M^{arked} An Android Parking Application

Design and Implementation Document

Honours Project – COMP 4905 Carleton University

Saketh Gubbala 100 830 684

Dr. Deugo April 21 2017

TABLE OF CONTENTS

1.0 INTRODUCTION: 3

1.1 Problem 4

1.2 Motivation 5

1.3 Goals: 6

1.4 Objectives: 7

2.0 BACKGROUND: 8

3.0 APPROACH: 9

3.1 System Models: 11

3.1.1 Use case Model: Use Case flow of events 12

3.2 Storage 14

3.3 Mobile Application 17

3.3.1 Dynamic Model: Sequence Diagrams 19

3.4 Architecture 21

5.0 CONCLUSION: 31

6.0 FUTURE IMPROVEMENTS: 32

7.0 References: 34

1.0 INTRODUCTION:

The purpose of this document is to provide development details of how the Marked system will be built. Marked is named as such due to the fact that parking enforcers “mark” a vehicle with coloured chalk to identify if a vehicle has not moved for three hours. As such this system allows for those who park on the street to get notified if their cars have been marked. A car is “marked’ when a noticeable streak of coloured chalk can be seen on the tires. This provides enough time for the user to remove their car before they receive a fine. Once a user has parked their car, they may indicate where on the map(provided by the app) the car is parked. This location is seen as a tag by all other users of the application. If a user notices that a car that is not their own is marked, they may indicate so on the map. This document is split into the following sections: Subsystem Decomposition, Design Strategies, Subsystem Services, and Class Interfaces. Within the Subsystem Decomposition section, the current design will be decomposed into subsystems, and analysed for the design choices made. In the design strategies section we will be examining different software architectures and will explain the one we have chosen. We will also go over how we store persistent data in our application and what challenges we have with our persistent data and the design patterns we are designing in our system. We will be also be analysing the multiple services of each subsystem while using UML diagrams to explain how these services help our system function and how they communicate between the subsystems as they are using the system. This document serves as a reference for anyone who wishes to modify the system.

1.1 Problem

Parking tickets, everybody has gotten one. Whether they've paid them or not, they’ve seen that yellow paper on their windshield. With that said, In a world that revolves around money, we are always trying to come up with new ways to save. That is why I created this app, Marked. Marked includes new ways to communicate with individuals within certain parameters of you to ensure you will not see that ticket on your car. The problem in this situation is that there are currently no apps on the android marketplace that harnesses the community, the collective good of the people around us to help one another. There exists countless applications that provide parking assistance, but none that allows friends or colleagues to help one another prevent parking tickets besides the word of mouth. Consider this for example: At a workplace you notice that your coworker’s car is parked on the street, and you notice further that there is coloured chalk on the tires of his or her car indicating that their car was only recently “marked”. How could you notify them without using word of mouth or texting them? What if this individual was a stranger you have not met, but none the less works at your company? This app attempts to solve this problem.

Many individuals deal with receiving parking tickets. With this app, once you have parked your car you can update your location on the app, and others can see where you are parked. If an individual notices chalk on another person’s vehicle tires, they can notify said person on the app. This allows the owner of that vehicle to go take the chalk off and allow more parking time.

1.2 Motivation

Some people are saving for a house, others a car, and some are expecting a child and need to save for those expenses. And in the midst of their saving, they get slammed with parking tickets. Some people do not have enough disposable that they can give to the city after being served parking ticket after parking ticket. I have seen many friends struggling with student loans who also have several parking tickets that are due. This motivated me to create Marked. If I see a car that has a mark on the tires I wish to let that individual know of an incoming parking ticket, because I too wish that when people witness my car being marked that they help me prevent tickets. Far too often you see people spending their last hard earned dollars that could go to a meal, or a warm winter coat but instead they sacrifice it to pay a parking ticket that could have been avoided. This application does not exist to allow individuals to illegally park wherever they wish, it is simply a way for the community to help one another when no other parking option exists..The inception of this idea began at my previous workplace where colleagues would consistently leave the building every three hours so as to check on their car. These individuals would then let others who have also parked on the side of the street know by word of mouth if their respective cars were tagged or marked. It should be noted that these cars were only parked there due to appropriate parking locations being excessively far away or far too expensive.

1.3 Goals:

The goals I have placed from the beginning and have accomplished include creating a space for communication, creating a tool that can be considered an asset to everyday life, and lastly to help people save money. The first goal, communication, was to create a way for people to communicate with each other in a positive way that encourages helping one another. This was achieved by creating an interactive tool that allows for individuals to connect. By connecting with each other and letting each other know when their tires have been marked. The communication feature encourages friendships amongst the people in an innovative way. With that goal in mind I created this app which allows for communication amongst peers in your area.

The second goal is creating a tool that can be used widely in everyday life. Many people have to deal with parking. Often times there are no parking spaces available and you're stuck parking on the side of the street. This app reduces the hassle of paying those parking tickets and this generation of individuals will surely take advantage of this tool. Our generation is very tech savvy and are always looking for the most convenient apps that will help them with their day to day lives.

Lastly, the goal to help people save money. This is the hottest topic throughout society, finding a way to save money such that it can used for something far more useful or entertaining. By creating this app I have created a way to avoid parking tickets, which in the long run will significantly decrease the amount of wasted disposable income. In 2017 in Ottawa alone, there has already been $2.3 million due in outstanding parking fines. Since android has a nearly 76% of the market share on mobile devices, it so seems fit to develop an android app to reach as large as a target audience as possible

1.4 Objectives:

The main objectives required of the Marked application is to create a platform where the users can notify one another if a car is “tagged”. To achieve this, the user must be able to securely login. Thus a secure user authentication system with an SQL backend is required. The user must also be able to create a profile where they may add or edit details of their account including details of their vehicle. This application must recognize their location on a map, and allow them to “mark” that location as the point where their car is parked. This mark must also somehow include basic car info. I.e. licence plate or brand/model. As such, for this application to have a map, it must incorporate Google Maps using Google API. The user must also be able tag other cars as “marked”, and finally the users that have their cars alerted must be able receive notifications.

2.0 BACKGROUND:

There exists a large variety of parking applications on the Android Play Store. Many of the apps offer something unique to the user that the others do not. Do you wish to find parking locations for disabled people? There is an app for that (Street Parking Melbourne). Do you wish to pay for parking without fumbling for change? There is app for that (Green P). There also an application that allows you to park your car on driveways of property owners that wish to earn a little income on the side. However, there are simply no apps that take advantage of a passerby that has witnessed a car with coloured chalk on the tires. There are countless apps that simply implement an alarm for a typical three hour time limit, and there are a countless more that saturate an app with inconsequential features. Marked attempts to go beyond gimmicks by offering a simple and useful app for an intended purpose.

