LetsGo iOS app
Organization: Carleton University
Course: COMP4905 – Honors Project
By: Abdulbadi Sabir
Supervisor: Louis Nel
Date: December 15, 2017
Abstract

‘LetsGo’ is an IOS app where people can either create events, setting the time, location, description, limitations (age, gender…) or they can join nearby events which are created by other users using the App. Examples of such events can include a soccer match, for instance if a group of 16 individuals need to find another 6 in order to make it an 11 vs 11 match up, or a board game, where 1 person can create an event of size 4 to play a board game that requires 4 people, or any other type of outing/activity where the user is short on people/is looking for others that share their interests and are willing to participate in that activity. Each user is assigned a rating which reflects their punctuality and sociability.
Acknowledgments

I would like to thank my family and friends, whose negative and positive feedback provided the sustaining motivation required to complete this project. I would also like to thank all the tech giants out there, from whose shoulders everything came into perspective. I would also like to thank the Google support team, whose hard work provided a seemingly endless stream of available and accessible information. Finally, the Carleton University Computer Science department and my supervisor for giving me the chance to pursue this project.
# Table of Contents

1. Motivation 5
   1.1 The Problem
   1.2 The App
   1.3 User Punctuality
   1.4 Event Customizability
   1.5 User Safety
   1.6 Application’s Competitive Differentiator
   1.7 Interface Aesthetics and Usability

2. Methodology 12
   2.1 App to Project
   2.2 3-Tier Architecture
   2.3 User Interface Tier
   2.4 Functional Logic Process Tier
   2.5 Data Storage Tier
   2.6 Implementation Process

3. Technical Aspects 16
   3.1 Classes
   3.2 The Codable Protocol
   3.3 Encoding and Decoding Dates in Swift
   3.4 Uploading Images in 3-Tier
   3.5 View Controllers
   3.6 3Tiers, 2 Databases

4. Results 29
   4.1 Implemented Functionality
   4.2 Further Work and Conclusion
<table>
<thead>
<tr>
<th>Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>11</td>
</tr>
<tr>
<td>Figure 2</td>
<td>16</td>
</tr>
<tr>
<td>Figure 3</td>
<td>23</td>
</tr>
<tr>
<td>Figure 4</td>
<td>24</td>
</tr>
<tr>
<td>Figure 5</td>
<td>27</td>
</tr>
<tr>
<td>Figure 6</td>
<td>29</td>
</tr>
<tr>
<td>Figure 7</td>
<td>31</td>
</tr>
<tr>
<td>Figure 8</td>
<td>33</td>
</tr>
<tr>
<td>Figure 9</td>
<td>35</td>
</tr>
<tr>
<td>Figure 10</td>
<td>36</td>
</tr>
<tr>
<td>Figure 11</td>
<td></td>
</tr>
</tbody>
</table>
Motivation

1.1 The Problem

Advancements in technology, though not without their benefits, have people staying in locked onto their screen at home or constantly hooked onto their phone even during social settings. One study found that on average people spent a total of five hours on their smartphones per day; a third of their waking hours [Gregoire, 2015]. Although that might incentivize aspiring mobile app developers out there, a continuation of this trend does have adverse implications for humanity as a whole; do we really want a future where the tool meant to make our lives easier has become our life? It is noticeable everywhere; a group of friends sitting together disengaged, each mindlessly thumbing away at their miniature computers. The same goes for many other scenarios as well; a lecture, a meeting, a date, a movie, and so on and so forth. When it comes to the exhibition of anti-sociability induced by our inherent fascination of the revolutionary pocket-sized software-hardware mashup, we’re all guilty of it, and we’ve all been victims to it.

Furthermore, there exists a large segment of people (all humans?) that have some sort of activity of interest for which they are in fact willing to set aside their beloved smartphones and actively/biomechanically engage in. Throughout the university experience, and even before then, a commonly reoccurring narrative during social interactions went: ‘I want to do this or that,’ to which the response would be, ‘Well why don’t you?’ and whether it be a soccer game (or any sport), a board game, pool, gathering, some team-oriented video game, card game, or any other type of outing, the answer is usually the same, with something along the lines of, ‘There aren’t enough people to do it’. Transitionally, how can technology be used to solve, instead of augment, both aspects of the encapsulating problem: One, people’s increasing passivity and real-life disengagement as technology grows, especially with regards to sociability, and Two, the empirically prevalent disinclination for people to do group activities of interest due to the
cumbersome process associated with getting together the minimal number of participants required for said activities to take place.

It is worth noting that the first part of the problem is a global issue, one for which no definitive, un-superficial solution exists, nor will one be attempted, as the search of an all-encompassing solution lies well outside the scope of this project. Instead, a practical approach would be in taking measured steps, in hopes of reversing the continuing passivity trend, which is what this project aims to do. It does so by providing a feasible, technological deliverable, namely an app, that attempts to address the second part of the problem, as both parts of the problem are intrinsically linked; solving one will either solve, or bridge the progress gap needed to formulate a solution for, the other.

Before delving into the details of the app, it is relevant to acknowledge, and give credit to, the gone-viral PokemonGo app, which served not so much as a source of inspiration for this project, but more so as a ‘proof of concept’. The app is centered around physical activity, with each user trying to, through the app, capture globally placed digital ‘Pokémon’; an absurdly trivial objective for anyone in their adulthood. The user would also often run into others who are trying to capture the same Pokémon, with harder to get-to Pokémon earning the user higher prestige points etc. The description is not meant to diminish the app’s achievements, but give rise to the question: Why, then, was it so successful?

