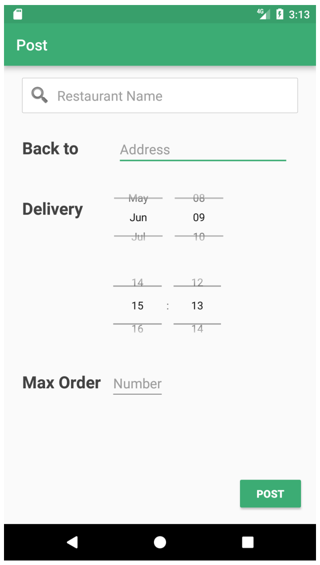

Figure 11: Driver Post Page

4 Lessons Learned

4.1 Ideas

- **Simplicity**
  One big lesson we learned through this development procedure is simplicity design. Different from web or PC, mobile apps emphasize simplicity. Because the screen size is limited, mobile designs usually have larger relative font size. In the beginning, we tried hard to squeeze space for more functions. However, the end results seems to be really hard to use. We finally realized that the key of mobile design is "minus" rather than "plus". Trying to simplify our ideas and reducing the redundant functions improves our product.

- **User Perspective**
  As developers, we often neglect the feelings of customers. In the early stage of development, we spent plenty of time learning technical knowledge and apply them to our design. The procedure was challenging and exciting. However, for several times, we added features which do not fit users' need. For example, we implemented Google Map API that can show locations as a pin in a map. However, in our app, the destinations are the same for the whole community and therefore showing locations

on a map is redundant in our app. Instead of thinking in developer's perspective, we should think as users, reduce redundant features and elaborate on the core features of the app.

4.2 Implementation

- **Development Procedure**
  When we just started our project, we spent much time discussing the procedure of app development. Later we realized that it is important to first have a flow chart and a detailed description of classed needed, database schema, and necessary methods needed for reading and modifying database. Otherwise we cannot implement the logic flow between pages, and interaction between frontend, and backend. With this framework in hand, we are then able to write the different classes and activities, and divide the work among us.

- **Version Control**
  Another lesson we learned is the version control for development. We have found it hard to unify the version among multiple developers. Although Github has provided us ways to version control, it is still somehow not convenient as sometimes the same piece of code could be modified by several people. Thus to incorporate several changes, we need to read line by line and commit changes one by one. So finally we just decided to merge work onto one person's laptop and then upload to Github. It works but in terms of efficiency and future work, we still need to find some better way to combine multiple person's work.

5 Future Improvement and Perspective

As we discussed in previous sections, there are still many improvements that can be made in the future, both technical and non-technical ones. In this section, we will state some of them. Besides, we will talk about our long-term perspective of this application.

- **Separate apps for drivers and customers**
  One potential improvement is to build separate applications for drivers and customers, just as Uber and Lyft do. Our current application is a little too difficult for first-time users to use. It requires them to understand the logic behind our application, especially that of drivers' post. We have discussed the application with several classmates and some of them have suggested that they want a simpler user experience.

  By providing two separate applications to our drivers and customers, each of them will be able to have an easier-to-use application. For instance, the drivers will be able to make a post directly after they log in to the app about where they would like to have a meal and when they will return. They don't have to find a post button to press. This also reduces customers' frustration when they see a button with meanings they don't know.

  The limitation of this is that if a customer would like to delivery in his or her spare time, which is the case we were hoping for at the beginning, he or she has to download two applications instead of one.

- **Redo the application and Focus on community**
  Another option to improve our product is to redo the application entirely and simplify our ordering

and posting process. The main innovation of our application is to take community as our basis of food delivery service. We can just provide an application for users to communicate freely with each other about where to eat, what to eat, who is going, and things like that. If we can help connect users with simpler user interfaces and shorter time, we will be able to have more users using our application.