At the basic level, Marked takes advantage of the Google Maps API. An API or Application Programming Interface allows one to access Google services such as Google Cloud Messaging, AdSense, Translate and many more. Maps is one such service that is offered in both free and paid options. With it being a largely free service, this has led to developers incorporating it into millions of apps. Such a large user base has further led to the constant improvement of this service on both the development and user end. One such example of improvement is the categorizing of Location Services as a dangerous permission. Prior to API 23, developers could easily attain the user location without concern, but after much privacy concerns by the general public, it is now required that users be asked for access to their location. This is but a small example of Google services consistently improves. Over the past decade the developer tools provided by google has resulted in the use of location services to determine traffic conditions, the use of satellite imagery for more accurate map views, and now street views can be used extract traffic restrictions and read information boards. All of the mentioned uses may one day be implemented in the Marked app to make it more robust and useful for the general user. For now though, a more basic implementation of google maps will be used develop the the core functionalities.

3.0 APPROACH:

The first step in creating a solution for our problem is to define the criteria or requirements that the application must satisfy. These requirements are detailed below.

Functional requirements are a form of high level design,and in this case describes the core behavior of Marked. In general, it is a description of what Marked must do to be considered a parking system, and typically involves features such as placing a marker on a map, editing one’s profile, and even running an algorithm.

Functional Requirements:

F-01 Users must be able to edit their profile **F-01-01** Users must be able to edit their contact information **F-01-02** Users must be able to edit their vehicle information
F-02 Users must be able to edit location of their car **F-02-01** Users must be able to use location services to “park” their car **F-02-02** Users must be able to manually adjust location of parked car
F-03 Users must be able to alert owners **F-03-01** Users must be able to send alerts that a car has been marked

While functional requirements detail the main functionality and use of the app, non-functional requirements are requirements that are not necessary to the function of the system but involve criteria that is used to judge the quality or effectiveness of the program. In other words, these requirements place constraints on how well the program satisfies its functionality and are typically measurable performance or security features of the system.

Non Functional Requirements:

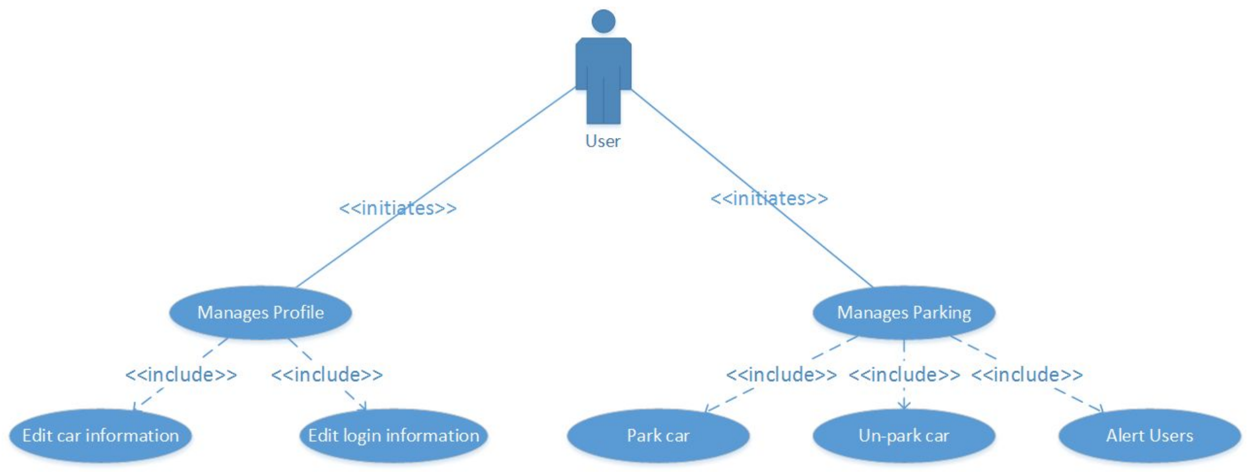
NF-01 - Must be written in Java
NF-02 - Must use Android Studio
NF-03 - Must be portable (Cross-platform) (Maybe)
NF-04 - Must be modular (Ability to swap out GUI platform) (Maybe)
NF-05 - The GUI has to be responsive
NF-06 - Must store the data in an SQLITE database

3.1 System Models:

System models allow one to conceptually visualise the system in terms of how it is implemented, and provides a clear and more concise way to represent input and output at each step. Although this may belong in a design document, it will help understand why the application is built the way it is later on. In this case, system models are comprised of the following: Use Case, Object, and Dynamic models. In this report, dynamic models can be expanded further into sequence diagrams which model the flow of logic within the Marked system.

The Use Case Model below describes how the user can interact with the system to reach a certain goal. The model consists of a set of use cases, an actor and how they relate. Each use case in the model describes the sequences or steps that an Actor should take in order to reach the intended goal.

Use Case model for user:



Use Case Descriptions for User

UC-01	**Edit car information**	The user edits details of their car in profile.
UC-02	**Edit login information**	The user edits password and email in profile.
UC-03	**Park car**	The user parks a car associated with them.
UC-04	**Un-park car**	The user un-parks their previously parked car.
UC-05	**Alert user**	The user notifies another user that their car has been marked.

3.1.1 Use case Model: Use Case flow of events

Use Case Identifier	UC-01
Name	Edit car information
Participating Actors	Initiated By User
Flow of Events	1. The User selects the Profile option. 2. The program provides a list of all editable personal and car details. 3. The User selects the car details to edit. 4. The program provides any existing details about the car and the option to edit it.
Entry Conditions	The User must be logged in.
Exit Conditions	A new car object associated to the user is created if there already isn't one.
Quality Requirements	
Traceability	F-01

Use Case Identifier	UC-02
Name	Edit login information
Participating Actors	Initiated by User
Flow of Events	1. The User selects the Profile option. 2. The program provides a list of all editable personal and car details. 3. The User selects the personal details to edit. 4. The program provides any existing details about the car and the option to edit it.
Entry Conditions	The User must be logged in.
Exit Conditions	The profile properties are changed and saved.
Quality Requirements	
Traceability	F-01

Use Case Identifier	UC-03
Name	Park car
Participating Actors	Initiated By User
Flow of Events	1. The program identifies the user’s location. 2. The User selects the Park option. 3. The program then places a marker at the user’s current location.
Entry Conditions	The User must be logged in.
Exit Conditions	The user’s parking location is saved.
Quality Requirements	
Traceability	F-02

Use Case Identifier	UC-04
Name	Un-park car
Participating Actors	Initiated by User
Flow of Events	1. The User selects the Un-park option. 2. The program removes the marker from the previously parked location. 3. Any warnings regarding parking are removed.
Entry Conditions	The User must be logged in.

