CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

ParkMe: A Parking Request App in Swift

A thesis submitted in partial fulfillment of the requirements

For the degree of Master of Science in Software Engineering

by

Leor Benari

August 2017

The thesis of Leor Benari is approved by:

____ ______________

Dr. Adam Kaplan Date

____ ______________

Dr. Taehyung Wang Date

____ ______________

Dr. Li Liu, Chair Date

California State University, Northridge

Table of Contents

Signature Page ii
List of Figures iv
List of Tables v
Abstract vi

Chapter 1: Introduction 1
Chapter 2: Related Work 4
2.1 Sharing Economy Model 4
2.2 Development of Technical Solutions 5
2.3 Solutions to the Parking Problem 6
2.4 Technical Stack for Solution 9
Chapter 3: Methodology, Requirements and Architecture 10
3.1 Project Goals and Challenges 10
3.2 Functional and Nonfunctional Requirements 11
3.3 User Interface 13
Chapter 4: Technical Approach 21
4.1 Swift 3.0 21
4.2 Cloud Server - Firebase 21
4.3 Maps – Apple Mapkit 23
4.4 Email Registration & Login with Firebase 25
4.5 Communication between Buyers and Sellers 27
4.6 Payment - Stripe 29
Chapter 5: Design Patterns Utilized 30
5.1 Protocol and Delegate Pattern 30
5.2 Model-View-Controller 31
Chapter 6: Application Testing 32
6.1 Study Details 32
6.2 Study Analysis 41
6.3 Performance Testing with JMeter & Blazemeter 42
Chapter 7: Conclusion and Future Work 47
References 49

List of Figures

Figure: 3.1 Sequence Diagram	13
Figure: 3.2 Login/Signup Page	14
Figure: 3.3 Available Parking Spots	15
Figure: 3.4 Purchase Spot Screen	16
Figure: 3.5 Seller Receives Request	17
Figure: 3.6 Apple Directions	18
Figure: 3.7 Arrival Notifications	19
Figure: 3.8 Activity Diagram	20
Figure: 4.1 Firebase Services	22
Figure: 4.2 Login and Signup Code	25
Figure: 4.3 DB Sell_Requests	27
Figure: 4.4 DB Buy_Requests	27
Figure: 4.5 Function to accept offer	28
Figure: 6.1 Trial #1	34
Figure: 6.2 Trial #1 Buyer and Seller Event Distribution	35
Figure: 6.3 Trial #2	36
Figure: 6.4 Trial #2 Buyer and Seller Event Distribution	37
Figure: 6.5 Trial #3	38
Figure: 6.6 Trial #2 Buyer and Seller Event Distribution	39
Figure: 6.7 Demographics - Gender	40
Figure: 6.8 Demographics - Age	40
Figure: 6.9 Study Comparison	41
Figure: 6.10 BlazeMeter Performance Trial #1	43
Figure: 6.11 BlazeMeter Performance Trial #2	44
Figure: 6.12 JMeter Response Time per Number of Active Threads - Trial #1	45
Figure: 6.13 JMeter Response Time per Number of Active Threads - Trial #2	45

List of Tables

Table: 6.1 Trial #1 34

Table: 6.2 Trial #2 36

Table: 6.3 Trial #3 38

Abstract

ParkMe: A Parking Request App in Swift

By

Leor Benari

Master of Science in Software Engineering

Over the past few years, there have been a proliferation of service technologies that leverage the principle of shared economy, which allows for the ability to buy, sell or rent goods and services from one person to another. Given the difficulty of obtaining parking at CSUN, other universities, as well as many parts of the world, the proposed project idea is a parking application that will serve as a marketplace exchange of parking spots. While there are applications that allow people to obtain parking spots via available parking spots from businesses, such as parking garages or other managed parking areas, this app will demonstrate how the shared economy business model can be applied to an application for finding parking spots. In addition, this thesis will detail specific use cases, such as the use in universities, where this app would provide benefit over the existing parking application model.

This app will allow people to interact with other people, where one person may exchange their parking spot for a price that is set by the other person. The application will feature registration and login systems, as well as the ability to purchase or sell specific parking spots, and to see those spots on a real-time map. The available parking spots and

price will be shown, sorted by the nearest to the buyer. In addition, the application will be backed by a cloud server, capable of providing real-time database services for storing user data, transaction data and other important information.

Chapter 1: Introduction

Parking has often been an issue for many students attending universities, such as CSUN. The campus' own newspaper, The Daily Sundial, has detailed this problem, suggesting that the parking congestion will continue, unless additional parking structures are built [1]. During busy parts of the day, students arriving to class are sometimes required to wait until someone leaves, and gives up their spot. Despite innovative attempts by Associated Students to assist students in finding parking in and around campus, those who drive to CSUN still face challenges finding parking. Larry Isrow, CSUN parking manager, said of the 12,043 parking spaces available, about 15 to 20 percent of those spaces are available at any given time. However, he also detailed how, even though there may be some spaces available, that not all of those spaces will be close to one's class [2]. With this said, there are often busier parts of the day where parking is scarcer, and certain times, such as the beginning of the school week, where parking is an even greater challenge. Parking has also been shown to have been a challenge at other universities [3], as well as other major cities all over the country.

In examining the need of a parking application such as this, it is necessary to consider a few use cases. First, using the use case of a university setting, certain times of the day are typically much more difficult to find parking spots than others. A typical parking application would only inform a user that parking can be found in a certain area (i.e. a parking garage), and would not show parking spots from public parking spots where someone is about to leave. Using this application, users will be able to see available parking spots (both university and street parking) in the area, and will also be able to potentially find a spot closer to campus. In another use case, an individual may be

in a popular area of a city, where the available parking is either not available, or too expensive. It would be possible to purchase a parking spot from someone who is leaving, who has a spot at a desirable public street parking section, which would be less expensive than nearby parking garages, and which could be potentially closer to desirable venues, such as restaurants, shops, etc. It can also be noted many areas that normally have easy parking, such as malls, shopping centers, etc. tend to experience a rush of parking during specific seasonal parts of the year, such as during a holiday season or notable sales [4]. Lastly, if someone is using a parking meter and still has time left, they could then sell the spot to someone else, allotting them the remaining time.

When an individual is ready to leave, they would be able to use this app to let other people with the same app know that they are leaving and where they are located. When they press the “Sell Spot” button, they will be able to input the specific price that they want for the spot. Once the spot is available, other potential buyers will be able to view that spot, along with other available spots, on a list that is sorted by the distance to them. Once they select a parking spot, a message will be sent to the seller, allowing them to either approve or deny the buyer’s purchase. Once approved, the buyer will get directions to the parking spot selected, and the service is complete. It is important to note, however, that if the parking spot is in a school parking lot or by a parking meter, it is still the buyer’s obligation to have a valid parking permit or to pay for the meter. Further, if the spot is on the street, the buyer must still adhere to all street parking restrictions, such as not being able to park longer than a certain time, or that parking is prohibited at certain times. This goal of this application is to provide a platform for customers to get access to parking spots, and does not give them the ability to park anywhere.

Much of the functionality of this application addresses similar use cases to classic ridesharing apps (such as Uber and Lyft), as well as other apps that utilize location-based services. The app was built on native iOS, using the Swift programming language, as well as Firebase, for authorization and backend services. Throughout this thesis, rationale will be provided as to why this specific technological stack was chosen, including an analysis of real-world simulation tests and stress tests that were performed to test the usability and scalability of this application.

Chapter 2: Related Work

This section will discuss the concept of the shared economy model, will discuss its efficacy and will go into applications across various fields. In addition, the previous work done in the field of parking solutions will be discussed, both in studies already done, as well as commercial applications.

2.1 Sharing Economy Model

As of this time, we are seeing a rise in the amount of businesses using the shared economy principle. Shared economy describes a type of business model that builds on the sharing of resources between individuals through peer-to-peer services, allowing customers to access goods when needed. While sharing of goods has often been a practice employed by people closely tied in communities, in recent years, we have seen this practice progress from a community practice, to a profitable business model. This shift has largely been spurred by peer-to-peer networks, consumer-to-consumer (C2C) market platforms, as well as by the underlying technologies that have allowed them to prosper and reach so many people [5]. This platform has demonstrated not only ongoing value to current markets, but also a potential for future growth, with some projections putting the sector's revenue at $335 billion globally by 2025 [6].

This type of business model is used by ecommerce companies, such as eBay, Amazon, etc., as well as companies that offer the exchange of services, such as Uber and AirBnb. Through appropriate technology and infrastructure, these companies act as a medium between other buyers and sellers, helping to facilitate and regulate transactions.

In defining a successful shared platform, Credit Suisse details three key components: Efficiency, trust and value proposition. Efficiency is achieved by enabling

the buyer to be matched to the seller's services that they require, quickly and efficiently. Trust is achieved by a mix of up-front screening, ongoing feedback and external enforcement. Lastly, value proposition deals with creating a product that both buyers and sellers are attracted to, allowing a healthy marketplace to exists [7].

2.2 Development of Technical Solutions

There are many well-known companies in this sector that leverage this business model. Some of the most popular are the ride-hailing companies Uber and Lyft, as well as the lodging booking service Airbnb. Uber and Lyft, also known as ridesharing services, promise to increase reliability and reduce wait times of point-to-point transportation [8]. Airbnb, another example of a company utilizing the shared economy model, allows individuals to advertise, lease or rent short-term lodging to those interested, often at a reduced rate of existing hotels. In addition, other companies have utilized this business model in novel ways, allowing for various other services, such as the delivery of food, renting of cars/bicycles, or even a service that picks up and does a customer's laundry [9].

These technologies have demonstrated proven efficacy in multiple markets. In one study, it was shown that Airbnb had impacted hotel revenue in Austin, causing a decrease by 8-10% [10]. Moreover, there has been an additional effect in hotels being forced to make their rates more competitive. Numerous studies have shown the impact of ridesharing companies on the traditional taxicab marketplace [11]. In Los Angeles, the total number of taxi-based trips has fallen by 30% [12]. Ridesharing drivers typically tend to earn more than taxi drivers, with many of them forced to join one of the ridesharing-

based companies. Similarly, we have seen further proof of the efficacy of many other businesses utilizing this sharing economy model.

Developers and entrepreneurs have expressed a vast amount of interest in this sector, and thus in ways to develop their own businesses and apps that leverage this shared economy business model. In trying to develop an application that leverages this business model, it is important to examine some necessary technical components. It is necessary to have a backend database that can store user data, transaction data, etc. In addition, the database should be able to handle many concurrent connections, in case it needs to scale rapidly. One may also choose to utilize a Service Oriented Architecture (SOA) or microservice-based architecture approach to their scaling, as does Uber, Netflix, etc. [13].

Lastly, if the application has use cases that adopt geolocation functionality, certain geolocation frameworks or classes, such as Apple's Mapkit or Google Maps, are necessary. It will be shown in later chapters how these technologies work and why they were appropriate for this application.

2.3 Solutions to the Parking Problem

There has been some other notable work on this subject, both in studies designed to explore alleviating the parking problem in general, as well as actual applications currently available and operating commercially. While no true peer-to-peer parking studies or commercial application had been found, this section will go into detail on the type of work that was done.

Previous studies have sought to address the parking issue in various ways. One example has been the use of image processing techniques to captures and processes the rounded images drawn at a parking lot, and to determine where the empty parking spots were located [14]. Sometimes information was obtained about available parking spaces by the use of sensors, real-time data collection, and mobile-phone-enabled automated payment systems that allow early reservation [15]. Another study details service design for intelligent parking based on theory of inventive problem solving and service blueprint [16]. This study sought to improve efficiencies of the particular mall's parking operation, such helping to find available spots, integrating payments via their phone app and improving other functions.

Another example is the use of navigation for customers to parking structures in conjunction with their destination [17]. The purpose is to get parking information directly from the city, various retail spots and urban spots, and to provide navigation to its users based on that information. Users could see information about the nearest parking structures, as well as also add credit to parking meter with their phone. There has also been work done where embedded wireless sensors were used on a network of parking lots and connected to cloud-based management systems to help users better get efficient parking[18].

Current commercial parking applications comprise of either a mobile application in iOS, Android, a web application, or a combination of one or more of these. The vast majority of parking solutions rely on making reservations and booking parking spots from parking garages. One example of such an app is SpotHero.

SpotHero is an on-demand parking mobile app and website-based marketplace that connects parking lots, parking garages and valet services that comprise the marketplace for parking spaces with drivers who need to find and reserve them [19]. Another application is ParkWhiz, which similarly provides both a mobile and web-based interface for finding and purchasing parking spots at parking garages [20]. For both applications, the user may enter their address, as well as a time and date of the parking reservation.

While these apps do have a valid role in the current market, there are gaps where they do not cover the full range of parking problems that users face. From entering CSUN's address into their web application, the closest parking spot available was shown to be Panorama City. In fact, that was one of the few parking spaces available in the San Fernando Valley, and, as was stated before, there are many areas of parking need, such as universities, busy streets, malls, and other areas, that are currently not being addressed. As was described, this is primarily a solution that connects users with parking garages and valet services, and does not fulfill the full breadth of use cases stemming from the parking problem.