Although no prior experienced using the app had been undertaken, the dynamics of its in-game user experience are not very difficult to grasp. Therefore, assuming a firm grasp of the game’s components (and rightfully so, given their simplicity and the number of acquainted-with users) and taking into account that there are many other Pokémon-related games out there, none of which were nearly as successful, it would not to be unreasonable to conclude that the main factor that was key to its success was either a) The core requirement of physical activeness for which the game is built around, b) People’s wanting to go places where they’ll meet others who they’ll have a ‘special’ bond with, the bond being their shared goal of hunting Pokémon; which hints at the whole ‘human beings are born to connect with others’ psychology principle, or c) Both. The suspicions are that it’s a, and if this were a multiple choice test, one would be torn between a and
c, but would ultimately pick a. Anyway, whether it’s a, b or c, both factors are fundamental building blocks for the project app’s intended functionality, hence, if one or both has been proven on such a large of a scale as PokemonGo’s, it can be assuredly lain to rest that, at the very least, the app’s conceptual foundation is cemented. Why is all of this important? Who cares about all this multiple choice mumbo jumbo?

Well, it gets at both parts of the main problem by proving a critical point, one which serves as a stepping stone for this entire project, which is that Technology can in fact be used to increase activeness, and that requiring the user to physically move as an integral framework can add to, as opposed to hinder, the user experience.

1.2 The App

The app ‘LetsGo’ aims to get people out of their artificial bubbles and on to pursue more socially active ordeals. Users have the option to either create or join (rsvp to) an event/activity of interest; a sort of Kijiji for events. If the user wishes to create an event, they’d need to set, which from a developer’s perspective, are comprehensibly the minimal specifications required of any event: the title (soccer, card game, fishing, debate, video game, etc.), the time, duration, age restriction, gender restriction, location, maximum number of people, and some sort of description. Every time a user creates an event, it is added to a global list, which is accessible to all users. If a user has a specific type of event they’d like to go to, they would open up the list of events, they then can either scroll down or use a search box to find all events of that type, and then ‘join’ one they deem suitable. For example, if the user is interested in basketball, they can type ‘basketball’ in the search box, at which point all basketball related user-created events get listed, along with their respective details. The user can then decide to click on ‘GO’ straight from the list, or click on the event to view all of its details, from which they can then determine whether they’d like to rsvp or not. Once an event has started, a ‘here’ button appears which the user can press to confirm their attendance. The purpose of the app is to provide a systematic, convenient, and aesthetically pleasing interface for handling, and getting users to coordinate, events. There are many facets which need to be worked out in order to preserve the uniqueness and integrity of the app, five of which are either make-it or break-it ones, they are:
User Punctuality
Event Customizability
User Safety
Application’s Competitive Differentiator
Interface Aesthetics and Usability

1.3 User Punctuality

Getting users to actually go to an event is of utmost importance. No app can actually force a user to do something (nor should any app strive to), instead, the focus should be on incentivizing users. For the purposes of this app, some form of practical mechanism had to be put in place that effectively handled no-shows. One popular video game that helped address this is called ‘Dota 2’, where 5 players on a team are matched up against 5 other players on a different team, in a ‘game’ that could last up to an hour. When the video game first launched, there was a real problem with players leaving amidst a game (usually because they were dying a lot in-game), causing their respective teams to lose, and in a lot of these cases if the players had stayed their teams could’ve won. The developers then introduced a rating system, where a player’s rating represented their ‘skill level’, and if a player left a game his rating would get deducted. Needless to say, as a lot of gamers do have big egos when it comes to their skill superiority, the new system was a huge success, with a significant decline in the number of ‘leavers’. With its effectiveness in mind, the task at hand was to come up with a LetsGo-specific penalty system that intelligibly emulated the aforementioned rating system.

As a result, pretty much the exact same system was novelly thought up, one where a user’s rating would indicate his/her punctuality and sociability, and would be translated into a visually recognizable ‘bar’ of some sort, visible for anyone who visits their profile to see. The formula that would capture the underlying logic of such a rating, for a commercially viable version of the app, would construe a flexible variant of the (no. of events-rsvped)/(no. of events-attended) quotient, but not the exact interpretation of, as it would be unreasonable to give a recently signed-up user, who’d only rsvped to and missed one event, a ‘rating’ of zero, and so on. The
aim is to provide an as fair as possible reflection of their punctuality, and have the rating become a prestige of sort that would incentivize the user to maintain; a regular no-show would risk not being welcomed/included into future events, since other users might not want to associate, and participate in events, with a user who has a low rating.

On a side note, although punctuality would be the main determinant when calculating a user’s rating, their sociability would also be considered, where their sociability would be an aggregate of the feedback received per event, although giving feedback has to be optional for the user, as in my opinion making it mandatory would kill the user experience. How exactly such feedback would be handled is, admittedly, yet to be determined, although a plausible idea that presents itself is adding a ‘thumbs up’ button which appears, and that an event’s participants can press, after an event has started. It is useful to note that although feedback is very important for the developer, and it might seem appealing to overhaul an app with feedback mechanisms, most users are not very receptive to the additional ‘hassle’ that comes with having to provide such feedback, and for LetsGo, the main emphasis is to minimize the amount of work a user has to do, and part of that means that when an event is over, it’s over; no extra work on the part of the user is required.

1.4 Event Customizability

Unlike the other facets identified, this one might be a bit harder to varnish, at least on a conceptual canvas, from its title alone. What is being referred to here is the degree to which users should be allowed to customize their events. Should a user be able to upload ten photos? Can they choose the amount of restrictions they wish to impose? Should the user have the ability to move displayed details around (such as the title, the location, description…) into the manner they’re most comfortable with, and if so, would that entail all details or just a select few? First of all, it’d already been established that there is a generic list of details/restrictions that are a minimal requirement of any event, so granting the user deletion rights in that respect was out of the question, so secondly, should a user be able to haphazardly add details, or anything else for that matter (photos, videos, links and so on), at their own discretion? Thus, now that the dilemma’s more vivid and the battlefield’s been set, the topological step would be to formulate a
plan of attack. A plan was needed that would put this matter to rest in one fell swoop, one that ignored, no, one that readily defied, the conventional rules of engagement; this ‘customization’ stage of the logistics-planning phase had to be adequately addressed and settled early on, never to be brought up again, mostly because it solved the event-templating issue, off of which all future work would be based, but also partly because it was consuming too much time and energy to figure out.