Exit Conditions	The user's parking location is set to null in database.
Quality Requirements	
Traceability	F-02

Use Case Identifier	UC-05
Name	Alert user
Participating Actors	Initiated by User
Flow of Events	1. The User clicks on a marker. 2. An infowindow displays general information of the car parked in that location along with an "Alert" option. 3. The User clicks on the Alert option if the car has been "marked" by a parking enforcement officer.
Entry Conditions	There must be at least be one parked car.
Exit Conditions	A notification is sent to user associated with the parked car.
Quality Requirements	
Traceability	F-03

3.2 Storage

Once the decision to implement the Marked system as an android application had been made, it was then decided that determine the details of how information is going to be stored was more crucial than designing the UI. Despite prior experience working with a SQL server which allows for retrieval of information across a network, it was decided that a local storage system on the phone implemented by SQLite will be sufficient for the time being. The idea with this reasoning being that if Marked is implemented in a modular fashion, in that if the UI, the logic layer (middleware), and the database are developed as separate subsystems, it would then be possible to take advantage of remote servers offered by amazon web services or Google cloud storage at a later date. The main concern being the the development and implementation of the core functionalities.

Since I had decided to store the persistent data as a relational SQLite database, each entity object that would be required in the program had to be determined as well as their relationship with other entity objects. The database essentially holds two objects: users and cars, each with their own tables, with an additional table essentially describing the relationship. It was important that a proper choice of primary keys be used to prevent duplicates. The entity objects as they appear in the database is detailed below.

User: The User object easily maps into its own table with the following attributes:

- Username: This is the primary key. it allows us to distinguish one user from another.
- Name and email string attributes.
- A relationship of many to many between users and cars is described as a separate table (cars_owned_by). This table just consists of a primary key with two other attributes: Username and Car Id (CID), which are foreign keys belonging to the User and Car tables respectively.

Car: Cars also map easily into their own tables.

- CID: An int attribute that is the primary key of the project.

- Make: Represented by a string attribute is the brand of the car such as Ford, Mercedes, mazda, etc...
- Model: The model of the car allows users to identify and further narrow down the car described on the app as they pass by. It is also represented by a string value.
- Colour: Like the attributes above, this is also a string. Together with the make and model, this trio of attributes allow for a quick visual verification when alerting a user.
- Licence_plate: If cars parked nearby are far too similar to differentiate with a quick glance, the licence plate is the final check a user would have to make. However, as the licence plate number can be deemed to0 personal an information, the user may refrain from including this on the app. Since most vehicle can be identified with just the colour, make and model, this piece of information is useful, but not absolutely necessary.
- The owners who possess the vehicle are described by the relationship in the Owned_by table in the database (see User).

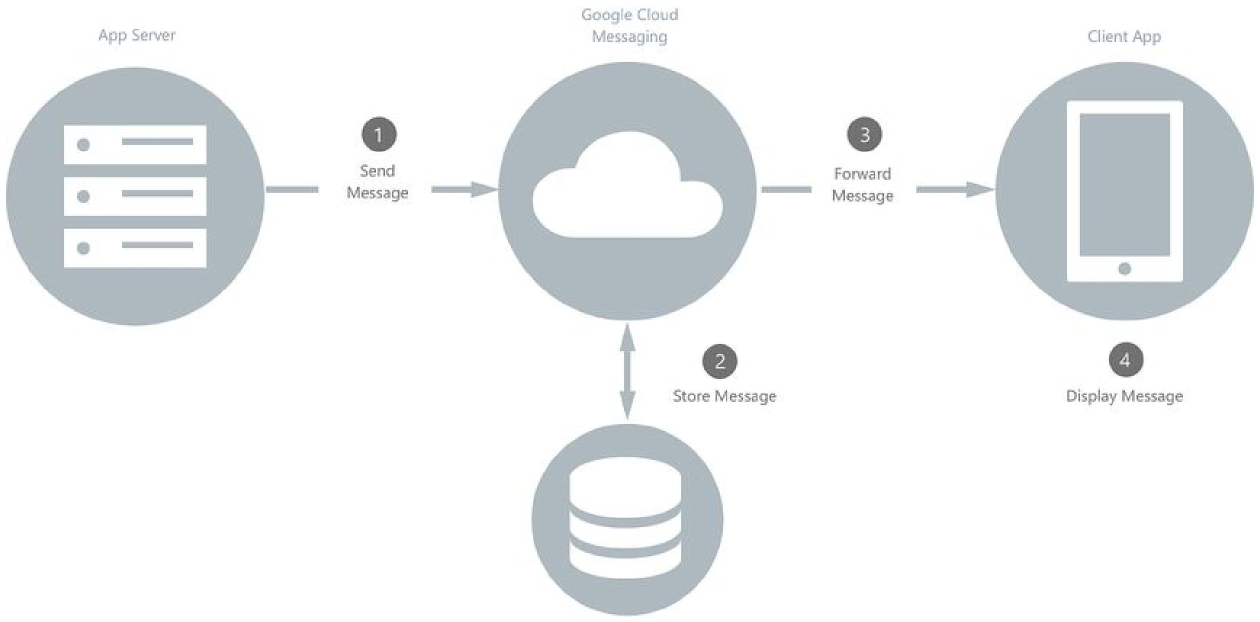
Once the objects, and the details of their relationship were fleshed out, the criteria that “Marked” needed to possess in order attain the above information from a user, and subsequently allow for a seamless experience from logging to finally alerting another user if they witness a parking enforcement officer tag their car.

3.3 Mobile Application

As an android application, Marked could have been written in a variety of IDEs from Eclipse to NetBeans, however Android Studio is a superior platform to build an android application as it was specifically developed for such a purpose. Its integration of gradle, and stability makes it an invaluable tool, especially for a beginner in android app development. It is also appreciable that Android Studio provides an easy way to build user interfaces by simply dragging and dropping elements onto the layout.

The most important feature of this application is to be able to park one's car. As such this involves using google maps as the base for which the rest of the features are built around. Google has ensure that it is relatively easy to implement the maps into one's app. Since a large portion of the android development community utilizes the maps API such that an activity is offered especially for such a purpose. This is especially convenient because the maps service is provided for free given that when the app does get published the Google logo is clearly visible. It is however important to ensure that internet permissions are included in the manifest for the maps to work properly.

It's not just important to park one's car, it is equally important to be notified if and when a car has been tagged by an parking enforcement official. As such another important feature is the notification system. Previously, a developer could take advantage of the Google Cloud Messaging system or GCM for short. Such a service enables to send push notifications to a device to inform the user about a myriad of warning or just general information. Now however, GCM has been deprecated, and Google has released a new system named FCM where the F stands for Firebase. Firebase is a company acquired by Google in 2014 that offers a suite of features and services from realtime databases, and storage to cloud messaging. In essence, FCM retains the infrastructure of GCM but developers no longer need to write registration logic. And as we are only required to send push notifications, we will only focus on one key feature of FCM here, downstream messaging. It important to remember however, that each client device is provided a key upon registering with FCM.