The services that they offer are different, in that they provide pre-payment options for parking spots in parking lots and garages in major cities. There are also parking API's available, that allow developers to see available parking spots. Some examples of these API's are ParkWhiz and Parking Panda. As of right now, there is no app that offers a peer-to-peer, shared economy parking solution where users can sell their spots to other users.

2.4 Technical Stack for Solution

This proposed parking application will be written natively in Swift, and will use Firebase for Authentication and database services. Swift and XCode will provide the language and forms for designing the app, as well as some important location-based frameworks and classes. In addition, it will provide a way to implement useful third-party libraries.

Firebase will also be helpful for this app. It will help streamline the registration and login process. It will help with storing various user’s sell requests and buy requests, and will facilitate communication between buyers and sellers. Also, as various user's location changes, it can be tracked in real-time with some of Swift’s methods for calculating location. It can also scale as the user-base grows, and offers free or inexpensive options.

Chapter 3: Methodology, Requirements and Architecture

This section will detail the software process used when creating this app, important functional and nonfunctional requirements, architecture and UML diagrams (sequence and activity diagrams), as well as screenshots, demonstrating how the app works for buyers and sellers.

3.1 Project Goals & Challenges

This section will go into the goals of this application, as well as discuss some of the challenges involved and how they were addressed. The aim of the application was to serve as a service provider that would allow users to connect with other users to either buy or sell their parking spots.

The first challenge was that the application needed to implement accurate location services. When selling a spot, the user's exact location is needed. The buyer's location is also needed when they buy a spot, both so that they can get directions to the parking spot, and to calculate the distance between them and the available spot. Further, the buyer's location needs to be updated constantly, both so the seller can track them and so that both parties can get informed when the buyer arrives. Apple Mapkit and CoreLocation have been established as proven frameworks in the use and implementation of location-based services for iOS apps.

The second major challenge was to find a way to ensure that there was communication between different users. For example, a seller would need to get a message that a buyer wanted to purchase their spot. Similarly, buyers would need to know whether their request was accepted. In order to address this, the app utilized

Firebase's Observe methods. These methods allow for the asynchronous listening of value events in the database, and can also listen for changes in particular fields. When a particular entry in the database appears, or is changed, a message is triggered and sent to the buyer or seller. Further details of this will be addressed in the chapter on technical approach.

Another challenge was combining both the seller and buyer functionality into one app. Many apps with similar features choose to create separate apps for the different parties involved. For example, Uber and Lyft create a different app for both the driver and the rider. Separating apps is helpful because the functionality in each role is different. Doing something similar with this app, however, would prove cumbersome to users. If users would have needed to open a different app, depending on whether they were selling or buying a spot, would have detracted from its usability. Instead, it was determined to use one app that contained both functionalities. This challenge was addressed by using different screens for separate functionalities, as well as using different classes for both the buyers and sellers.

3.2 Functional and Nonfunctional Requirements

Functional Requirements

- A user should be able to sign up and login with a unique email and correct password.
- When signing up or logging in, simple validation (email not in use, password strength, correct email, connection available, etc.) is performed.

- A user can sell their parking spot for a specific price, which will then be advertised to others wanting to purchase it.
- A user can buy a parking spot from a list of available spots, sorted by nearest.
- When a user selects a parking spot, that spot is then visible on a map.
- When a user clicks on “purchase spot,” the seller then receives the request, to approve or deny.
- When a seller accepts a buyer’s request, a map then opens with directions to the parking spot.

Nonfunctional Requirements

- The login and signup systems should work reliably.
- The user should experience a minimalist, clean and friendly user interface.
- The real-time database should be able to scale reasonably, with generations of new parking spot sell requests and buy requests.
- Processing time between the app and database should be fast.
- The database and authorization service should remain available at least 99.95% of the time (as per the Firebase Service Level Agreement).

In figure 3.1, the detailed description of actions is shown in a UML Sequence diagram.

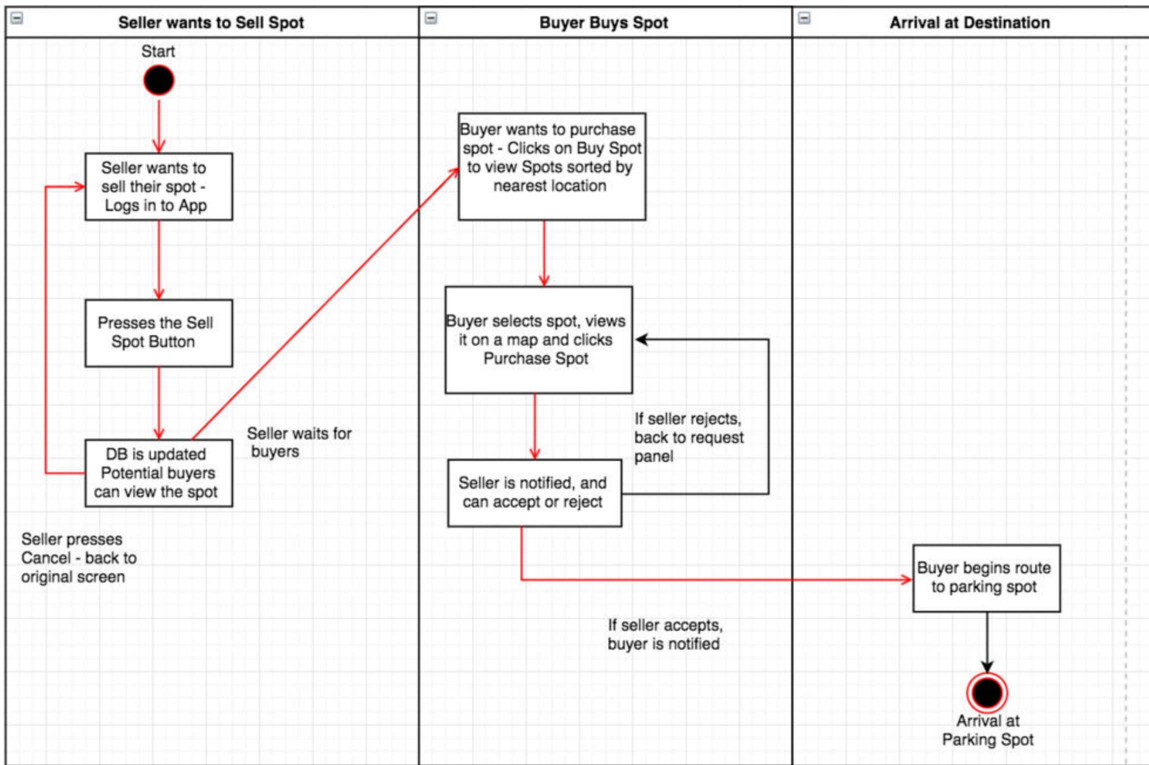

Figure: 3.1 Sequence Diagram

3.3 User Interface

This section will cover the user interface and flow of the application. In figure 3.1, the detailed description of actions is shown in a UML Sequence diagram. The seller or buyer will first log in (or sign up) to the app. Once they are logged in, they will be on the main page of the app (shown below in figure 3.2), where they can choose to either sell or buy a parking spot. If they press the sell spot button, the Firebase database is then updated with a request entry, containing the user's email, GPS location, and unique Firebase user ID. At this point, the "Sell Spot" button then becomes a "Cancel" button, where the seller can cancel the sale of their parking spot.

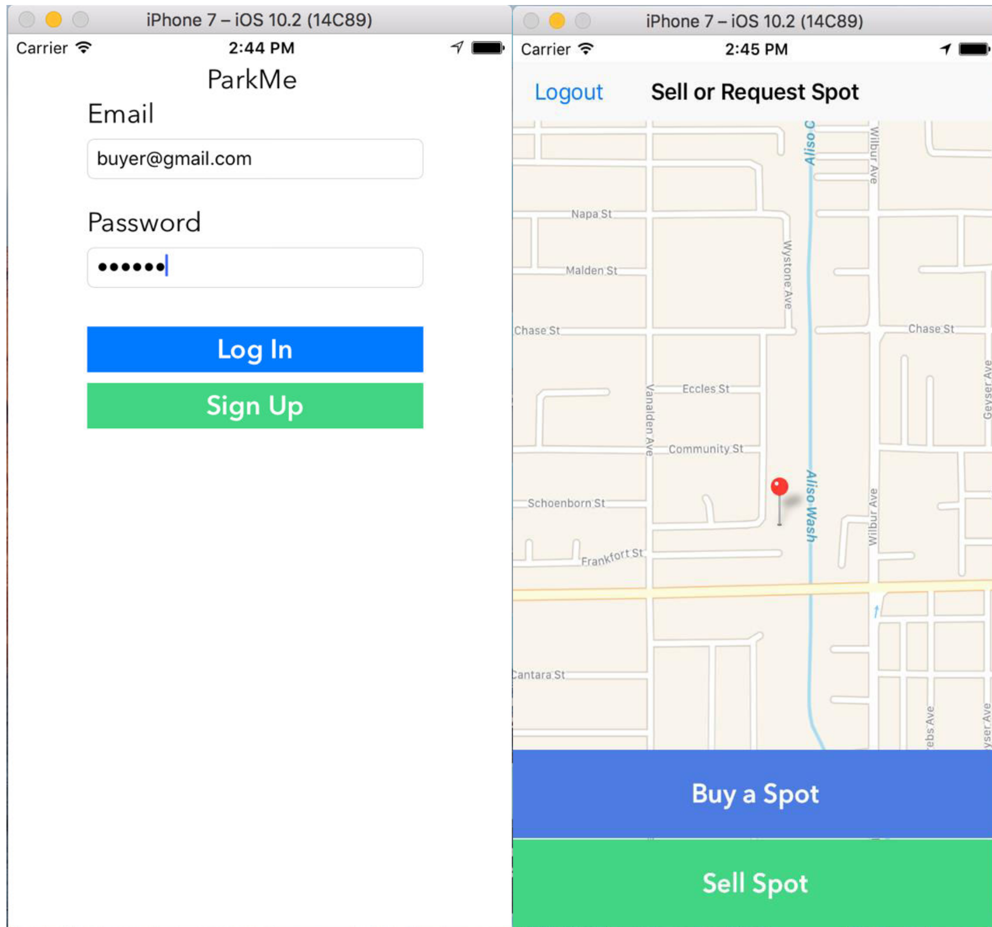

Figure: 3.2 Login/Signup Page and Main Screen

In figure 3.2, the login/signup and main page are shown. When signing up or logging in, standard validation checking is performed, where email validity and password are checked, password strength, whether there is a problem connecting, as well as if the email is already in use. Once on the main screen, if this is the first time using the app, the user is asked for their permission for the app to use their location. Once that is done, they are shown their location on the main map. If the user clicks on “Buy a Spot,” they will then be displayed a list of parking spot locations, sorted by nearest.

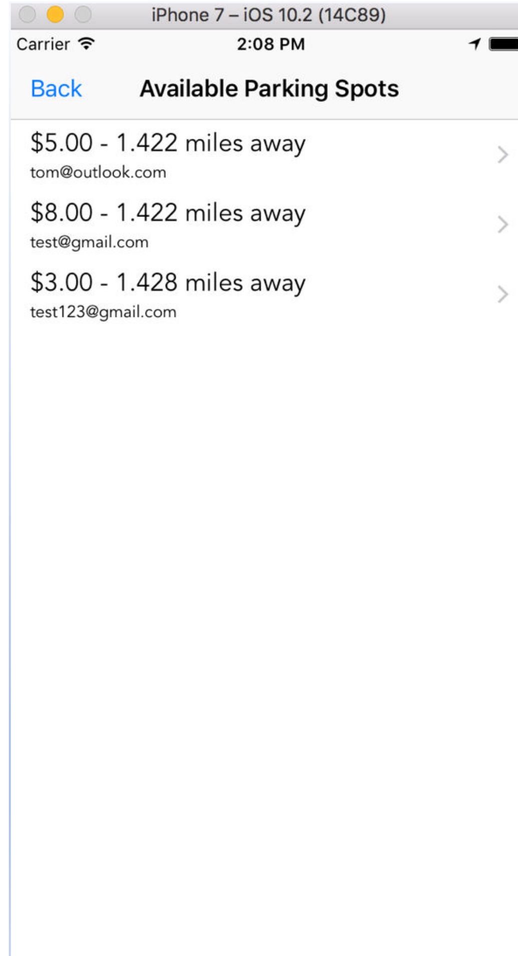

Figure: 3.3 Available Parking Spots Screen

Figure 3.3 shows available parking spots, sorted by distance. It uses a standard table view, and shows the user's email and distance from the potential buyer. Once the buyer selects the desired parking spot, they will then be taken to the next screen (shown in figure 3.4), where they can then view the available spot on a screen, see the seller's username and price and purchase the parking spot.