The strategy employed was that of traditional observation, coupled with ‘divide and conquer’, a relatively younger strategy, and one that is very popular in the computer science community, though the context of its application therein is usually different. Plainly, all the social networking giants out were separately inspected and analyzed, with the purpose of finding an example that portrayed an instance where ‘less customization flexibility was more effective than more, or vice versa’, and why. Those finding were then put together into an appropriate procedure (the ‘conquer’ phase). During this frantic search, an article was stumbled upon that examined this very issue with respect to two well-known tech giants out there, namely Facebook and MySpace, which stated:

“Facebook seems to have mastered the customization feature, as it allows users to customize their profiles/timelines and pages, while maintaining the Facebook layout. This "controlled" consistency prevents users from feeling lost and getting frustrated when they visit other profiles and pages. With MySpace there is too much customization allowed, causing users to change their profiles into confusing and unidentifiable themes that often frustrated other users, and this became a distraction that drew focus away from the core purpose of the site.”[McNeil,2012]

The same paragraph can also be reread where the words ‘Facebook’ and ‘profiles’ get substituted by ‘LetsGo’ and ‘events’. The initial built-up distraughtness that’d stemmed from the seemingly never-ending search for a solution was quickly replaced with an ecstatic resolve. This tiny paragraph had major implications; one, it met the ‘one fell swoop’ requirement, which would allow one to finally to move on and address other logistical issues, and more importantly, two, exactly what worked, and what did not, has now been pinpointed, and that gained insight can now be applied to the LetsGo app. Hence, innovatively, a heuristic was identified: Provide a “controlled”


**LetsGo layout that would allow users to customize their events whilst preventing them from feeling lost and getting frustrated.** That layout will be addressed in the ‘interface aesthetics and usability’ section.

### 1.5 User Safety

This problem area should be self-explanatory. Would the users be willing and comfortable with going to places with others whom they don’t know and have never met before? As can be inferred, this particular complication can potentially completely shut down all prospects of a commercially-sustainable version of LetsGo, if the emergent answer is *no*. In pursuit of a resolution, a similar strategy to the one described earlier was used, one which used ‘observation’ as a solvent, but instead of ‘divide and conquer’, the new solute was ‘by process of elimination’. The main premise was that if one reasonably successful app could be found whose user experience was subsequently evaluated to be encroached with an analogous safety impediment, then this matter could be lain to rest, at least for the development phase; no one can really know the answer to the safety question until the (hypothetical) launch phase.

The apps that were to be examined were ones in the ‘similar concept’ category. The first one that comes to mind would be Facebook Events. Facebook Events allows Facebook users to create an event, using the original Facebook platform, and send invites to their Facebook friends whom they’d like to include in said event, who can in turn choose to rsvp. A lot of those events would have some sort of privacy setting enabled which would allow those invited to invite their friends as well. The key in evaluating if this concept was similar to LetsGo’s was the word *friends*, which overtly implied that they knew each other. The consensus was that, although users are coordinating to physically go to an event like they would in LetsGo, they are comfortable in eventually going because of the credibility their pre-established friendship adds to said event. Although a good chunk of events that these users go to are not necessarily setup by a close friend, the subconscious logic that takes place, perceptively, is: ‘I’ve met this person before, I can safely assume they’re credible given our interactions, therefore there’s a high probability that nothing bad will happen to me in that event’. The same logic can be extended to include friends of friends, or tweaked to capture the phenomenon where a lot of ‘your friends’ are going to an event, where one doesn’t
know the host, yet decides to go anyway because of the resultantly added credibility that comes from seeing many of their friends rsvped.

The next one examined was not an app, but a website called EventStarter, which allowed users to create events, usually of a more corporate nature, and invite others by email. It’s core concept was quickly prescribed as being more akin to the Facebook Events safety element, although it was noted that users were required to enter their credit card information, which might have added to the safety aspect of things, that and the whole corporate narrative. Still, the concept that was to be proved was one that was relatively riskier, not safer, than Facebook Events. Other apps/websites (Eventbase, QuickMobile) were also examined, but none were found that, where LetsGo’s concerned, was distanced enough in terms of safety from Facebook Events. That was until an app called ‘Party with a Local’ was chanced upon. The app basically helped travelers plan out their holidays, by coordinating with locals in their destination country to find parties and events; to find ‘tour guides’ of some sort who will show them a good time. Now, one cannot pretend to know the reasoning behind the user’s willingness to go to an event half-way across the world with a stranger that is respectively as far away, but again, in order to evaluate its relevance, the key word here would be stranger. Two other key words should also be addressed, specifically party and travelers, which do hint that the targeted user-base might be extroverts, or a variant thereof. Regardless, the short-term strategic objective had been secured, a user-base willing to participate in events with strangers was found, and not just any strangers, half-way-across-the-globe strangers. As they don’t say in Space; ‘I will place the flag here and reposition it the next time I visit the moon’.

1.6 Application’s Competitive Differentiator

One issue that others whom this idea was shared with were kind enough to point out was, ‘Aren’t there plenty of other apps out there that do the exact same thing?’ Well, yes, where linguistics is concerned, there indeed are plenty of apps that use words like ‘events’, ‘create events’, ‘join events’, ‘find events’ and so on. The obvious assertion here is that LetsGo is just like any other event-featured app, which begs the question what is that separates this app from all the rest? While there are many apps out there that do in fact allow users to create and join/rsvp to events, none of
them, apart from the highly-situational one mentioned in the previous section, are based solely around the idea of doing so with complete strangers. Furthermore, the word ‘events’ is used when describing the LetsGo app’s base-feature because that is the closest word which articulates the concept of reference, although in practice, users create both events and activities. Although these are the two features unique to LetsGo, another thing that sets it apart from its competitors is that it is completely event-centered, and that it hopefully, as previously mentioned, provided a more systematic, convenient, and aesthetically pleasing interface for handling events.