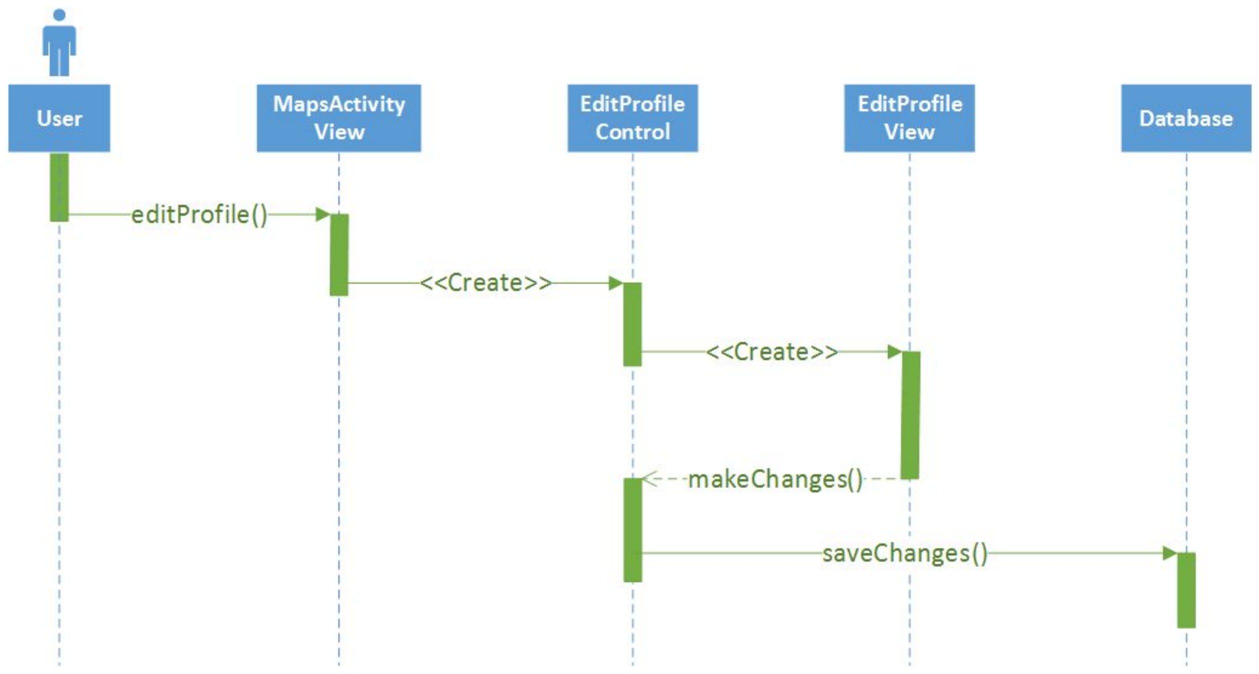


The above image shows the architecture of GCM and also describes how downstream messages are processed. When an application wishes to send a push notification, a message along with the token identifying the target client device is sent from the app server to the Google Cloud Messaging server where the message is stored and forwarded to the client device. Since we don’t have a server in our case, the token and message are sent from the local storage on the device wishing to send the alert. It should be noted that FCM has a significant advantage over many other systems including Apple’s own for push notifications. The advantage being that the FCM can deliver messages to any client app. The keyword being “any”. This means FCM can send messages regardless of platform, whether the client app is android, ios, or even windows. This will prove to be especially useful if I choose to release the same application on IOS.

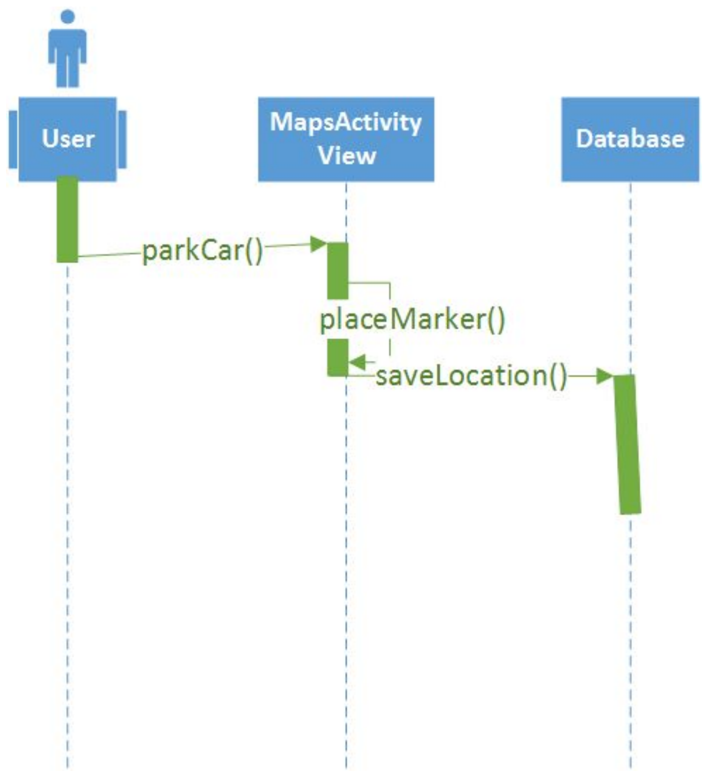
3.3.1 Dynamic Model: Sequence Diagrams

Sequence diagrams are meant to show the flow that pertains to a specific use case. The only relevant direction in the diagrams is down, which denotes time (or the sequence of actions). Basically, something that happens above something else happened before. Sequence diagram shows the actor involved, as well as the control and view objects. Dashed vertical lines show the life of an object, and an ‘X’ shows when it is being destroyed. Green boxes appear along the vertical lines when an objective becomes active and interacts with another object. Arrows show communication between two objects.