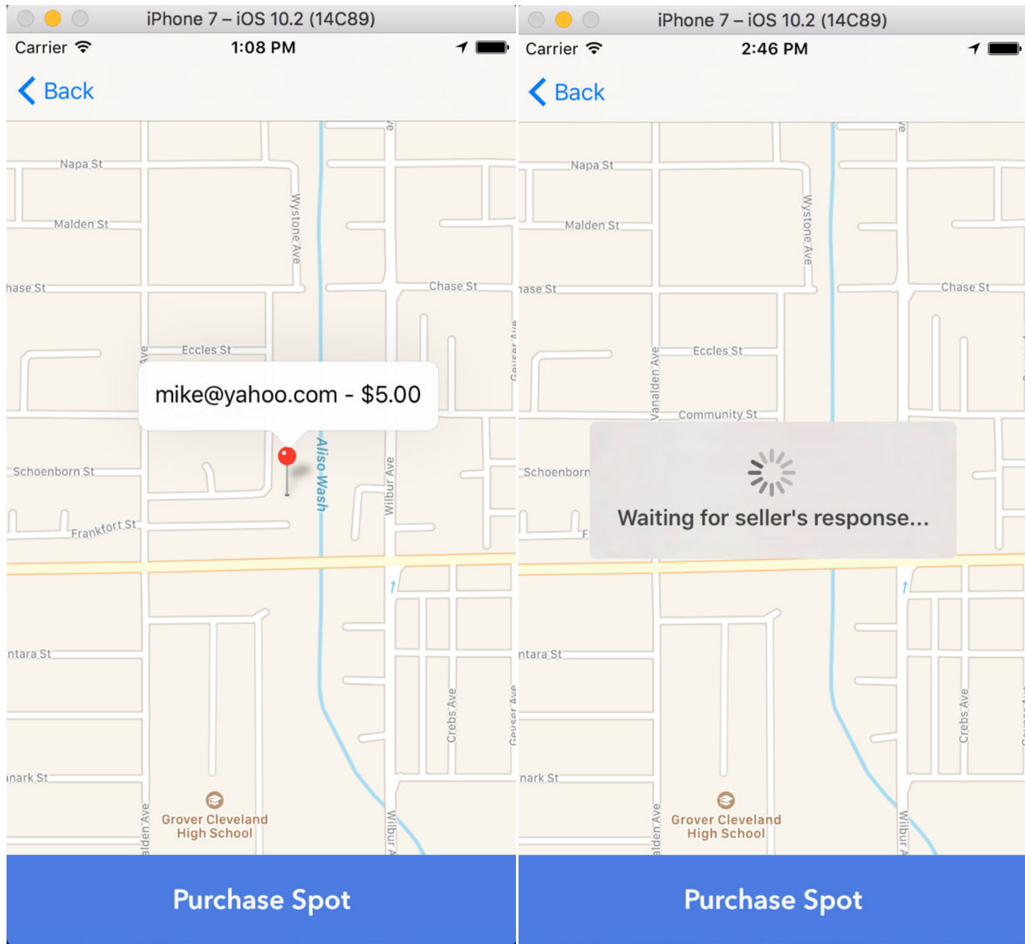

Figure: 3.4 Purchase Spot Screen

Once the buyer clicks on "Purchase Spot," the Firebase Database is updated with a new buy request entry. This includes the parking spot GPS location, seller name, buyer name and the unique Firebase buyer ID. At this point, the buyer is left waiting for the seller to respond to their request.

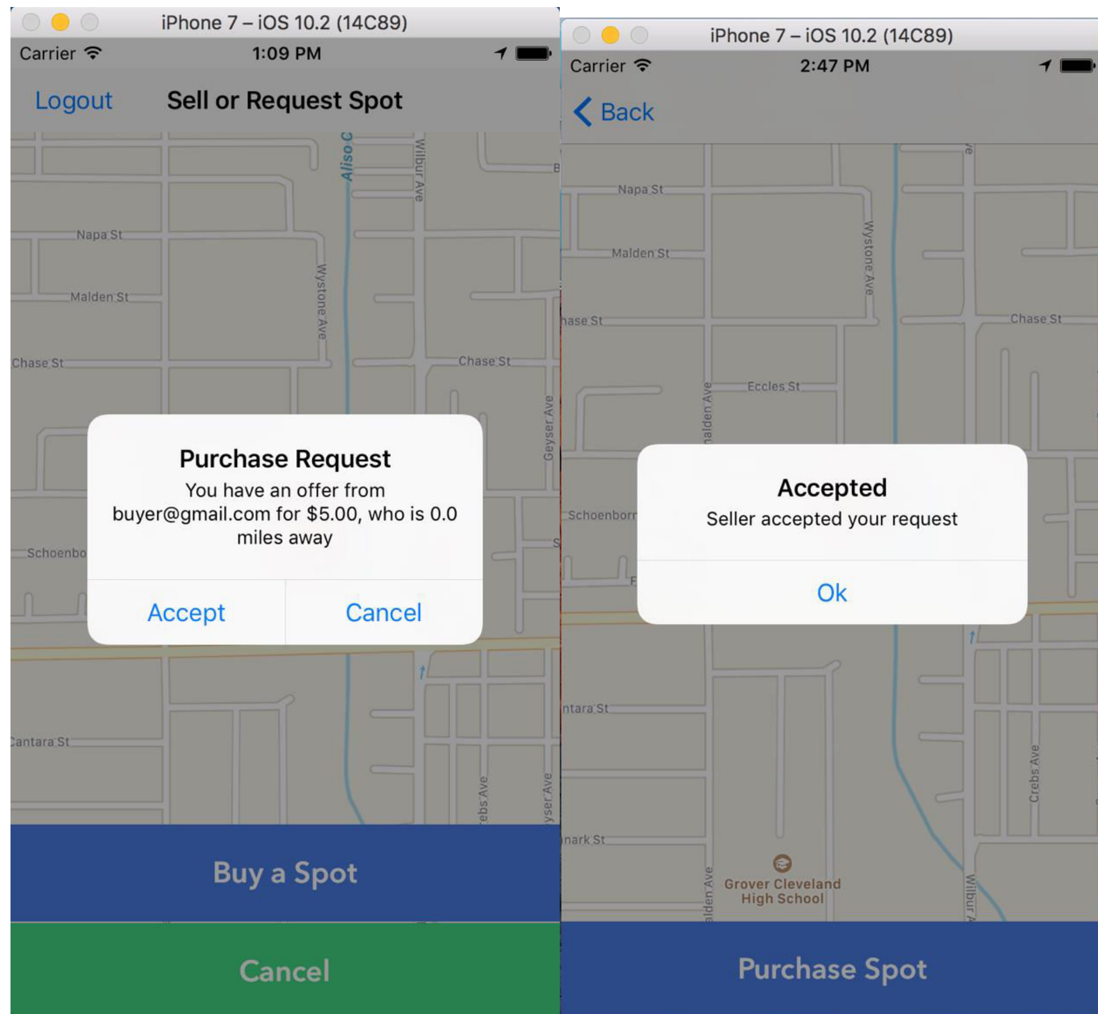

Figure: 3.5 Seller Receives Request and Accepts

Figure 3.5 shows the seller receiving the parking spot request from the buyer. This request shows the buyer's name, their distance (in miles) from them, as well as the price of the spot. They can then choose the "accept" or "cancel." If they cancel, the buyer will receive a message that the seller canceled their request. If they click on accept, the buy request entry that was created gets updated with a new entry, indicating that the seller accepted the request. The buyer will then get a message that the seller accepted their request.

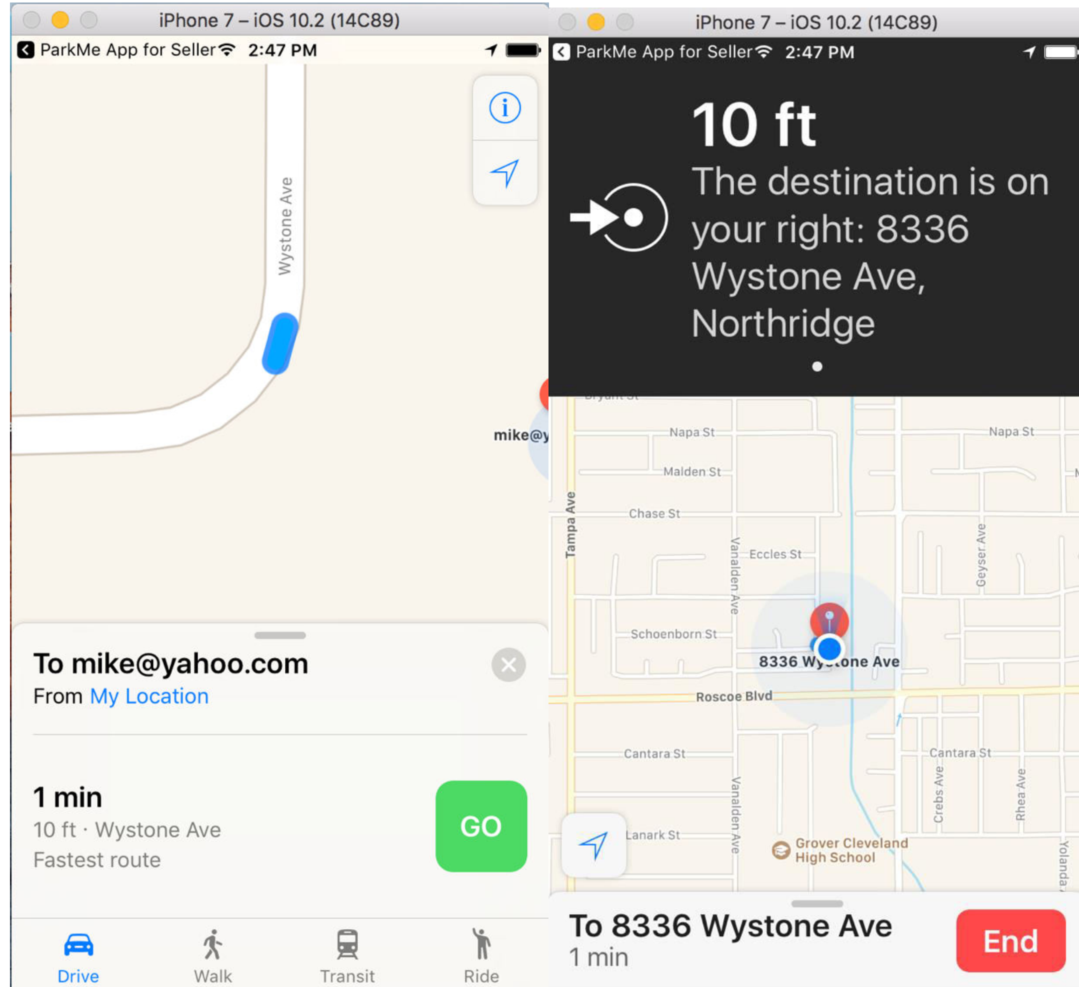

Figure: 3.6 Apple Directions

Once the buyer clicks on "Ok," the starting and destination location are then automatically put into Apple Directions, a route is generated and the buyer is given directions to the parking spot, until they arrive, or end the directions. Once the transaction is complete, a stub is generated in the Firebase database, containing the buyer email, seller email and unique ID (UID). Once the buyer travels to within 10 feet, both the buyer and seller receive a notification that the buyer has arrived (Figure 3.7). The seller is thanked, and instructed to vacate their parking spot for the buyer.