It was read somewhere that there were two types of innovation: innovation of product, and innovation of service. That is, one can either create a new physical product (like the IPhone), or find a more efficient way of doing something that’s already being done. The example given was that of Amazon and Chapters; Chapters sold books to customers, but then Amazon came along, and also sold books to customers. The difference? Amazon found a new way of doing it, which is what LetsGo does; it introduces a new way of thinking about, and handling, events.

To summarize this point, and to put it bluntly (or bullet-ly) for any potential skeptics out there, three things distinguish LetsGo from its competitors, it:

- Introduces the concept of coordinating events with strangers.
- Is not centered around ‘traditional’ events, but includes all types of group-related activities, allowing users to quickly and efficiently setup a hobby that requires participants, which they might not have been able to do otherwise.
- Provides a more systematic, convenient, and aesthetically pleasing interface for handling and quickly setting up events, whether formal or informal.

1.7 Interface Aesthetics and Usability

All that was left now was to handle the ‘look’ of the app, and how it will be interacting with the users. For a Computer Science student, one has to really step (more like leap) out of their comfort zone, and get in touch with their artistic side, with the added risk of not finding it, to
deal with this roadblock. In hindsight, the strategy at play here would be that of ‘trial and error’, with a touch of ‘hope for the best’. The graphics and design problem was indeed an artistic one, and was handled accordingly, in the most non-artistic fashion possible; through graphics research. The first thing investigated was how the most popular social networking apps, Instagram and Snapchat, handled the ‘look’ of their app. They used circular portraits for displaying user profile pictures, they also would use shades of colors, as opposed to just plain colors; duly noted. The next step was to google things like ‘graphic designer advice’ and ‘graphic design start’ and other graphic-related searches, of which many returned pages that provided elegant tips, such as: ‘Think about the heart of your brand’, and, ‘Think about what your colors convey to the user’ and ‘Stay hydrated’.

Anyhow, since safety was an issue, it was decided that the LetsGo app should ‘convey’ safety, but nothing too dull. After some color research, it was found that the color green signaled trust and that the color purple was that of excitement. Unfortunately, further research was enlightening as it brought about the fact that green did not blend well with purple (although they did look delightful side by side). Keeping in mind that the research done on colors adhered to science, it meant that science was of the opinion that one of these colors was to be dropped, therefore, purple was let go, mostly because the safety theme was indispensable. A substitute for purple was now needed, because plain green was in fact dull. There were two lists of relevance, one was a list of all colors and their ‘meanings’, and the other was a list of all colors that went with green; without their meanings. The latter list was much shorter, and, accordingly, was traversed, with each of its individual color’s meanings being looked up during the process, with the aim of finding a color with a meaning that was closest to the ‘excitement’ theme. Finally, it was decided that ‘happiness’, which was apparently associated with the color yellow, was closest to ‘excitement’. So now that the two thematic colors have been identified, the next step was to do something with them. A color for the text was also yet to be determined, and white happened to go along with green, and was subsequently chosen. As a result of this thorough search, and after intensive doodling sessions on Adobe Illustrator, a thematic template for the ‘event list’ page was determined, that page was also the prime and only focus for this stage, given that it handled most of the user interaction, and was the ‘soul’ of the app:
Each event in the figure above would have a title at the very top, an optional square ‘event photo’ on the left, which the restrictions are listed beside. The circular pictures are attendees’ profile pictures, up to ten, with the start time and date on the right, above a ‘go’ button which the user can press to quickly rsvp to an event from the list. Part of the description would appear above the user pictures (up to 200 characters), and if the user clicks (or to be more exact, touches-up-inside) anywhere on the event apart from the ‘go’ button, they would be redirected to that events ‘profile’, from which they can view the full description, and all other information in a ‘full screen-ish’ format, there’s also a ‘join’ button that would appear, which the user can also use to rsvp.
Methodology

2.1 App to Project

The model for a fully implemented commercial version of LetsGo should most probably adhere to a 3-tier client-server architecture. A web server, which represents the middle or ‘logic’ layer, would handle all communications between the client/presentation layer, which in this case would be the app, and the database, which is the data storage layer. 3-tier is chosen as opposed to 2-tier as business logic, which would go in the middle layer, would inevitably be incorporated into the marketable app. Said app would have numerous aspects and features, only a handful of which will be focused on. The goal is to provide a flexible, scalable software framework, as well as a smaller fully-functional, self-sustaining API. Such an objective can only be met by identifying the so called aspects and features that are to be implemented. The method employed, that determined relevancy with respect to the project, examined all components separately. Components were sorted according to rough estimates of their essentialness, with regards to the fundamental functionality of the app concept, and their feasibility, given the timeframe and number of developers; the end result sought was a list of pages/screen-views, henceforth referred to as views, to be implemented. The resultant list of core functionalities and views:

- **Signup View** - handles the traditional user sign-up process, and prompts the user for a username, password, date of birth, first name, last name, gender and email.

- **Login View** - handles the log-in process, and effectively queries the database to authenticate user’s username and password.
- **User profile View** - neatly displays user profile picture, first and last name, and birthdate.

- **CreateEvent View** - allows user to create events to be added to the database accordingly. The user is prompted to enter an event title, time, duration, age restriction, as well as an event description. The user is prompted to choose a start time as well as a gender restriction from a drop-down list, and can search for an address to set as their location. Also allows the user to upload a photo from their phone’s photo library.

- **EventList View** – queries the database in order to visually display all events in a scrollable table, with each cell reformattting its respective events’ details into a neatly condensed format. Each cell would also redirect the user to its respective event’s view, should the user click/touch it, as well as containing a working ‘go’ button. Event cells are also sorted in order of proximity to user.

- **EventProfile View** – displays all event details that were prompted the CreateEvent View, and provides working contextual ‘join’, ‘unjoin’ and ‘delete event’ buttons, depending on whether the viewing user is the event creator, a non-rsvper, or an rsvper.