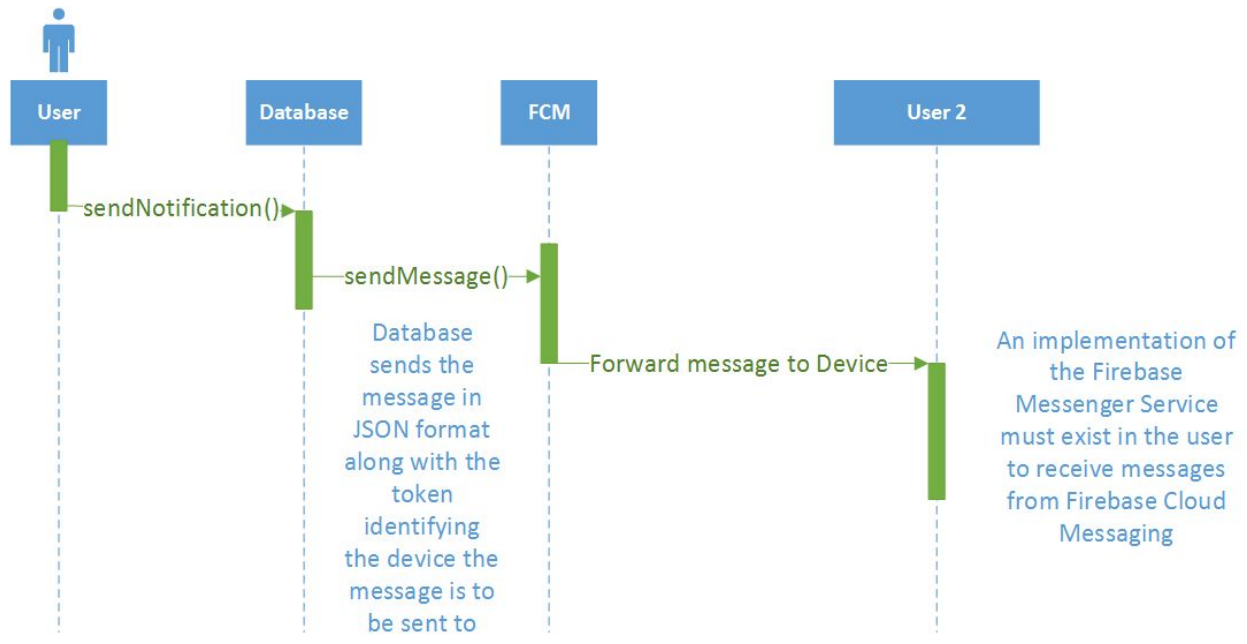
SEQ-01 - Sequence diagram for Edit Profile Use Case



SEQ-02 - Sequence diagram for Park Car Use Case

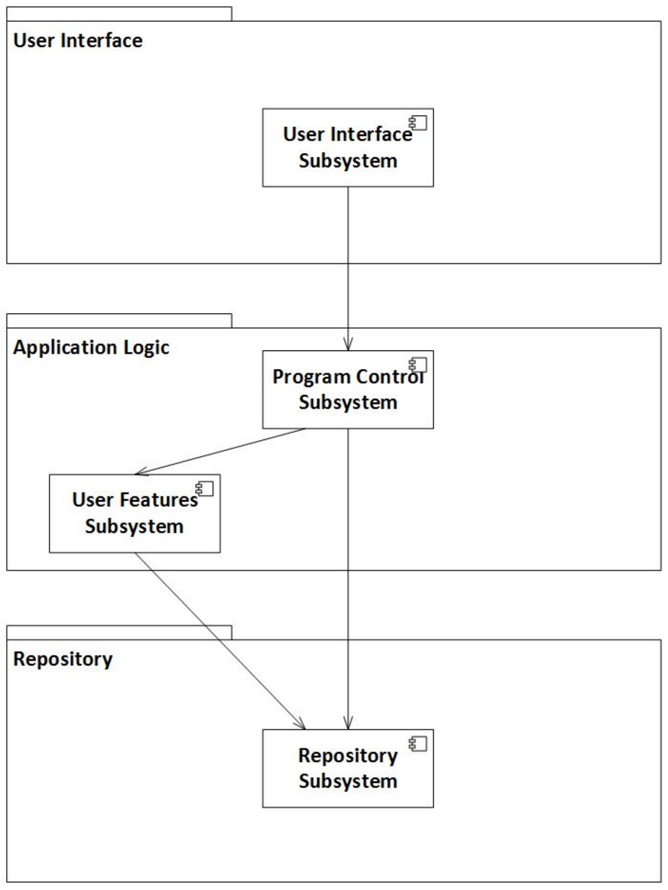


SEQ-04 - Sequence diagram for Alert User Use Case


User
Database
FCM
User 2
sendNotification()
sendMessage()
Database sends the message in JSON format along with the token identifying the device the message is to be sent to
Forward message to Device
An implementation of the Firebase Messenger Service must exist in the user to receive messages from Firebase Cloud Messaging

3.4 Architecture

The Architectural Style used is 3-tier. At first I was considering a repository style as it seemed like a good fit, but I have found the application to be more user driven than data driven since changes that are happening in the application is all based on the user input and not driven by the changes in the data. The system is driven when a user presses a button or does an action in the user interface. Additionally I found that three tier was superior to other styles in this case since it keeps User Interface, application logic and storage all separate from each other and makes the system easy to maintain and understand the system, this would benefit the system for future enhancement since it allows for easy scaling of the storage or even implementing a different storage system without having to touch any of the the application logic nor the User Interface , the same goes for the User Interface you could potentially swap out the User Interface with a different one without having to touch any of the application logic nor the storage. Furthermore this also allows for the ability of being able to split the development for different developers to be worked on concurrently which would allow for faster and efficient development since each group of developer(s) would only be focusing on the tier they are assigned.



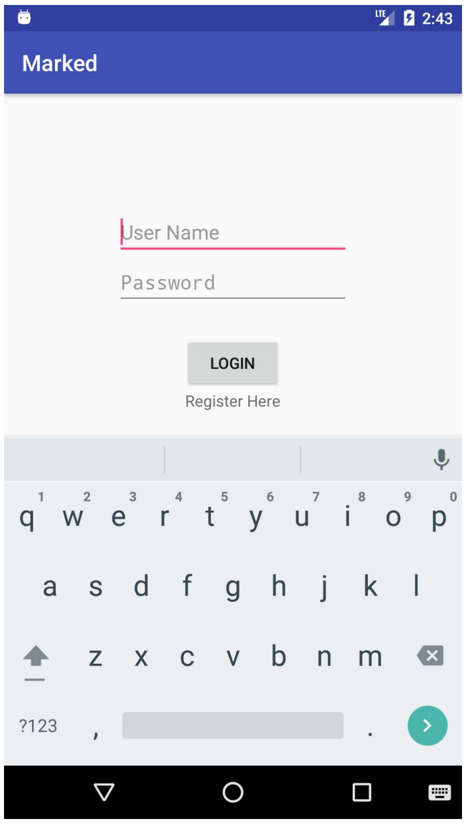
This three tier diagram effectively displays how the different subsystems are organized into each tier. The Marked system is decomposed by grouping sets of related classes into subsystems. Here the User Interface Subsystem contains the whole user interface as one subsystem which promotes cohesion as the user interface changes and talks to itself and promotes loose coupling as we limit the amount of communication from the subsystem to the other subsystems. The User Interface Subsystem contains all the layouts used within the application.

The Program Control Subsystem handles the main control of Marked and also handles the access to different features in the program. The classes within this subsystem are Maps_activity, Login_activity, and Register_activity.

The User Features Subsystem are grouped based on the features that are offered to the user. As such, the classes grouped within this subsystem are Car, EditProfile, and User. And finally the Repository Subsystem simply consists of the DBHandler java class.