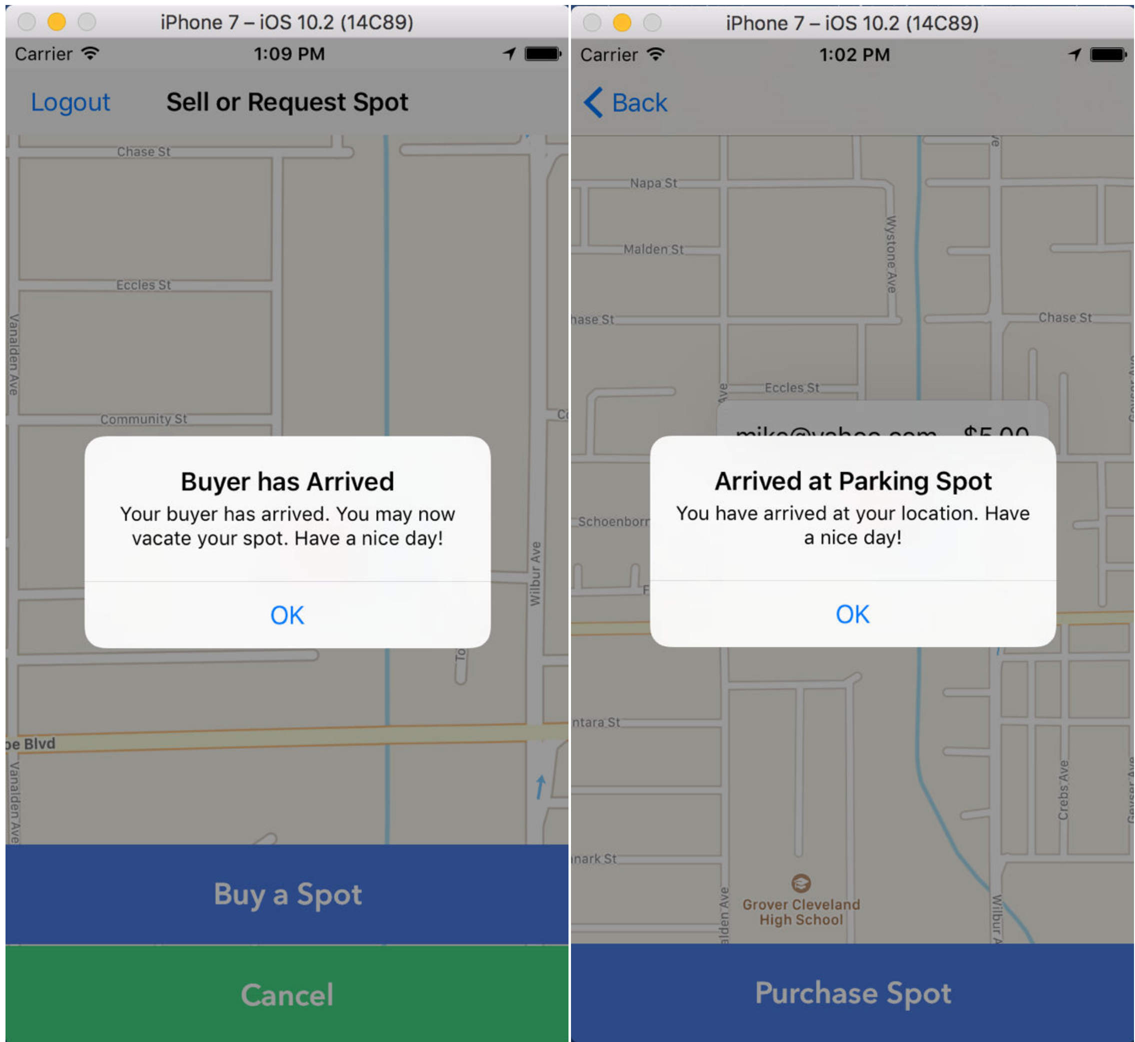

Figure: 3.7 Arrival Notifications

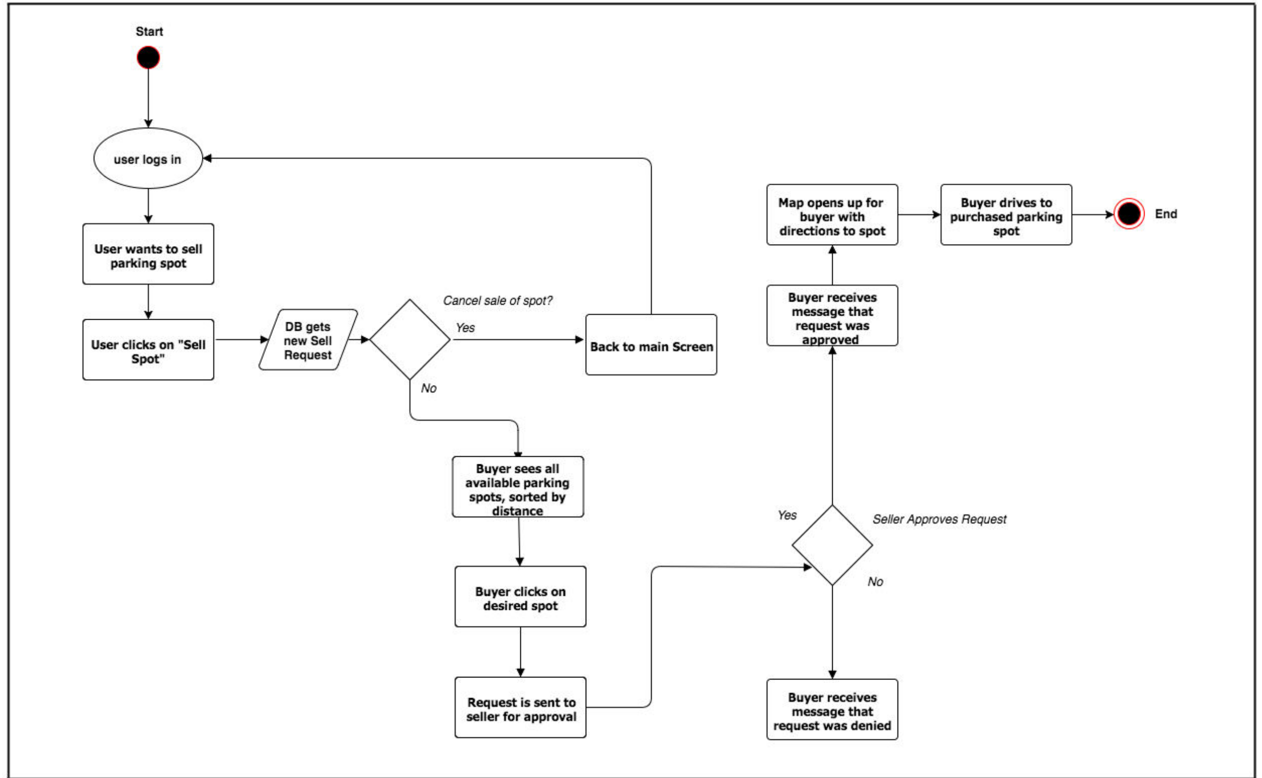

Figure: 3.8 Activity Diagram

Figure 3.8 presents a high-level overview of the dynamic aspects of the system (from the perspective of a UML Activity Diagram). The workflow begins from logging in, selling the spot, the buyer viewing available spots, and then choosing a spot and buying it. Decision nodes are placed and determine what happens if a sale of spot is cancelled, as well as what happens if the seller approves or denies a request.

Chapter 4: Technical Approach

The primary programming language that was used was Swift. Swift is a general-purpose, multi-paradigm, compiled programming language developed by Apple Inc. for iOS, macOS, watchOS, tvOS, and Linux[1]. Ever since Swift has been introduced in 2014, it has continued to rise in popularity. According to the TIOBE Programming Index, as of November 2016 statistics, Swift is currently ranked 12th of all programming languages, just one behind Objective C, which is ranked eleventh[2]. Since Swift features a modern syntax (like Java, C#, Go, etc.), it can be an advantageous language to learn for newer iOS developers. In addition, it has been receiving active support and promotion from Apple, and is open source[3].

4.1 Swift 3.0

Swift 3.0 was released on September 13th, 2016, and precedes the previous version of 2.3. It is currently the latest version of the programming language. Some major changes include new API design guidelines, Foundation types, simplification of parts of the language and method names, as well as other changes.

4.2 Cloud Server - Firebase

Information such as user data, transaction data and possibly other information will have to be stored in a database. Firebase is a mobile platform that helps developers quickly develop high-quality apps, grow their user base, and earn more money - They are

---

[1] http://developer.apple.com/swift
[2] http://www.tiobe.com/tiobe-index
[3] http://swift.org/migration-guide

part of a new, growing trend of services, called Backend as a Service (BAAS), or Mobile Backend as a Service (MBAAS). A BAAS is a model for providing web and mobile app developers with a way to link their applications to backend cloud storage while also providing features such as user management, push notifications, and integration with social networking services[4].

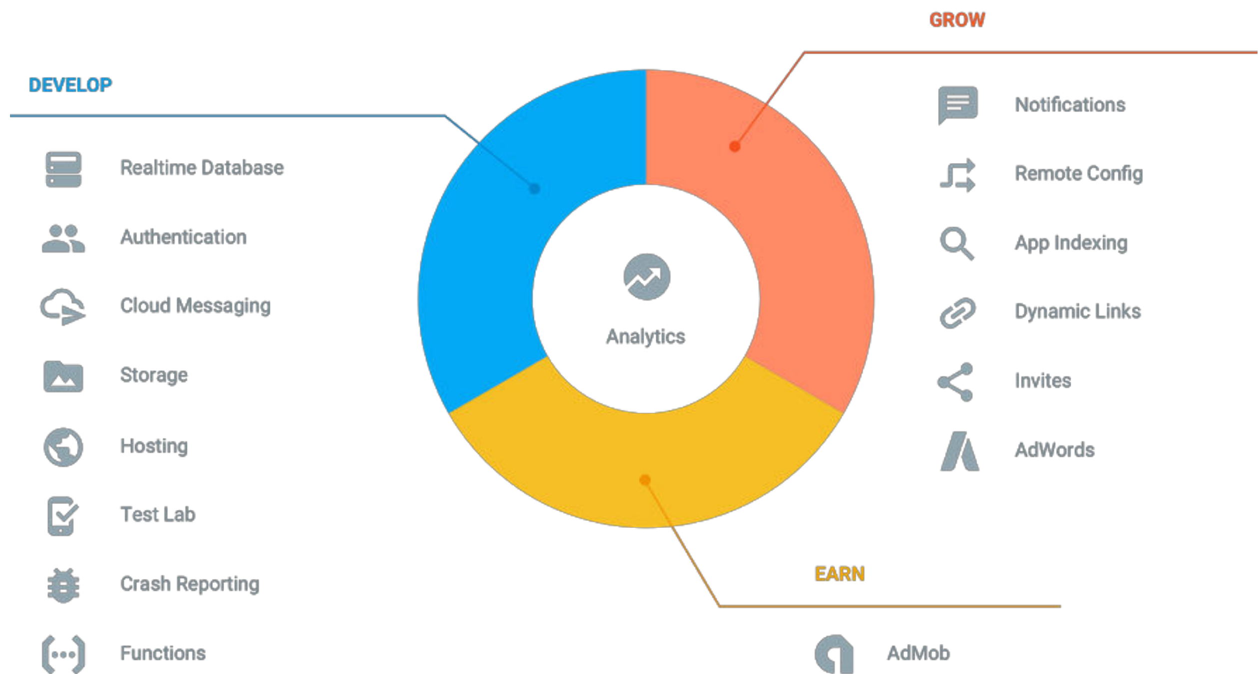

Figure: 4.1 Firebase Services

Firebase offers a suite of services, including storage, hosting, a real-time database and other features. Amazon Web Services offers reliable, scalable, and inexpensive cloud computing services[5]. Creating a backend on Amazon Web Services was considered - there would have been several ways to accomplish this. One way would be to create a server with all of the backend code for registration, databases, etc., and deploy it on an EC2 instance. Another possible way would have been to use a serverless architecture,

---

[4] http://firebase.google.com/
[5] http://aws.amazon.com/

using an API Gateway to enable the mobile application to call custom Lambda functions, and to store information in one of Amazon's databases (like DynamoDB or RDS). The advantage of using Firebase, and why it was ultimately chosen, was that it would be easier to implement, and would require less server-side code. Also, while AWS also offers their own BAAS service, called AWS Mobile Hub, Firebase was chosen because of their suite of features, ease of implementation, healthy developer community and rich API reference/documentation. In addition, they have a Service Commitment to maintain a Monthly Uptime Percentage of at least 99.95%.

Firebase supplies their own custom SDK that needs to be integrated with XCode by use of pods, or CocoaPods, which is essentially a dependency manager for Swift and Objective-C projects. To integrate the Firebase SDK with a given project, one must download and import the project's Google Services plist to their XCode project, and then create a Podfile in the project directory, with the command, "pod init," write the dependencies in the podfile (i.e. pod 'Firebase/Core'), and then enter the command, "pod install," and the user will have a new file that can be opened, which will be integrated with the Firebase SDK.

4.3 Maps - Apple MapKit

Maps will be needed for the application, so that users can see where they are and where the target parking location is. The simplest solutions would be to use either the Google Maps SDK or the native Apple Maps. Google Maps allows for the use of an array

of map styles, custom markers, info windows and polylines[6]. The custom markers are able to be used to display available parking spots.

It was decided to use native Apple Maps and MapKit for this project. The reasons were several-fold, but mainly because there were native features that could not have been gotten from Google Maps, such as animations and response to touch. The main classes and frameworks used to make the maps, location services, annotations and other features work were CoreLocation, MKMapView and CLLocationManager.

One of the useful frameworks used was CoreLocation, which helps to determine the current location or heading associated with a device. The framework uses the available hardware to determine the user's position and heading. It can also be used to define geographic regions and monitor when users cross the boundaries of those regions[7]. Both the seller and buyer location was obtained using CoreLocation.

The MKMapView class comes directly from the native MapKit framework. It is the primary way in which maps are displayed with the MapKit framework. The MKMapView object provides an embeddable map interface, similar to the one provided by the Maps application. Maps can be centered on a given coordinate, the size of the area to display can be specified, and the map can be annotated with custom information[8]. It is necessary to call the MKMapViewDelegate before using the class. This class was very useful in allowing both the buyers and seller's names to be displayed on the map, showing their emails as custom annotations.

---

[6] http://developers.google.com/maps/documentation/ios-sdk/
[7] http://developer.apple.com/reference/corelocation
[8] http://developer.apple.com/reference/mapkit/mkmapview

Lastly is the CLLocationManager class. The CLLocationManager class is the central point for configuring the delivery of location and heading-related events to an app. An instance of this class can be used to establish the parameters that determine when location and heading events should be delivered and to start and stop the actual delivery of those events[9]. The CLLocationManager needs to be used for any location services with MapKit, and assisted in determining both the seller and buyer's location. As is the case with MKMapView, the specific delegate must be called, which in this case is MKMapViewDelegate, before using its features.