The next sections deal with the process of finding the appropriate software to code each of the 3-tiers. All functionality provided by all the tiers had to be encapsulated in a manner that allowed for easy cross-platform implementation. The operating system that is used is macOS Sierra, and a MacBook Air for the hardware.

### 2.2 3-Tier Architecture

The 3-Tier client-server software architecture breaks down all the client-server interaction process into three separate layers. As previously mentioned, the three layers are: The presentation layer, the business logic layer and the data or data storage layer. The tiers are
basically the manifestations of those layers, meaning their actual, individual software/hardware implementations. The traditional 2-Tier client-server architecture is comprised of two parts; the frontend and backend. Both the presentation as well as the logic layer would get implemented together in the frontend (the app), and the data layer in the backend. The disadvantage of using the 2-Tier architecture is that it impedes scalability, and would make difficult the addition of ‘business logic’, such as secure payment transactions. The main advantage of using the 3-Tier system is that it allows for each tier to be upgraded upon or replaced independent of the others; without minor or no changes to the other tiers’ codes.

2.3 User Interface Tier

There are two prominent operating systems that are modernly used for client software, they are Android and iOS. For the purposes of this project, neither provided a significant advantage over the other, and iOS was chosen due to its compatibility with the simulating software and hardware. The most popular and accessible IDE by far was Xcode, with another one being AppCode. Xcode has a very logical and intuitive interface builder, which provides empty pages that represent views, allowing developers to drag other view objects, such as buttons, text-fields, labels etc. onto the view, which can then be connected to that object’s logic-handling code. It also provides many useful importable APIs, such is MapKit, which is the iOS equivalent of GoogleMaps. Xcode supports two programming languages: Objective-C and Swift. Swift uses a more intuitive syntax, and was chosen due to the amount of guiding resources available.

2.4 Functional Logic Process Tier

The choice of server is not important for the implementation of this particular layer, as long as it was a known HTTP server and supported PHP. The one chosen was Apache, the most known and tested, and comes packaged with OS X. What is more important, however, is the language of choice. Many languages can be used for server-side scripting, including: PHP, Perl, Java, Python, Ruby, JavaScript (Node.js) etc. PHP, especially in terms of cross-platform scalability, was the most advantageous, as it can run on both UNIX and Windows servers. Also, PHP can be imbedded into HTTP, and also provides many built-in methods for handling HTTP requests, as
well as handling JSON data conveniently. It can interact with all sorts of client-side softwares uniformly, so long as the uploaded data was encoded appropriately (into JSON).

2.5 Data storage Tier

There are many available databases that can be used, each with their own advantages and disadvantages. The main focus of this project’s architecture is the emphasis on future scalability. The many different databases include: MySql, Oracle, SQLite, PostgreSql etc. Although all use structured query language (SQL), their language syntax varies to a degree. SQLite is not a client-server database, but is embedded into the client software, and is thus only used with client-side only applications. Oracle, in terms of performance, is very similar to MySql, however Oracle does not provide a General Public Use (GPL) license, meaning developers would need the Oracle Corporation’s permission to use it commercially, which would hinder scalability. PostreSql, which was developed with an emphasis on extensibility, is commercially advantageous to MySql, as it is superior in most respects; better performance, which includes querying speed, and is also more secure and has faster data replication. MySql comes packaged with OS X. MySql and PostreSql have identical syntax; are SQL-compliant. MySql was chosen due to its familiarity, as it can easily be transferred to PostreSql, and provided the exact same functionality for the software framework objective of this project.

2.6 Implementation process

The initially intended approach was to implement each layer separately, starting with the user interface. Although theoretically possible, the approach did not work in practice as it had one major flaw; it significantly hindered manual testing. Thus, another, more effective approach, was used; the implementation of all 3 tier-related aspects per view, with manual testing in-between views.
Technical Aspects

3.1 Classes

The app consists of many inter-related classes that provide its desired functionality. In order to make sense of them accurately, they are split into two types: Object Classes and View Classes. The object classes contain all necessary member variables and client-interacting functions, which are in turn used by the view classes for display. The three main object classes, depicted in the UML diagram below, are thisUser, User and Event.
FIGURE 2

The getter and setter methods are omitted from the above diagram. The ‘?’ means that the value is optional; can be used before the assigning of a value. Although all the variables are public, except for password, the getters and setters are used for all access and manipulation of variables, except in one case, that when the User changes their photo, and for no particular reason. The thisUser class is not explicitly implemented as a subclass of the User class in code, which is because all of its variables are static. The unfilled arrowhead denotes their generalization association, as they virtually assume the same functionality, therefore it would be redundant to display the thisUser class’ relationship to the Events class. As can be inferred from their respective names, the thisUser class presents the actual user using the app at any one time, the User class represents any ‘other’ user with respect to thisUser, and the Event class represents a single event. The thisUser class’ member variables and functions are all static, with all its variables being assigned queried values once the user logs in, with the value persisting until the app is closed. The reasoning behind this is that the logged-in user’s details are insignificant with regards to hardware performance and memory, and should be readily accessible to the views at all times. The User class has no static member variables, nor any setter functions, as its main purpose is for it to be used for displaying queried user data. The Event class handles both the creation of events, which includes uploading an Event’s details into the database appropriately, and is also used by the view classes for the efficient displaying of events.