4.0 RESULTS AND VALIDATION:

The culmination of four months of planning, coding, and using the approach detailed in the above section has resulted in the app aptly name Marked. As the application has not yet been published on Google Play, one has to build the files using gradle, and run the application on an emulator. The emulator device chosen for displaying the results is . However, once published the application should work just as well on any android device.

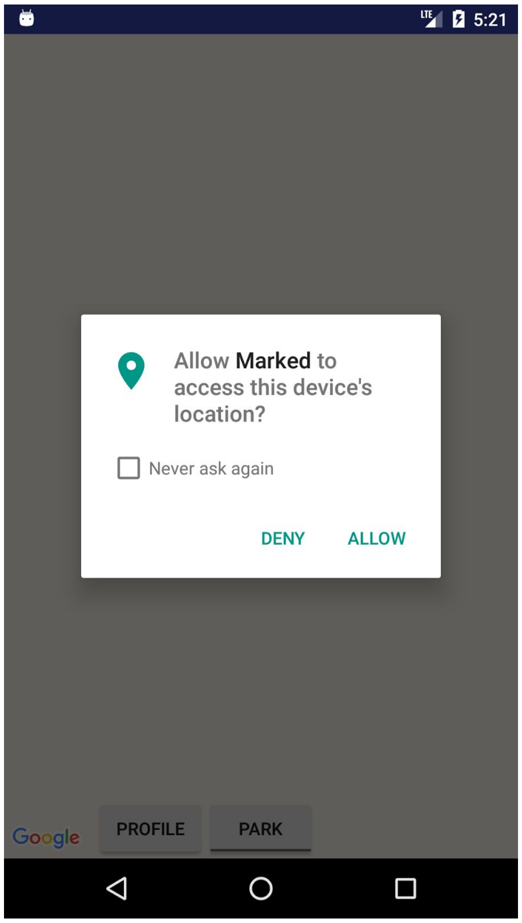


Upon the first installation and running of the app, the user is greeted with a login page with text fields reserved for the username and password and a

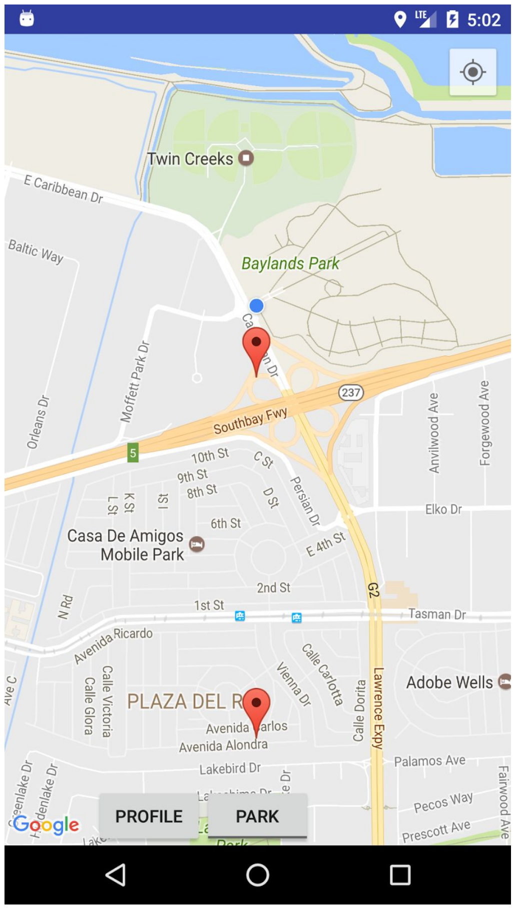
“Login” button. If the user chooses to login by entering arbitrary values as can be shown in image[ ], the program performs an internal check matching the given username and password. If the entered values do not match or even simply do not exist, the application will display a message notifying the user as such.

If the user does not posses any login credentials and does wish to register, he or she must click on the “Register Here” text visible below the textfields. Once the user does wish to register, they are led to the registrations page where they must enter their full name, the desired username, password, as well as an email address. Although the program does not currently use the provided email address for any purpose, it may be used in the future for validation. In the future I may also allow the user to login via their google account, but for demonstration purposes I wanted to refrain from using Google provided conveniences and perform any checks within the program for now. After the user has filled in the appropriate details and presses on the “Register” button, they are led back to the login page where the newly created credentials will be accepted. When the register button is pressed, the program once again performs a check determining if the user already exists within the system. If the user does already exist, a message will be displayed notifying the user to try again with another username. If no such username exists within the system, a new user object is created. In typical cases, once a user has registered, they are then placed in a logged in state, however to demonstrate that the username and password check within the login page does perform correctly, the user is asked to formally login.

Upon successfully logging in, the program requests the user to grant it permission to access their location services. Prior to to API 23 of android, developers could essentially use the user's location without necessarily asking for permission. But after concerns were raised that such services infringes on one's privacy, Google has deemed that on API 23 and greater locations services cannot be used without the user's agreement. Similar to how one would use internet permissions, the location services permission must be placed in the manifest to take advantage of it. The manifest exists in the root folder of the application, as the name suggests, holds comprehensive details and information regarding the Marked system. It provides a unique identifier for the app, describes all the components that exist within it, and declares all the permissions required in the app. If allowed, the application only requests this permission of the user once, and will remember this setting until it is uninstalled. If however, the user denies this request, the user is led back to the login page. If the user attempts to login again, the request will once again be displayed. This application is centered around the user's location, as such the user will not be able to login and this cycle will be repeated until the user selects the “allow” button.

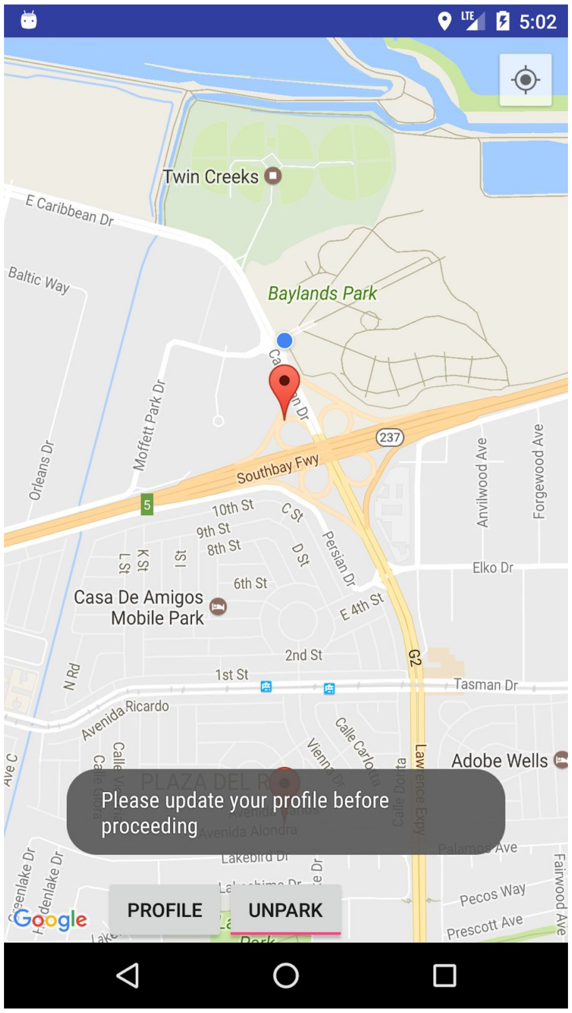


Once the request has been granted, the user is displayed the Google maps with the map centered around Sydney, Australia. However, upon clicking the location button on the top right hand corner of the application, the map will begin to move and will eventually center around the user's location, with the blue circle indicating the exact location with an error of a few meters. The markers currently on the map indicate places where other users have parked their car. My original intention was to only allow the users to park their car on land, however that involved a greater level of difficulty and