4.4 Email Registration & Login with Firebase

It was deliberated whether to build a system with standard log-in, requiring an email and password, an Oauth2 implementation, where users can sign-in with their Facebook or Google account, or both. It was ultimately decided to use an email sign-up and login system. The reason for this was because the user's email was needed to store in the database, to use in future sell and purchase requests. It was also important to keep the app design clean and minimalist, and implementing both may have cluttered the screen.

[9] http://developer.apple.com/reference/corelocation/cllocationmanager

```
func login(withEmail: String, password: String, loginHandler: LoginHandler?){

    FIRAuth.auth()?.signIn(withEmail: withEmail, password: password, completion: { (user, error) in

        if (error != nil){
            self.handleErrors(err: error as! NSError, loginHandler: loginHandler);
            } else{
            loginHandler?(nil);
        }
    });
}

func signUp(withEmail: String, password: String, loginHandler: LoginHandler?){

    FIRAuth.auth()?.createUser(withEmail: withEmail, password: password, completion: { (user, error) in
        if (error != nil){
            self.handleErrors(err: error as! NSError,
                loginHandler: loginHandler);
        } else {
            if (user?.uid != nil){

                // store user to DB
                DBProvider.Instance.saveUser(withID: user!.uid, email: withEmail, password: password);

                // login the user
                self.login(withEmail: withEmail, password: password, loginHandler: loginHandler);

            }
        }
    });

}
```

Figure: 4.2 Login and Signup Code

To create login and signup functions, Firebase Auth (FIRAuth) was imported into the project, and the specific methods were implemented for logging in and signing up. When signing up, a username (email) and password are taken. The data is then passed into a completion handler that takes user and error parameters. If there are any specific errors, they are handled by the loginHandler, which includes the main errors that could occur (i.e. email taken, email not valid, weak password, etc.). Once that occurs, if the user's unique Firebase ID (UID) is not null, basic information is then stored into the database, and the user is logged into the app. The login function works similarly, albeit more simply (because of less things to do) than the signup function. The functions are then ready to be called by IBActions in the LoginVC.

4.5 Communication between Buyers and Sellers

In building this app, one of the more complex tasks was determining how the buyers and sellers would communicate. In the database, whenever a user chooses to sell their spot, a sell request entry is generated (shown in figure 4.3, below). This sell request entry is a child node of the Sell_Request JSON object. This includes their GPS location (latitude and longitude), their name/email, as well as their UID. In addition, it was decided that each customer that wanted to purchase a spot would create a buy request entry (figure 4.4).

Figure: 4.3 DB Sell_Requests

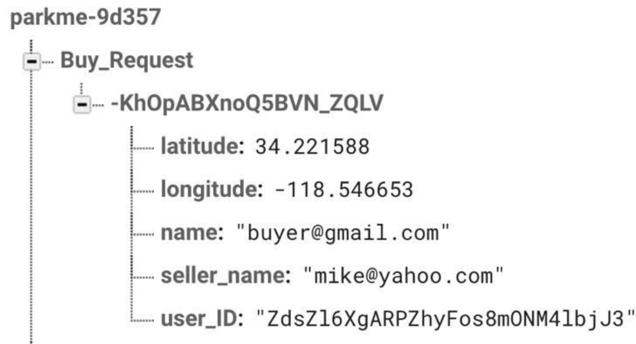

Figure: 4.4 DB Buy_Requests

```
func acceptOffer(buyer: String, accept: Bool){
    DBProvider.Instance.buyRequestRef.queryOrdered(byChild: "seller_name").queryEqual(toValue: self.seller).
        observeSingleEvent(of: .value, with: { dataSnapshot in
        let enumerator = dataSnapshot.children
        while let buy_request = enumerator.nextObject() as? FIRDataSnapshot {
            let data = buy_request.value as? NSDictionary
            if (data?["name"] as? String) == buyer {
                buy_request.ref.child("accepted").setValue(accept)
                if accept == true {
                    self.delegate?.updateRequesterLocation(lat: data?["latitude"] as! Double, long: data?["longitude"] as!
                        Double)
                }
            }
        }
    })
}
```

Figure: 4.5 Function to accept offer

In the "acceptOffer" function, what is being done is that Firebase is looking at the database, in "Buy_Request," taking seller_name, and is checking whether or not the seller is equal to the current user/seller. It is observing a single event and is passing the data to a completion handler. It then takes a snapshot of the data, or FIRDataSnapshot, and sets an object casted to an NSDictionary (data structure that can contain key-value pairs) equal to it. The seller is then prompted via delegation whether they want to accept this particular offer, and if they accept, a new nested object called "accepted" is created in the buy_request entry, and is set equal to "true." In separate code for the buyer, they are looking to see if accepted is equal to try. If it is, they are then notified that the request

has been accepted, which then triggers Apple Directions to start. Since Firebase offers a real-time database, with both the buyer and seller code, both are in constant states of listening for events, utilizing Firebase's asynchronous observer methods/functionality. These functions are listening to see if a particular child changes (added or removed) or appears. The result of which ends up triggering certain events that make the code work, and keeps buyers and sellers constantly in sync.

4.6 Payment - Stripe

Users will need to pay for parking spots, so a payment system will have to be implemented on the app. Stripe is a mobile payment service that offers credit card processing for all major credit cards. To install, it is required to add the Stripe pod to the project's podfile (where dependencies are stored), install the pods, and import Stripe into the project.

Chapter 5: Design Patterns Utilized

The following is an overview of the design patterns that were used in order to accomplish the goals of the proposed project, as well as the rationale and advantages for each pattern.

5.1 Protocol and Delegate Pattern

A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements[10]. Protocols allow two classes to communicate between one another via inheritance to accomplish a certain goal. Within the set of protocols, there are both formal and informal protocols, as described below:

- *Formal Protocol*: A formal protocol declares a list of methods that client classes are expected to implement. Formal protocols have their own declaration, adoption, and type-checking syntax.
- *Informal Protocol*: An informal protocol is a category on NSObject, which implicitly makes almost all objects adopters of the protocol. Implementation of the methods in an informal protocol is optional. Before invoking a method, the calling object checks to see whether the target object implements it[11].

[10] developer.apple.com/library/content/documentation/Swift/Protocols.html
[11] developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/Protocol

Essentially, formal protocols require the class that is using it to conform completely to its required methods. Failing to do so will result in an error, with the program being unable to run.

5.2 Model-View-Controller

Much of this application was built utilizing a Model-View-Controller (MVC) design pattern. In MVC, the View displays information to the user and, together with the Controller which processes the user's interaction, comprises the application's user interface. The Model is the portion of the application that contains both the information represented by the View and the logic that changes this information in response to user interaction[12]. This design pattern has many benefits, but for this application, the main benefit was separation of concerns, and decoupling of functionality. Some examples of this are the AuthProvider class, as well as the class for Constants, DB Provider, a handler class for parking functions, Sign in view controller, etc.

All of the Firebase authorization logic (Sign-in, Sign-up, etc.) is encapsulated in the AuthProvider class. In addition, the handleErrors function, takes all, or most, of the possible Firebase error codes (i.e. *.errorCodeWrongPassword, .errorCodeInvalidEmail, etc.*) and returns a meaningful message for each. In SignInVC, outlets are created for textlabels, buttons, etc. The only thing that is done is calling the AuthProvider class, and the subsequent methods from that class, and passing the input from the textlabels into the pre-coded functions, as well as handle the segues to the next view controller, with the Sign-In and Sign-Up buttons.

---

[12] http://www.jot.fm/issues/issue_2009_03/column5/index.html

Chapter 6: Application Testing

This section will detail the testing that has been done for this app, and will include a simulation-based test with real users, as well as performance testing of the Firebase database with JMeter and BlazeMeter.

6.1 Real World Test

In order to determine whether this application would be successful in a real world setting with real users, it was important to determine whether a successful user experience could be achieved. A quantifiable way by which this could be measured would be by conducting a trial in which groups of users of different sizes would use the application normally. The average time to perform key functions of the application for each group would then be measured and compared, to help determine whether the application could perform optimally once given some amount of scale.

Google Analytics was chosen as the tool to gather quantifiable data about customers, and their app experience. Google Analytics is an analytics service that offers free and enterprise analytics tools to measure website, app, digital and offline data to gain customer insights [22]. There were several reasons for choosing this analytics platform, some being ease of use and others being that it has been offered for a while and is backed by a large infrastructure.

The most important reason this service was chosen was because of their ability to track specific events, such as button clicks or notifications received, and to pass along custom data to that event. Once Google Analytics is set up, a developer is able to track specific events anywhere in their code, and can also pass along any information they

want. In the case for this application, a timer class was created, so that specific user actions could be measured (in seconds). The measured amount of time was then passed along to Google Analytics.

Specifically, what was measured was registration and login time, as well as the time needed to sell a parking spot, purchase a parking spot, etc. Additional measurements were taken for sellers on the amount of time to receive a response from a buyer (measured from when the spot is listed for sale), and how much time it would take the buyer to arrive at the seller's destination. Arrival time was measured both from the time the parking spot was listed for sale, as well as the total time the seller had been on the app.

6.2 Study Details

The application was distributed among a number of users in three different trials, and testing was done with those users to determine whether the application performed well as additional users were added. The trial included groups of 2, 10 and 20 participants. Reasonable time constraints (8-hour window) were given to ensure that users would be using the app concurrently, and thus simulating scale. Other than a brief description of what the application does, study participants were not instructed on how to use the application.

In addition to the time constraint, a constraint of 10-square miles (from CSUN) was imposed on all trials. The results were exported from Google Analytics within a certain date range. All values are provided in seconds.

The results for Trial #1, conducted with two people, are shown below:

Event Label	Avg. Value (in seconds)
Login	21
Sell Spot	32
Select Spot	15
Purchase Spot button pressed	48
Response from Buyer After Listing Spot	25
Buyer Arrival After Listing Spot	98
Buyer Arrival - Total Time	134

Table 6.1 Trial #1

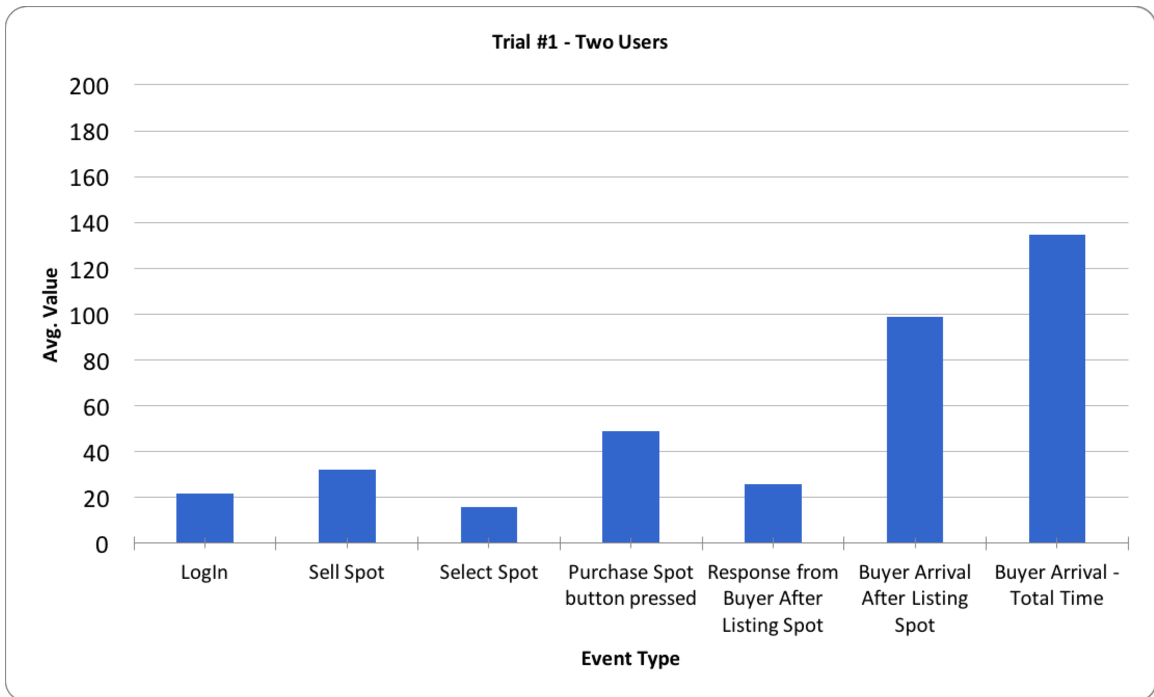

Figure 6.1 Trial #1

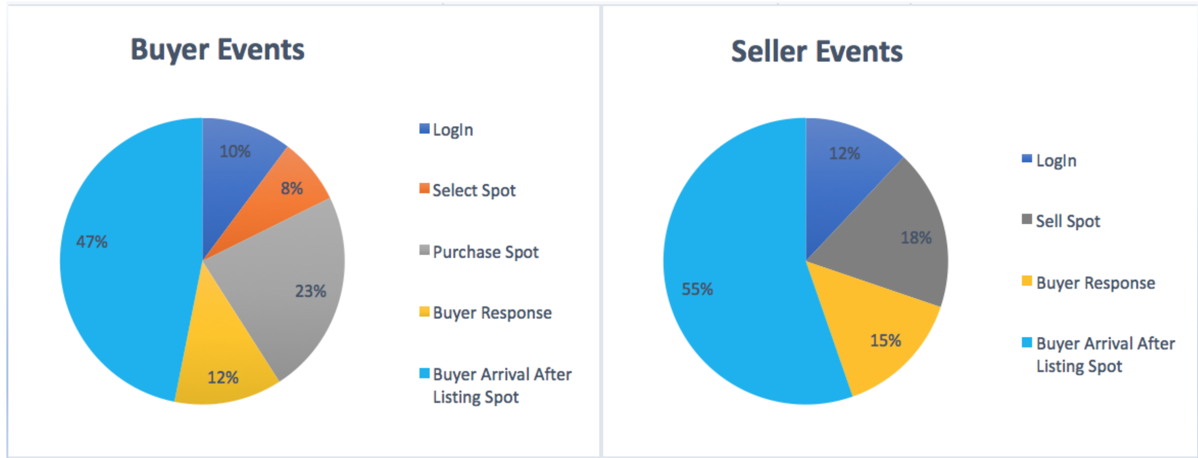

Figure 6.2 Trial #1 Buyer and Seller Event Distribution

For trial #1, with two users, very fast response times are shown for events on all aspects of the app. All participants received the same amount of instruction on our how to use the app (i.e. only a brief description of what the app does), so familiarity of the app is excluded as a possibility.