3.2 The Codable Protocol

All three of those classes are centered around, and make use of, two main structures. The first is ‘userinfo’ and the second is ‘eventinfo’. Both structures’ variables have corresponding variable in PHP, which is not completely necessary but helps in handling NULL values being uploaded, and they also have corresponding columns in MySql. The more recent Xcode versions provide a means of quickly and easily encoding data into JSON; they provide a ‘codable’ protocol, which is an extension that can be used in the declaration of any class/struct. Structures and classes are said to adopt a protocol when said protocol’s name follows that objects’s name during declaration, separated by a colon. The Codable protocol has a default JSON encoding technique
which is very effective for encoding basic data types, but in order to encode structs and classes that contain other data types (such as UIImages), those structs and classes must **conform to the codable protocol**. This basically means that after adopting the protocol as described above, the automatically inherited encode(to:) method would need to be implemented. Swift’s syntax for declaring functions can quickly be grasped from the encode(to:) method’s declaration, which is

```swift
func encode(to encoder: Encoder) throws
```

The ‘func’ prefix’s purpose is clear; to tell the compiler that this a method. The ‘encode’ before the brackets is the method’s name. The ‘to’ is the variable’s name, which would mean that the ‘to’ would have to be specified (typed) every time the method is called, with the assigned value to the right hand side of the colon. The lower-case ‘encoder’ is unique to this function, and would encode an empty container (or for easier interpretation, ‘value’) if the method fails to encode a value, the capitalized (‘Encoder’) is the variable’s type, so it can potentially be, in other methods, replaced with ‘Int’ or ‘String’ etc. If the input’s format is incorrect, the method would throw an error, hence the ‘throw’ keyword. On a side note, if the ‘to’ was prefixed with an underscore, it would not need to be specified (typed) every time the method gets called.

When a class adopts the codable protocol, it automatically adopts the encodable and decodable protocol, whose’ separately added functionality can be inferred from their names. Since both structures use only basic data types, the implementation of this method was not necessary for either one. However, in the ‘eventinfo’ structure, an implementation of the reciprocal method for the decodable protocol was required. For decodable to decode an Int, for instance, the encoded data should be of type Int. PHP has a `json_encode( ‘variable goes here’)` function, but that function encodes all data into String format, with the exception of Dates, which will be discussed in the section. As a result, when JSONDecoder() (which is the Swift object that handles decoding) tries to decode an Int into a hypothetical structure, it finds an object of type String instead, and throws an error.

Examining the inner-working of the following code snippet illustrates this more clearly,
MaxAge = Int((try container.decode(String?.self, forKey: .MaxAge) ?? nil)!

The underlying logic that is taking place here is that the decoder is told to expect a String, for the key ‘MaxAge’ which is declared in a separate enumeration (enum) in the same structure, that conforms to another protocol called CodingKey, which allows the struct’s variables to be ‘referred to’ during the coding/decoding process. The key in this code snippet refers to the MaxAge Int variable that is being set and was declared elsewhere within the struct. The ‘??’ nil coalescing operator checks if the resulting value from the decoding attempt is nil, and if not then that value would be used for the surrounding inner brackets, otherwise if is nil whatever is to its right is returned, which in this case happens to, coincidently, be nil. The important detail to note is that the decoder is told to expect a String, and if one exists that is not deemed nil by the ‘??’ operator, then the regular Int() method parses that String into an Int. Once the decodable method has been implemented, the new implementation is used, as opposed to the default one, which subsequently means that all variables intended for decoding must be specified in the implementation; basically, code that has roughly the same format as the code snippet above would have to be provided for each variable in the struct.

3.3 Encoding and Decoding Dates in Swift

During the coding phase of this project, what quickly became prevalent was the notion that extreme care needs to be undertaken when working with dates, and that applies to all three tiers; the client-side, the webserver side, and the database side. The way in which dates are handled in Swift is very counter-intuitive. Date variables need to be assigned Calendar variables to help them determine the context by which to handle the date value, and if no Calendar is provided a default one is used. Date values also contain time details, including: hours, minutes, seconds, and milliseconds. The JSONEncoder() object contains a variable called DateEncodingStrategy, which if not set converts a date to its UNIX format by default. The JSONDecoder() object has a corresponding variable called DateDecodingStrategy. The variable can be set to some custom DateFormatter() object, or a custom strategy can be implemented and used; The former strategy
is much more logically intuitive. A built-in Swift date handling strategy is ISO8601, which uses an ISO8601 standard format to display and interpret dates.

It is worth mentioning that although MySql contains a method for converting UNIX values to dates, some sort of bug transpired that would subtract some 40 years from the date, setting its year to the 1970’s; and whether that bug was on the client/server/database side was unclear. What proved to be successful was manipulating Strings into date formats and vice versa. As mentioned, a Data value in swift refers to both a date and a time, and a variable of such type would include both. In MySql however, dates and times are separate data types, although a similar data type to the Swift ‘Date’ one exists in MySql, called DATETIME. The following sequence of instances should clarify the complications associated with sending and receiving dates with times to and from the databases, more specifically, sending a LetsGo event’s StartTime variable value.

To send a date, the JSONEncoder().dateEncodingStrategy variable would be set to .iso8601. This means that the date value in question will be converted to JSON in the following format: "yyyy-MM-dd'T'HH:mm:ss.SSS'Z". The PHP json_decoder then decodes the JSON/text data as is, as a basic String and it gets stored in a variable, $StartTime. The $ symbol in PHP is called ‘sigil’, and prefixes all variables; PHP is not very data type supportive. The DATTIME value cannot be set using a simple “SET StartTime = '$StartTime' ” partial query as part of a full query, instead what works is telling MySql how to interpret the String’s format, by instead fixing said partial query to “SET StartTime = str_to_date($ei->StartTime,'%Y-%m-%dT%TZ')”; basically, notifying MySql of the format to expect (MySql takes all relevant values from the provided String, and discards the rest, such as milliseconds etc.); the single-quotes are also omitted in the latter example as the method returns a Date. The date is now stored in the database in the this format: "yyyy-MM-dd HH:mm:ss". When the Event class’ getInfo() method attempts to retrieve event information from the database and the server, the JSON encoded data provided is decoded into an eventinfo struct.

As mentioned earlier, an implementation for decoder had been manually implemented, which meant that setting the dateDecodingStrategy variable would now be ineffective, as the decoder
was using the newly implemented method as opposed to the default method which referred to the
dateDecodingStrategy variable for handling dates. As was done with the provided decoding
implementation of the ‘MaxAge’ variable earlier, the same overall logic was applied, but instead
of parsing to an Int, the returned String was maintained, and the StartTime Date variable had its
its value set through the DateFormatter() object, as simply parsing using Date(‘value here’) does
not work. The DateFormatter() had its dateFormat variable set to the String "yyyy-MM-dd
HH:mm:ss", which notified it that when using the date(from: String), where the ‘from’ variable
is expecting to be set to a value of type String, it should interpret and return the Date (which
StartTime gets set to) in correspondence with this format.