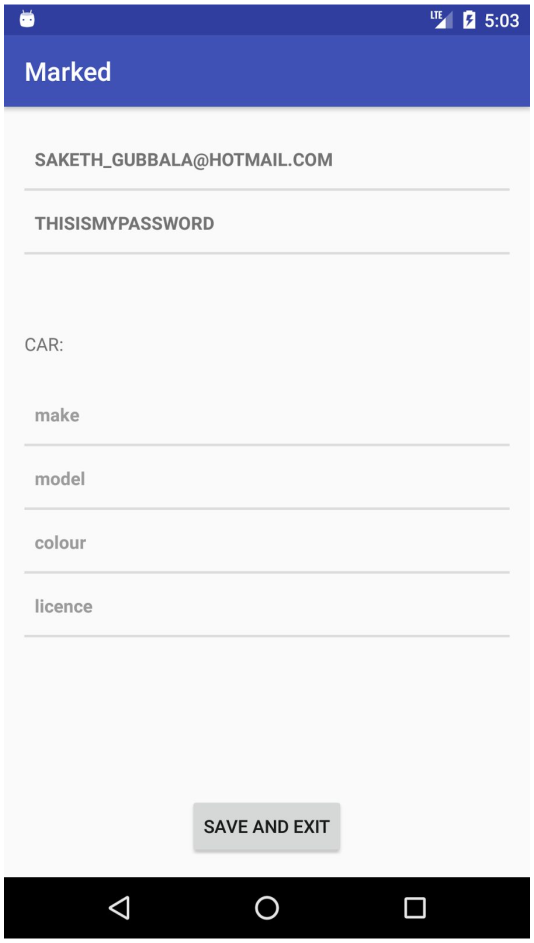


requires much more time spend on the application. This feature will be further investigated and I hope implement it in the near future.

Since the map is now completely loaded, and there are markers visible nearby, the user may believe that they are ready to mark the location where they placed their car by simply clicking on the "PARK" button. Although it is easily possible to place a marker at one's location, in our scenario the marker is associated with the location of a car. Hence, if the user clicks the park button, they will be displayed to message to first update their profile. This is because there simply isn't a car object associated with user, so a person cannot park a car that simply doesn't exist! To create a car object in this application, the user must first click on the Profile button which will lead to a page that will allow them to verify and edit the details of the user profile. The user profile contains not only the details pertaining to their login, such as email address, it also include information about their car such as make, model, colour, and licence plate. This information can be verified at any time by simply clicking on the profile button. It is also possible to edit the details displayed by clicking on each textview. These textviews themselves cannot be edited, however a click listener is placed on each of these views such that an alert dialog is displayed when the textview is clicked. Each of these alert dialogs contain an editable text field that reflect the contents of the original text view. After the field has been edited, the “ok” button may be clicked to return back to the original text view which now reflects the contents of the edited text field. It should be noted that if the user were to return to the map view without clicking the “Save and Exit” button, none of the details that were previously edited will actually be saved. When the email and username fields are modified and saved, the respective values within the user table of the database is altered, thus changing the way the user now


Please update your profile before proceeding

logs in. On the other hand if the user adds details regarding the car that previously didn’t exists, a new car object is created and placed within the “CAR” table. This

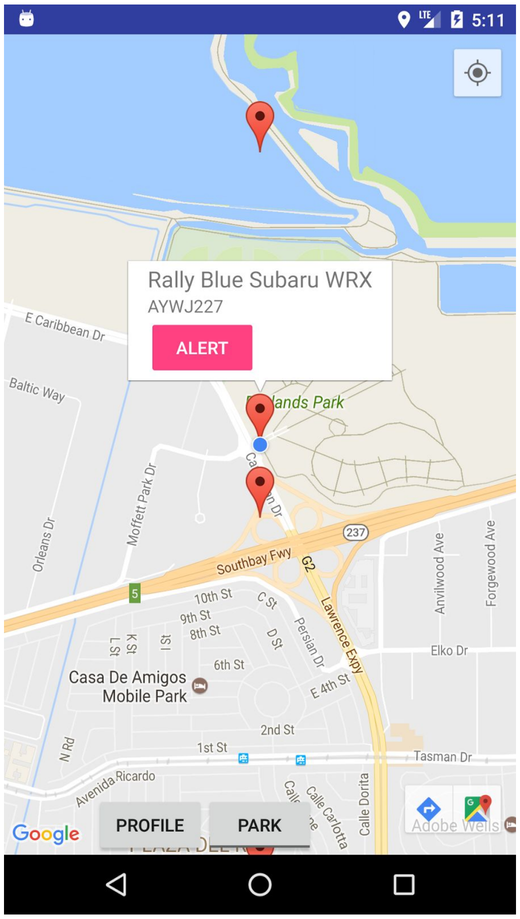


however does not mean that a car is associated with a person. For a relation between a car and a user to exist, both the username and car id need to be placed within a new database table as foreign keys. And this is exactly what is done when the save button is pressed. This many-to-many relation table potentially allows for a user to own multiple cars, and a car to be owned by multiple people, although the option to add another car has not yet been implemented.