Figure 6.2 shows the time distribution of tasks, split up between buyers and sellers. Initially, it is shown that buyers spend about half of their time in the application purchasing a spot, with the other half spent traveling to the seller's parking space. For seller's, most of their time is spent waiting for a buyer's response and waiting for them to arrive (70% total). Further trials are conducted to help determine whether these event times continue at the same rate.

The results for Trial #2, conducted with ten users, are shown below:

Event Label	Avg. Value (in seconds)
Login	43
Sell Spot	137
Select Spot	13
Purchase Spot button pressed	127
Response from Buyer After Listing Spot	57
Buyer Arrival After Listing Spot	127
Buyer Arrival - Total Time	180

Table 6.2 Trial #2

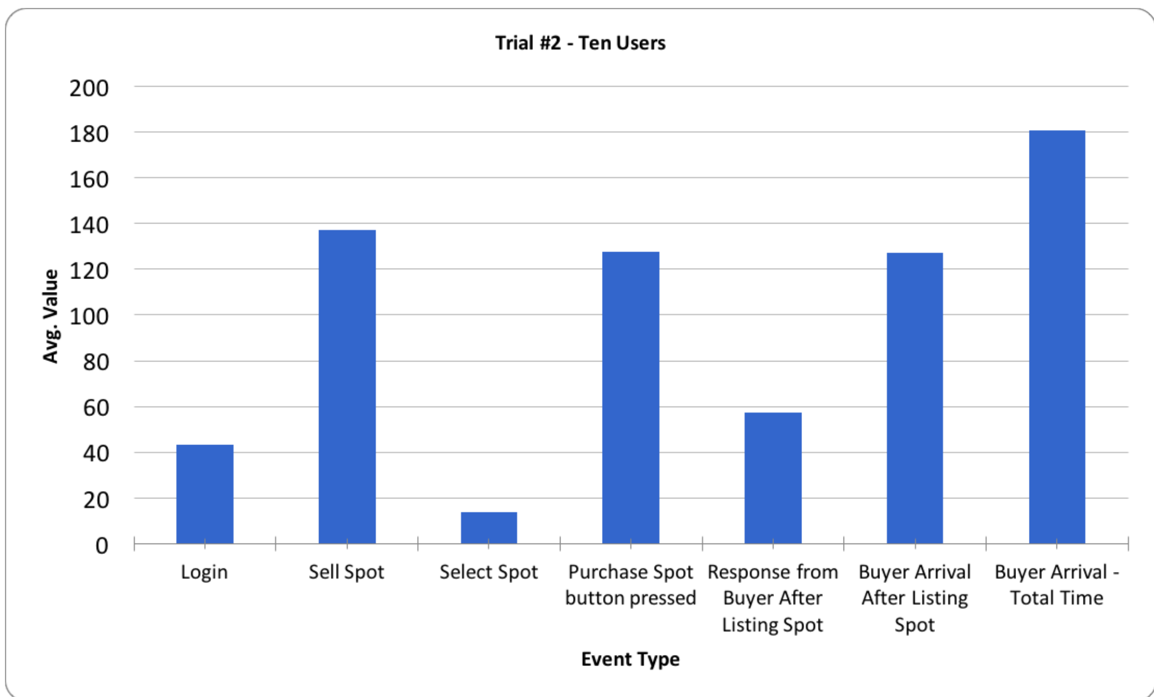

Figure 6.3 Trial #2

The data indicate that more time was taken for participants in trial #2 to complete basic functions of the application on most factors. It is actually significantly more, especially with selling and purchasing a spot. With the two-user trial, 32 and 48 seconds are taken for listing a spot for sale and buying a spot, respectively, and increases to 137 and 127 seconds with the trial with 10 users.

The increase is also seen in total time distribution for all tasks, taking 23% and 18% (figure 6.2) of all time spent in the app to purchase and sell a spot, increasing to 34% and 37% (figure 6.4) of those same events in the second, 10-user trial. Further studies with larger sample sizes are needed to help determine whether these data stay consistent or if they trend more towards the first trial.

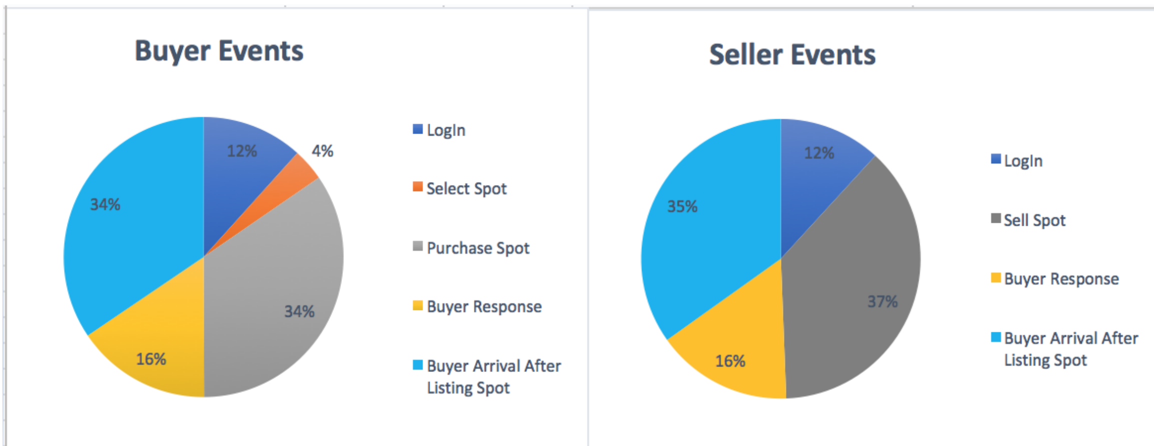

Figure 6.4 Trial #2 Buyer and Seller Event Distribution

The results for Trial #3, conducted with twenty users, are shown below:

Event Label	Avg. Value (in seconds)
Login	33
Sell Spot	75
Select Spot	8
Purchase Spot button pressed	108
Response from Buyer After Listing Spot	51
Buyer Arrival After Listing Spot	81
Buyer Arrival - Total Time	193

Table 6.3 Trial #3

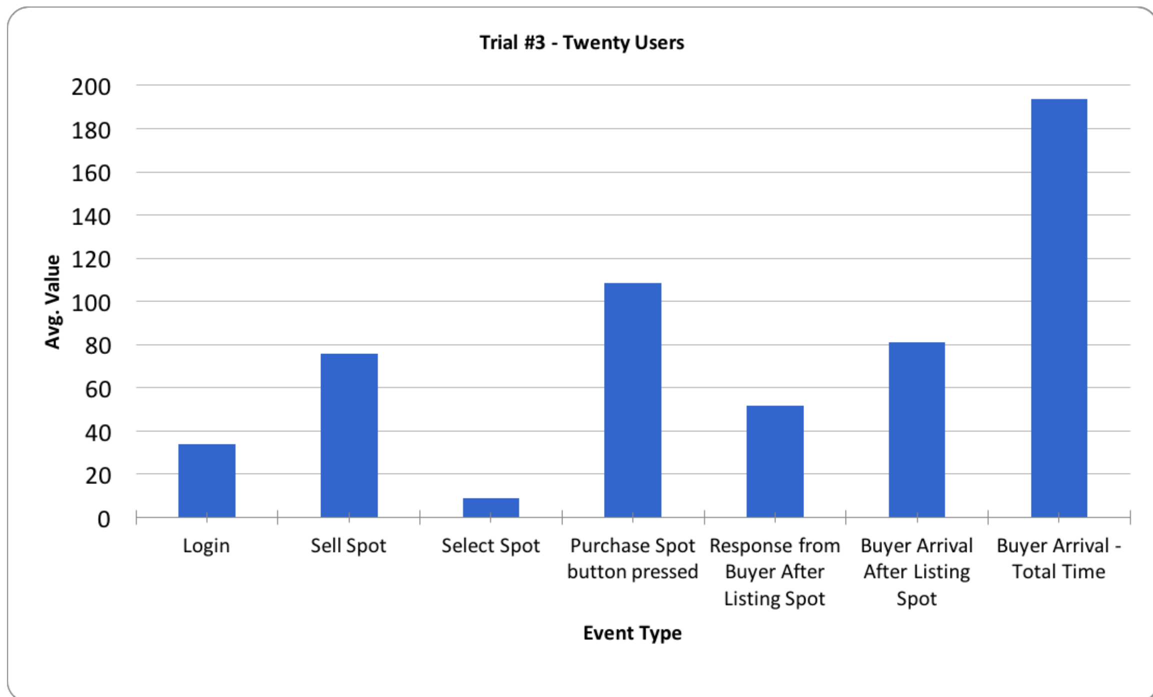

Figure 6.5 Trial #3

The data shows that, similarly to trial #2, more time was taken to complete basic functions in the application. In this trial, however, less time was taken for almost all functions than with trial #2 (see Figure 6.9, in section 6.3). There was also a slight improvement from trial #2 to trial #3, although the values seem to start to become more

consistent when trial #1 is excluded, leading to the possibility that trial #1 could have misrepresented values, or could have somehow been an outlier. It is also possible that issues may exist with scalability. However, while further studies and tests are needed to more conclusively determine whether trial 1, or trials 2 and 3, represent the real amount of time users will need to spend on specific app-related functions, the data seem to trend toward the trials with the larger sample sizes.

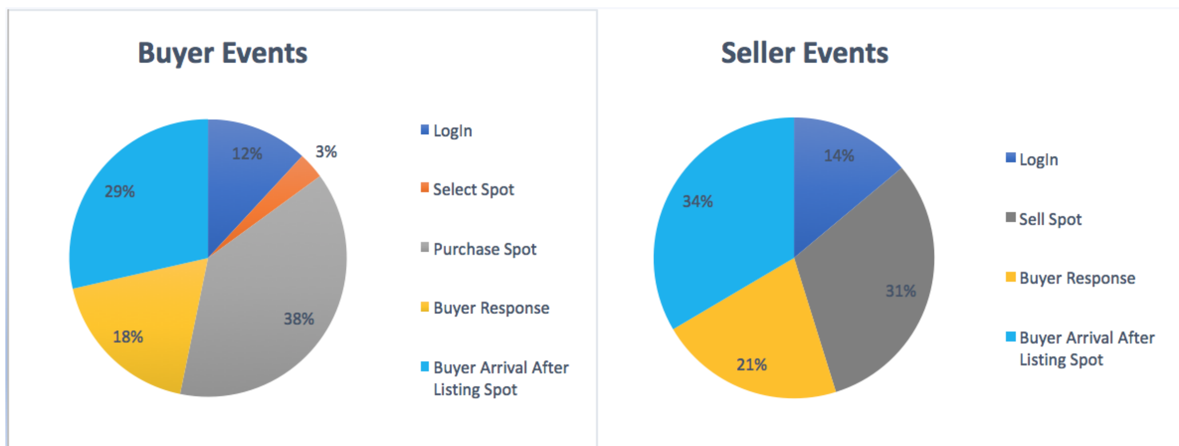

Figure 6.6 Trial #3 Buyer and Seller Event Distribution

Google Analytics also provided demographics information (as shown in figures 6.7 and 6.8) that were used for this study. The information provided shows that the sample size of users chosen was well balanced, in terms of both age, as well as gender. Also, although Google Analytics does not specifically provide it, users used a variety of iOS devices, ranging from iPhones 5, 5C, 6, 6 plus, 7, as well as a few iPads.