Another Date-related pitfall occurred when trying to set the StartTime from the
CreateEventView. Swift does not provide a straightforward process for directly assigning a
Date’s different dateComponents (another Swift object). The Date object’s Calendar seemingly
kept overriding the ‘hours’ component whenever it was explicitly set, with all the hours, minutes
and seconds increasing in direct correlation to the simulating hardware’s clock, although on a
different Time Zone. The Calendar object had a startOfDay(date: Date) method, which, coupled
with the addTimeInterval(interval: Int) that adds the amount of seconds evaluated, should
intuitively solve the problem, since the Date could be reset, and then added to appropriately.
However, again, the Calendar was resetting the day on a different Time Zone, and changing the
Calendar ‘locale’ property, which supposedly changed its local time and time zone, produced no
results, with the Date’s hours value getting set to 05:00 am as opposed to the intended 00:00 am.
The following code snippet solved the problem:

dateFormatter.timeZone = TimeZone(secondsFromGMT: 0)

A new TimeZone was to be initialized through the (secondsFromGMT: Int) overload of the
TimeZone constructor, which the dateFormatter’s timeZone variable was in turn set to.

3.4 Uploading Images in 3-Tiers
Swift provides a data type for storing images called UIImage. Unfortunately, UIImage does not conform to the `codable` protocol, which is logical as image data cannot be converted to JSON, unless Base64 encoding is used, which greatly impacts the quality of the image. Additionally, Swift contributed no alternative protocol or functionality in general, that provided a straightforward solution for transferring image files to the server. To address this issue, instead of using the POST/GET requests that handle all of the other app’s value-sending functionalities, a multipart/form was used.

A user can choose to add an image when creating an event, and can also choose to change their profile picture. The CreateEventViewController class and the MainProfileViewController class handle each of these instances, using implemented methods from the Events class and the thisUser class, respectively. When an event is created, the CreateEventViewController class sends all of the user’s provided details, except for the photo, which is uploaded once the Event class’ `updateImage(photo: UIImage?)` returns, to the server; the question mark next to the UIImage data type exclaims that the variable is optional and can have a value of nil, and if so, in this case, a default image is provided. The reason that the image is not uploaded alongside the other values is because the event’s id is needed for its storage, which occurs on the server-side, as opposed to the database-side where everything else is stored. In the case of the user’s changing their profile picture, the user’s id, which is also needed for the picture’s storage and is a static variable, is readily accessible. The reason both ids are needed is because PHP uses the ids as the the images’ names, and then sets the photopath value in either the ‘users’ table or the ‘events’ table, where the photopath is the full link to the server suffixed with the image’s filename, which in this case is also the id of the event or user involved. The following figure illustrates the 3-tier interaction that takes places when the Event class’ and the thisUser class’ `updateImage(photo: UIImage?)` methods are called:
Both the thisUser and the User classes have constructors which take in arguments of the userinfo struct type, and set their duplicate member variables’ values accordingly. The Event class’ constructor similarly takes in an eventinfo struct type, and returns the events id as an Int. The Event class has a non-static void member function, getinfo(), which gets all of its information from the database using its *already existing* id, and re-sets the values of its member variables. The method is used to ‘refresh’ the event’s details if some sort of update on the server/database side was to take place. It also has a static method that takes in an id as an Int, and returns a structure of type eventinfo, which can be used alongside the constructor after a list of event ids has been obtained for some particular use. The User class has a similar but non-static get by id function, while the thisUser class has the same void member function. The point is to provide a better understanding of what transpires when an object of these classes is initialized.

Although the UIImage data type does not *conform to the Codable protocol*, it does contain a useful constructor that takes an object of type ‘data’ as an argument. The following code shows has this particular overload of the constructor can be combined with other Swift objects and their respective functionalities to make use of the link stored in the potopath variable in order to access and display images *directly from the server*:
if self.photopath != nil {
    let url = URL(string: self.photopath!)
    let data = try? Data(contentsOf: url!)
    self.profilepicture = UIImage(data: data!)
}

The code’s logic is very straightforward. First, initialize a URL() object from its (string: String) override, setting that String to the unwrapped (hence the exclamation mark) photopath variable of the optional type ‘String?’. Next, try to parse the url’s contents into a data type of type ‘Data’, and if successful store them, and use the UIImage’s previously described constructor to display the data. It should be noted here that if the data in question is not that of an image’s, the app would crash, which is why the 3-tier workings that make use of the photopath variable are handled with great care and precision; a safer way of doing this would be through the use of a try/catch/throw geometry. The figure below shows how the three main object class’ getinfo() methods, that are mostly used along with the constructor, work with the above code:
This overall blueprint of sending and receiving images offered the most precision, whilst also minimizing unwanted traffic on the part of the server; another method would entail re-encoding the image on the server side, and then passing it to the client, which would add undesired pressure on the server’s operating hardware, or one that could re-prompt the server and the database for the photopath variable, unnecessarily adding the database into the procedure as well as increasing the number of requests the server has to handle.

3.5 View Controllers

View controllers are an app’s version of ‘pages’. Every app has at least one or more view controllers, each assigned a certain user-interface functionality. Of the ones discussed earlier (that were to be implemented), the two wherein most of the LetsGo app’s implementation is concentrated are: The CreateEventViewController class and the EventsViewController class (not to be confused with EventViewController), which sorts and lists events by their proximity to the user. The CreateEventsViewController prompts the user for details in a reasonably aesthetic and relatively easy to use manner. The prompting method by which it does so ensures that all details provided are of the appropriate format, and parses them when necessary to ensure they meet the server’s expectations. For instance, when picking a gender restriction the user is presented with a drop down scrollable list, which is generated completely programmatically (no interface dragging/connecting involved) and is an instance of the UIPickerView Xcode object, which they have to choose from, and if no selection is made the CreateEventsViewController class won’t segue to the next ViewController, namely the EventViewController and a notification error appears informing the user of the missing detail, or in other cases (if a non-numerical value was entered in MaxAge) the error message would inform them of the ‘wrong’ format.