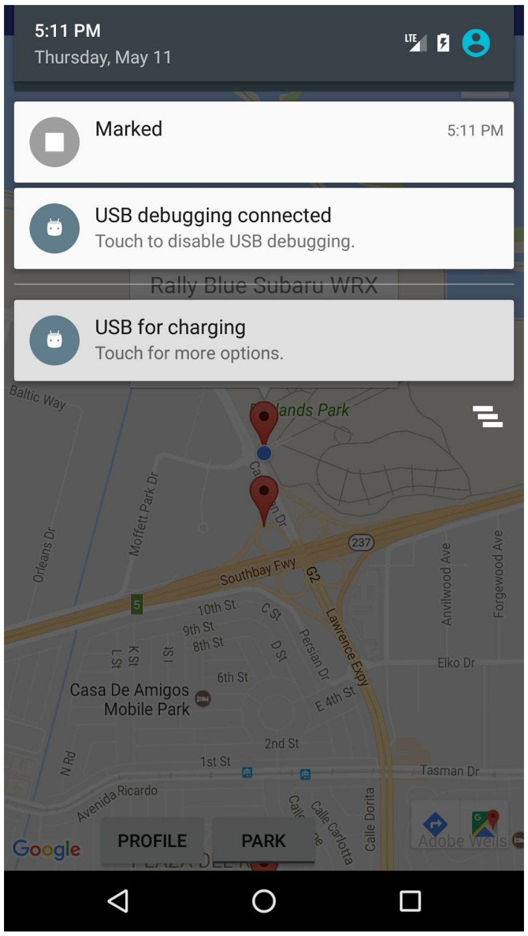
If on the other hand, a car object associated with the user already exists, the appropriate details are simply altered within the database, and a new car object is not created. It should also be noted that only the make, model, and colour fields are mandatory for the parking of a car, as the user may not be comfortable divulging their licence plate number.

Now, the user should have no problem parking a car by simply clicking the park button. Once the button is pressed, the longitude and latitude associated with that location is inserted as a LatLng value in the car object, and subsequently the car associated with that marker is placed within a hashmap using the marker and car as a key - value pair.

It should be understood that only car that are currently parked are stored in this hashmap, this allows us to place every car that is parked on the map as soon as Google maps are loaded on login. When the user clicks on a marker, an infowindow is displayed containing the car information that a user previously added on the profile page. An infowindow is the pop-up that is displayed over the map when a marker is clicked. A typical infowindow is not a live view, in that it is simply an image rendered over the map. Therefore a typical infowindow does not allow a user to interact with a button as I had wished. Any buttons that are placed within it will not recognize touches or clicks as android will simply dismiss all interactions within it. Infact, Google recommends that developers not place any interactive components within these windows despite Google implementing the very same in their google maps application. A lesson in “do as I say, not as I do” it seems. However the entire infowindow itself can act as a button so that a click anywhere on window can be used to trigger an event. But this was not what I wished, I did not want the user accidentally clicking anywhere on the window and alerting another user for no reason. Hence I created a wrapper around the map fragment in the form of a layout, and created a custom infowindow that will override the



touch events of the previous layout. This infowindow holds two text fields that displays the make, model, and colour on one line as the title, and the licence plate on another line. Below these text fields lies a button that will be used to send alerts. This is in fact an adaptation of a solution provided by the developer of android application name CG Transit [ ], a public transit application used in the Czech Republic. This allows to place a button within the infowindow whose touch events will no longer be disregarded by the android application.


5:11 PM
Thursday, May 11

Marked 5:11 PM

USB debugging connected
Touch to disable USB debugging.

Rally Blue Subaru WRX

USB for charging
Touch for more options.

The alert button, the feature that makes this application unique from all other parking application available on the google marketplace. The user should only consider clicking this button when they observe that a nearby car is “marked”. Once this button is pressed, a thank you message is displayed to the user indicating that a notification has been sent. Subsequently a message containing the target user’s car information is sent to the database wherein their token is obtained and a new message in a JSON format is formed. This JSON message is then sent to the Firebase Cloud Messaging server where the information is processed and forwarded to the client device. This information is then shown as a notification since Marked implements the Firebase Messaging

Service which receives the downstream message and displays it to the user. It should be noted that FCM sends the message based on the token and client Id to the appropriate device where the application is being run and not specifically to a user.

5.0 CONCLUSION:

The main goals set out in the beginning of the document and the beginning of the semester were mostly accomplished. I have demonstrated that I could build an android application that a normal consumer would deem useful. The goal of completing a unique app that does not exist currently on the Google marketplace is mostly complete. Mostly because it is not yet ready to be published on the play store. With some fine tuning of the user interface along with swapping out the local backend with a real time database such as the Firebase database with the same architecture would allow me successfully launch this app. My objective of building a secure user authentication with an SQL backend has also been sufficiently accomplished albeit my desire to build an even more secure system by using email authentication or even Google's own login system. Furthermore, as set out in my objectives, I was able to integrate Google maps into the application by using Google's API, allow users to clearly mark their location by use of a marker, and finally "tag" someone's car if it has been marked by a parking enforcement officer. After testing the application with a few colleagues, and friends, it was found that the application responded fairly quickly to all inputs, and also found that the interface was rather intuitive. However, a large number have indicated that an alarm that is set off after a three hour period that a car is allowed to typically park on the streets would be advantageous. The past few months working on this application has been extremely intriguing, and has been a significant learning experience that will allow me build better and more robust applications in the future. In conclusion the application has been found to successfully accomplish all goals set out, and has been found to be largely useful.

6.0 FUTURE IMPROVEMENTS:

An application is never quite done. As I began to work on it over the past few months, there were several additions that I wanted to make given enough time. One of those additions is the implementation of an early detection algorithm that would consider the patterns of alerts sent out in nearby locations to warn a user ahead of time about a possible ticket. For the time being, only a crude implementation has been made thus far that takes into the latitude and longitude such that if a car is alerted, all other other cars within one hundred metres of that car are also alerted. After further testing, one individual has recognized that a way to communicate with others would be useful in a scenario where people consistently park around a location such as a school or workplace. This would allow individuals who pass by each other’s cars everyday to more effectively communicate and avoid parking tickets. Two other improvements that this application sorely needs have been mentioned throughout the report. One being the implementation of a real time database instead of the location storage used in the current implementation. This one critical part had not yet been implemented due to my wish to focus on completing the objectives set forth early on. Even without the real time database, it was possible to demonstrate the functionalities of the application, thus I deemed it secondary and therefore wanted to focus on this aspect once all other had been successfully completed. The second is the addition of a timer that would count down from the moment the car has been parked. This would allow the user to be reminded in case there is a lapse in attention and there are no nearby passers-by who could alert the user. One last concern that had crossed my mind from the inception of the idea was the credibility of each user. Since a user could alert any other car in a vicinity, how do I ensure that there are no false alerts, and how do I prevent users who have previously sent false alerts from sending even more? One possible solution would be to implement a reputation system similar to that of stackoverflow where user can be up or downvoted. This would in return mean that when an alert is sent out, the user receiving a notification must also know who exactly alerted them.

7.0 References:

[1]http://stackoverflow.com/questions/14123243/google-maps-android-api-v2-interactive-infowindow-like-in-original-android-go?noredirect=1&lq=1

[2]https://developer.xamarin.com/guides/android/application_fundamentals/notifications/google-cloud-messaging/

[3]https://dzone.com/articles/why-android-studio-better

[4]https://firebase.google.com/docs/cloud-messaging/android/receive

[5]https://www.tutorialspoint.com/android/android_push_notification.htm