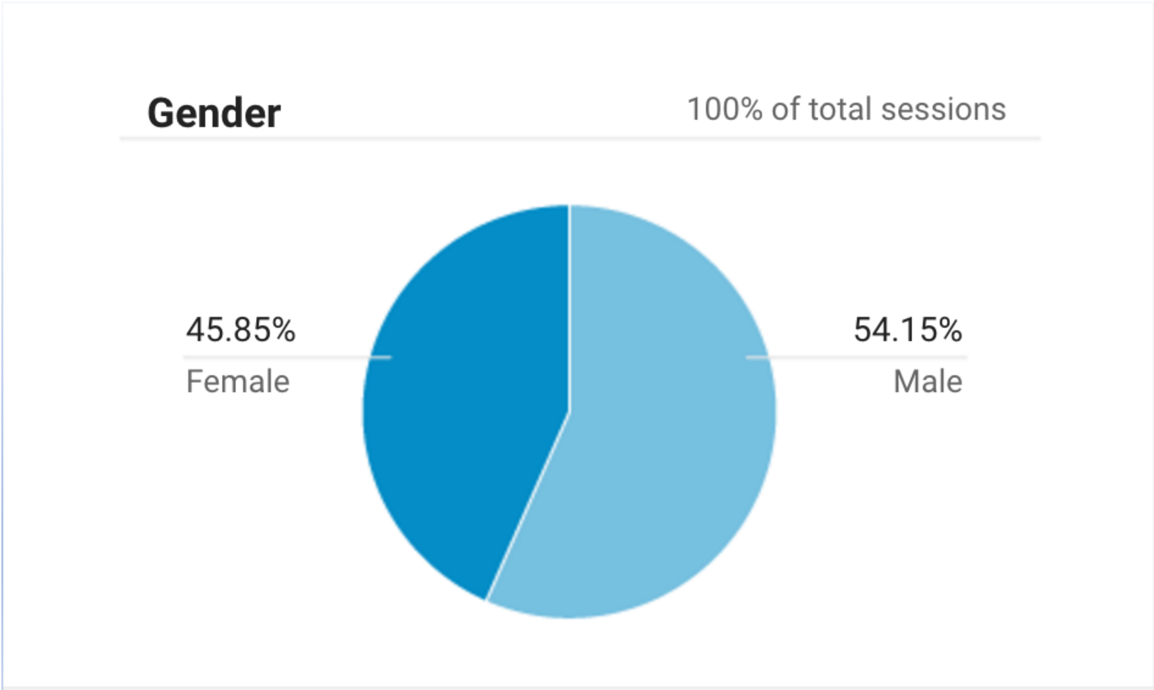

Figure 6.7 Demographics - Gender

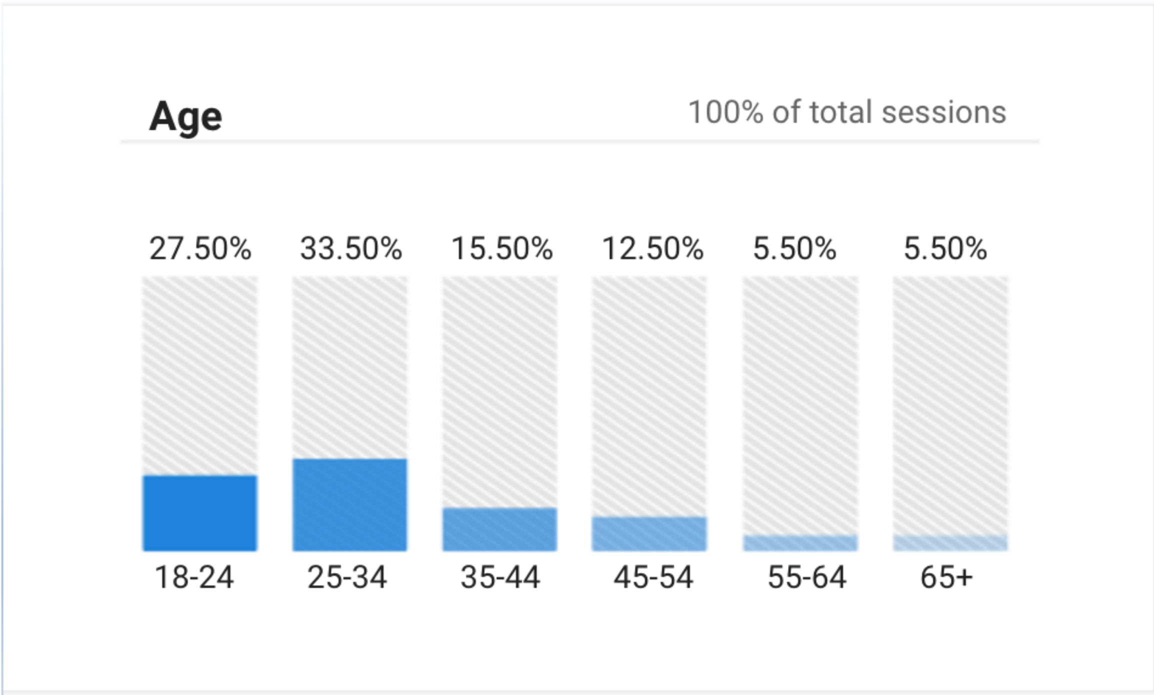

Figure 6.8 Demographics - Age

6.2 Study Analysis

The goal of the study was to determine whether the application could perform optimally once it was given some amount of scale. Figure 6.7 shows a comparison of the different studies.

The results indicate that users took more time to complete the application's basic functions with more users. Familiarity with the application should be ruled out, as all users had the same instruction prior to use of the app. It is also possible (although less likely because of sample size) that there are issues with scalability.

Although issues with scalability are less likely the problem, the ability to scale is further analyzed with automated testing in the next section. Regardless, the data sets indicate that most users were able to complete basic functions in a reasonable amount of time in all studies.

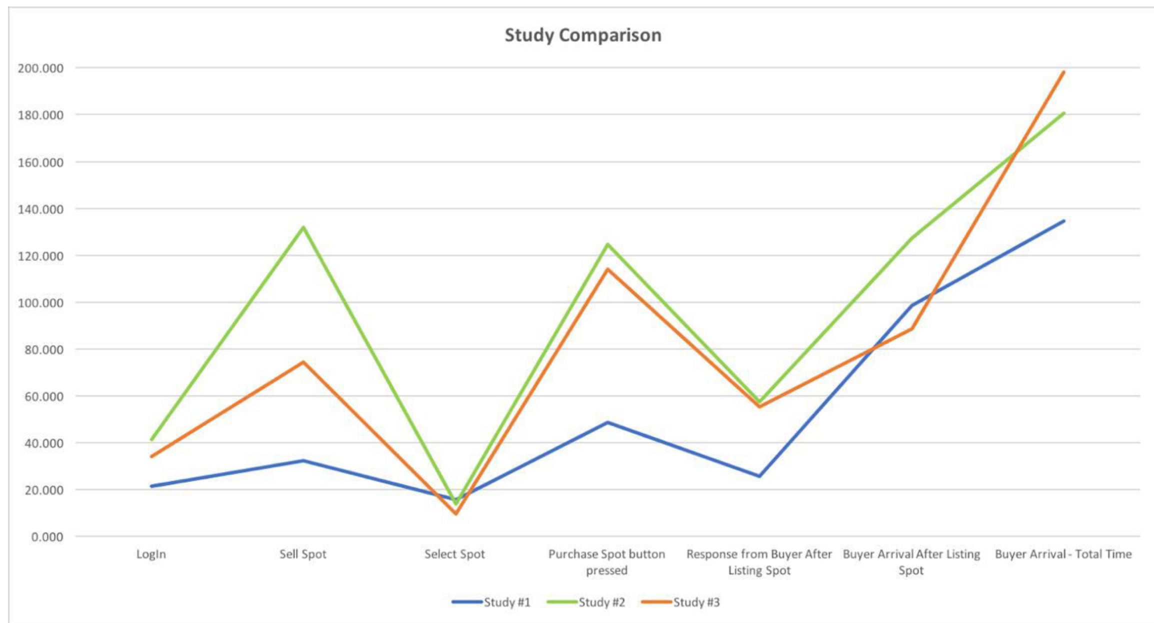

Figure 6.9 Study Comparison

6.3 Performance Testing with JMeter & BlazeMeter

In order to examine the scalability of this application, it was important to set up a framework for testing the performance of the Firebase database, as additional load was put on. For this, JMeter and BlazeMeter were used.

Apache JMeter is an open-source, cross-platform java application that is designed to load test function behavior and measure performance, especially as additional load is placed upon the application [23]. In order to achieve this, it was necessary to setup a script that simulated adding an entry into Firebase's database. The scripts were designed for both Sell_Requests and utilized the REST endpoints for the application's Firebase database, so that identical data could be written to the database to simulate additional sell and buy requests. JMeter's HTTP client was used to communicate with the database via an HTTP POST request. Once the scripts were set up, performance tests were conducted on BlazeMeter.

BlazeMeter allows users to run large-scale load tests in the cloud. It provides a GUI interface that enables users to upload their JMeter scripts, and to run tests on them. BlazeMeter also includes a panel to analyze the results of tests run on their platform. BlazeMeter was used to conduct tests, and the results were analyzed, in terms of number of concurrent users, error percentage, and mean response time (in milliseconds).

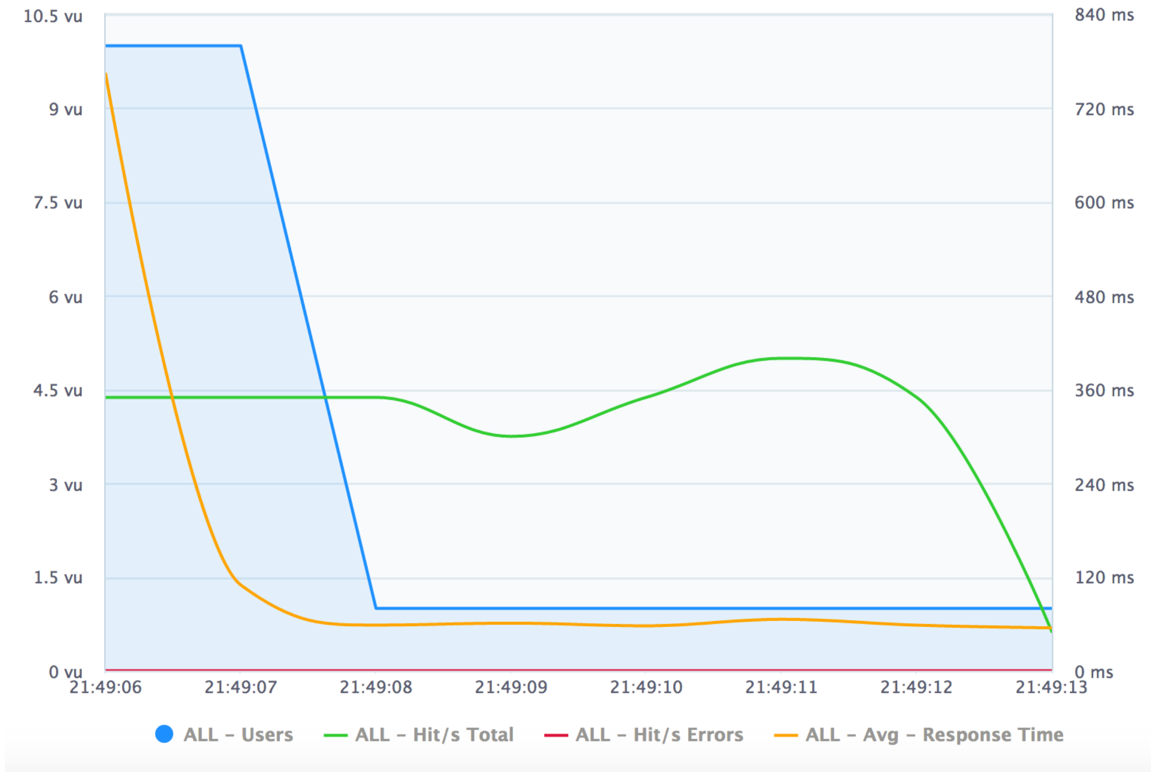

Figure 6.10 BlazeMeter Performance Trial #1

Figure 6.10 shows performance metrics for the first BlazeMeter test trial with a maximum of 10 virtual users. The test was to simulate making continuous calls to the database. For this test, there was a 0% error rate, and the mean response time was 165.74ms (milliseconds). The total number of new nodes created in the database was 500.

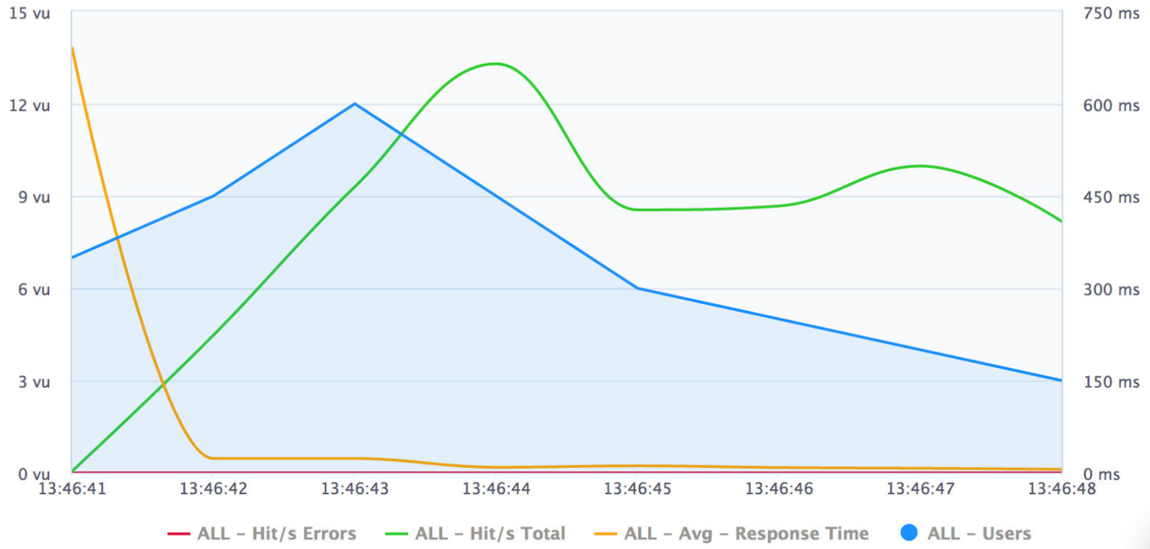

Figure 6.11 BlazeMeter Performance Trial #2

The average response time was 12.19 ms. The average hits/second were 357.14, with a total of 500 calls to the database to create new sell requests. This test also had a 0% error rate.

In addition, testing was done directly with JMeter, utilizing 5, 10, 25 and 50 threads/users. Figures 6.12 and 6.13 illustrate the results of adding more users, or threads, and the resulting response time. Again, as with the BlazeMeter tests, the error rate remained at 0% for all tests.

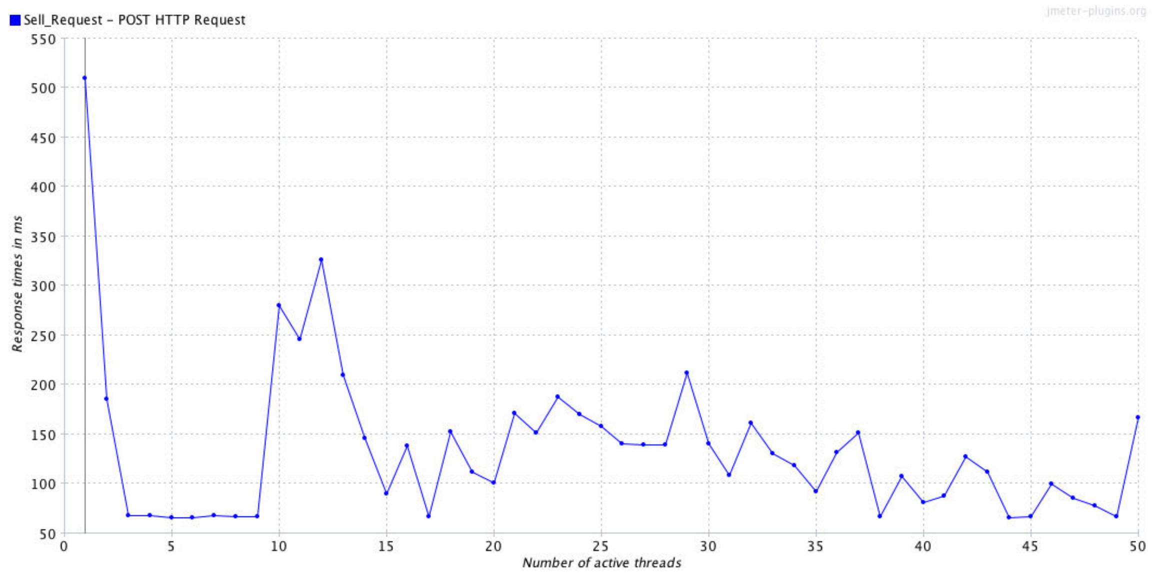

Figure 6.12 JMeter Response Time per Number of Active Threads - Trial #1

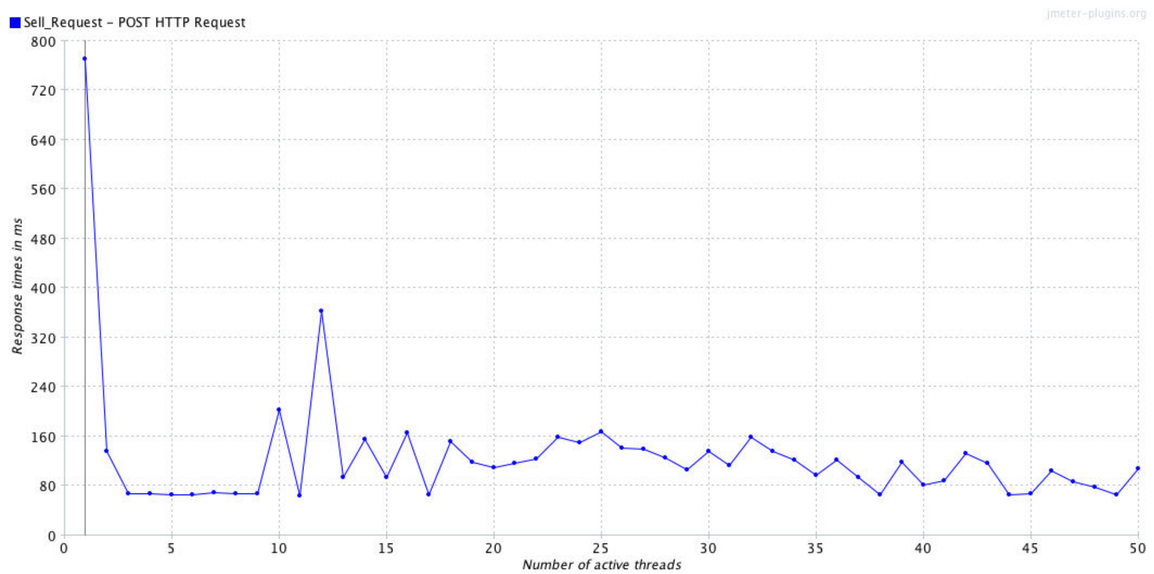

Figure 6.13 JMeter Response Time per Number of Active Threads - Trial #2

In conclusion, all of the JMeter and BlazeMeter trials showed excellent response time, as well as minimal errors (in this case, none). These trials can be even further scaled up in the future to test Firebase's backend capabilities and scalability even more, but

preliminarily, it has been shown that Firebase's backend is very capable and scalable. In addition, this should help to point out that the increase in time taken to perform functions was likely due to users, and not due to the system in place.

Chapter 7: Conclusion and Future Work

The app developed for this thesis has demonstrated how a new business model, based on the shared economy principle, can be used to handle the parking problem that many people face today. This thesis has identified specific use cases where this approach would be advantageous over the existing parking model, where users are sold parking spots from parking garages and other managed parking systems. Of the examples where this app would be more advantageous are universities, busy areas with street parking, or anywhere where parking is difficult to find, where parking garages are not available.

It was shown how a specific set of technologies, including the Swift Programming language and Firebase for backend services (Database, Authorization, etc.), can be used to develop an impactful, real-world application. Specific project challenges, such as the need for constant, accurate location services, designing a single app for buyers and sellers, as well as the need to maintain communication between users were discussed, as well as strategies to mitigate those issues, such as separation of views for buyers and sellers, an MVC-based design, the use of appropriate map and location-based frameworks, and other solutions.

In addition, both real-world simulations and automated testing have been conducted to assess the usability and scalability of this app and its underlying backend system. The user-based testing revealed an increase in total time to use the app and its functions as more users were added, however, the possibility exists that the two-user trial run may have been an outlier, as results seemed to stay consistent with the larger trial sets. Moreover, the automated testing conducted with JMeter and BlazeMeter showed

very fast response times, with no indication of significant performance decreases with added users.

In the future, it is the intent to integrate parking spot API's. The app would still have the main functionality of being able to buy and sell parking spots (i.e. a peer-to-peer marketplace), but would also have the option to look at available parking garages, and reserve/book parking spots. Including both of these functionalities, however, would need to be done seamlessly and intelligently, so as to not over clutter the application, and confuse users. In addition, a timed bidding system can be implemented, where multiple buyers will be able to bid for a parking spot. For the vast majority of parking spot purchases, it will not be necessary, but for those valuable parking spots where multiple buyers exist, it could be a novel way to increase how much sellers earn. Lastly, Push Notifications would be beneficial. When a user needs a parking spot in a certain area and their phone is locked, they would get an alert notifying them if one was made available.

References

1. "Parking congestion to continue unless another structure is built | The ...." 7 Dec. 2015, http://sundial.csun.edu/2015/12/parking-congestion-to-continue-unless-another-structure-is-built/. Accessed 20 Apr. 2017.
2. "Parking problems continue to plague students on and ... - Daily Sundial." 1 Oct. 2014, http://sundial.csun.edu/2014/10/82389/. Accessed 24 Mar. 2017.
3. "UCLA's Parking Permit Crunch is Hitting Students Hard - Curbed LA." 21 Jan. 2016, http://la.curbed.com/2016/1/21/10844766/ucla-parking-permits-students-staff-shortage. Accessed 21 Apr. 2017.
4. "Mastering the not-so-festive dance of Christmas mall parking - MSN.com." 5 Dec. 2016, http://www.msn.com/en-ca/autos/news/mastering-the-not-so-festive-dance-of-christmas-mall-parking/ar-AAl9Bia?li=AA8hc8 Accessed 20 Apr. 2017.
5. "Understanding the Sharing Economy--Drivers and Impediments for ...." http://ieeexplore.ieee.org/document/7427780/. Accessed 23 Apr. 2017.
6. "How the sharing economy can make its ...." http://www.mckinsey.com/business-functions/strategy-and-corporate-finance/our-insights/how-the-sharing-economy-can-make-its-case. Accessed 19 Apr. 2017.
7. "The sharing economy - Oxford Martin School - University of Oxford." http://www.oxfordmartin.ox.ac.uk/downloads/GI_215_e_GesamtPDF_01_high.pdf. Accessed 19 Apr. 2017.
8. "App-Based, On-Demand Ride Services: Comparing Taxi and ...." http://www.its.dot.gov/itspac/dec2014/ridesourcingwhitepaper_nov2014.pdf. Accessed 19 Apr. 2017.
9. "Washio picks up your dirty laundry, dry cleaning with the tap of an app ...." 30 Jan. 2014, http://www.washingtonpost.com/business/capitalbusiness/washio-picks-up-your-dirty-laundry-dry-cleaning-with-the-tap-of-an-app/2014/01/29/08509ae4-8865-11e3-833c-33098f9e5267_story.html Accessed 23 Apr. 2017.
10. "The Rise of the Sharing Economy: Estimating the ... - AMA Journals." 29 Dec. 2016, http://journals.ama.org/doi/abs/10.1509/jmr.15.0204. Accessed 28 Apr. 2017.

11. "Study Explores the Impact of Uber On the Taxi Industry - Forbes." 26 Jan. 2017, http://www.forbes.com/sites/adigaskell/2017/01/26/study-explores-the-impact-of-uber-on-the-taxi-industry/. Accessed 28 Apr. 2017.
12. "Uber and Lyft have devastated L.A.'s taxi industry, city records show ...." 14 Apr. 2016, http://www.latimes.com/local/lanow/la-me-ln-uber-lyft-taxis-la-20160413-story.html. Accessed 28 Apr. 2017.
13. "Service-Oriented Architecture: Scaling the Uber Engineering ...." http://eng.uber.com/soa/. Accessed 5 Jul. 2017.
14. "HTML - Scientific Research Publishing." http://file.scirp.org/Html/3-1560270_66224.htm. Accessed 20 Apr. 2017.
15. "iParking – a real-time parking space monitoring and ... - ScienceDirect." 26 Apr. 2017, http://www.sciencedirect.com/science/article/pii/S2214209616301930. Accessed 3 May. 2017.
16. "Service design for intelligent parking based on theory ... - Science Direct." 18 Nov. 2014, http://www.sciencedirect.com/science/article/pii/S1474034614000974 Accessed 20 Apr. 2017.
17. "City of White Plains Parking App: Case Study of a ... - ResearchGate." http://www.researchgate.net/publication/304287333_City_of_White_Plains_Parking_App_Case_Study_of_a_Smart_City_Web_Application Accessed 20 Apr. 2017.
18. "Internet of Things Approach to Cloud-based Smart Car Parking." 21 Sep. 2016, http://www.sciencedirect.com/science/article/pii/S1877050916321603 Accessed 3 May. 2017.
19. "How SpotHero Works: What to Know Before You Park - SpotHero Blog." 5 Aug. 2016, http://blog.spothero.com/how-spothero-works/ Accessed 5 Jul. 2017."Save money with the free ParkWhiz App for iPhone or Android." http://www.parkwhiz.com/parking-app/ Accessed 20 Apr. 2017.
20. "Save money with the free ParkWhiz App for iPhone or Android." http://www.parkwhiz.com/parking-app/ Accessed 20 Apr. 2017.
21. "Google Analytics Solutions - Marketing Analytics & Measurement ...." http://www.google.com/analytics/ Accessed 28 Jun. 2017.

22. "Apache JMeter - Apache JMeter™." http://jmeter.apache.org/ Accessed 30 Apr. 2017.