The EventsViewController gets the user’s location and stores his or her longitude and latitude. It then makes use of the getEventsByProximity(longitude: Double, lattitude: Double, searchValue: String) -> [Int]? method, which takes as arguments the previously obtained user’s latitude and longitude, as well as a searchValue which represents what the user types in the search box that appears in the ViewController. The method returns an optional Int array which would be the list of the now sorted Event ids. If no value is given for the searchValue, the EventsViewController
class used an empty String instead. The following PHP-SQL code is used to get the list of sorted events:

```php
$searchValue ? $sql = "SELECT id FROM events WHERE Title LIKE '%$searchValue%' ORDER BY (POW((longitude-$longitude),2) + POW((lattitude-$lattitude),2));" : $sql = "SELECT id FROM events ORDER BY (POW((longitude-$longitude),2) + POW((lattitude-$lattitude),2));";
```

The ‘?’ ternary operator in PHP evaluates the variable to its left to see whether it has been set/is not false/is not nil/is not 0, and if so, the block of code that precedes the colon gets executed. Otherwise, the block of code to the right of the colon is executed. The two ‘%’ symbols guide the LIKE command’s execution, by telling it to look for an occurrence of the search value in any position within the ‘Title’ column-value. The sum of the event’s longitude minus the user’s ‘uploaded’ longitude (the $ in this case denotes that the variable is not of SQL) is squared to get its positive value (since distance is being measured here), the same thing goes for the latitudes.

Some ViewControllers are more covert; there is not always a ‘path’ within the app to get to them. Instead, they serve as helpers for another ViewController’s user interacting objects. For example, when the user is creating an event via the CreateEventViewController, he has to choose a location. A textfield is provided, one which when clicked on (touch-up inside, more specifically), redirects the user to another ViewController which handles his search of an address. On an additional note, to give a ‘smooth’ transitional effect to the redirection process, the textfields cursor had to always remain hidden, and it had to disable editing. Xcode provided straightforward ways of dealing with the latter, but not the former. To disable the cursor, a custom UITextField class had to be implemented, which overrode the textfield’s drawing constructor, in a way which it never appeared; if editing were to be enabled, it would indeed appear.

The different ViewControllers within the app add functionality to their view-objects which segue the user to a different ViewController. The figure below, taken from the interface builder’s storyboard, depicts all the different ViewControllers and their segue relationships.
Though the ViewController’s names are not visible enough to read, their intended purpose could be inferred from their design. The ViewController with the disorganized red, yellow and green (delete, join and unjoin) buttons at the bottom is the EventViewController, these buttons are only there for implementation reference, although when the app runs, each gets repositioned, hidden or replaced depending on the user accessing the view and their ‘status’ in relation to that event (Owner, has not joined yet or has already joined).
3.6 3 Tiers, 2 Databases

For practical testing purposes, the non-client side storage was separated to two databases, although it was only a symbolic separation, as the same hardware would in practice be used for both. When an event is created or a user signs up, a respective MySql table that had either the name ‘Event’ followed by that event’s id or ‘User’ followed by that user’s id, depending on which object type was being added, in a different ‘storage’ database, which was only used for data storage and retrieval, as opposed to the main ‘LetsGo’ (the main database’s name) database which was used for more complex querying. The ‘Event(id)’ table would store a list of user ids for that event, the ‘attendees’, and a Boolean value for each, which keeps track of who attended and whom did not. The ‘User(id)’ table would keep track of all the events that user has rsvped to, and only contains one column, ‘eventids’. The reasoning behind such a database separation is that as user-based manual testing increased, the number of tables being created increased, which was only an issue when some sort of flaw was found whereby the app had to be retested from scratch; all tables dropped, and the two main users and events tables emptied. When those tables were put into another database, whenever the resetting process occurred, the dispensable and easily substitutable ‘eventstorage’ (as it was called) database would get deleted and then re-created.
Results

4.1 Implemented functionality

All of the desired features and functionalities were successfully implemented. The following will consist of a series of steps whereby screenshots of each intended functionality’s views are presented, and a corresponding explanation to commune what the user would experience, and any notable information:

✓ Signup functionality

FIGURE 6
The users can choose to either sign up or log in, and their choice is handled appropriately; redirects or *segues* them to the appropriate view. If the user wishes to sign up, they are directed to the next view which prompts them for the first and last name, as well as to specify their gender. Once all of the requested is provided, the gender is chosen and the first and last name consist of 3 or more letters then continue button is enabled, which they can then click/touch-up-inside to ‘continue’.

The following view prompts them for their date of birth, which they chose from a ‘datepickerview’ whose presentation and format is handled by Xcode’s UIDatePickerView. Once a date has been chosen, and the user is authenticated to be 10 years of age or older when they click the continue button, they are segued to the final sign up view. The final view prompts them for a username, password and email. If all are provided the ‘Sign Up’ button is enabled, however if the username or password has less than three letters, or the email is of an invalid format, the `shouldPerformSegue(for segue: UIStoryboardSegue, sender: Any?)` method would return false, and an error notification is provided to inform them of the incorrect value. The superficial check for the ‘correct’ email format only checks if the ‘@’ symbol is present, and whether the email consists of 6 or more characters. A unique thing to this ‘Sign Up’ part of the functionality is that all four views are handled by the same ViewController class, and to allow that to happen a ‘navigationcounter’ variable was added, to effectively manage the persistence of data from view to view.

✔ Login functionality
The common Log In functionality is implemented here. The thisUser class’s `userExists()` method queries the database to find a matching user, and if one is found returns true. The password is then evaluated to see if it matched the one associated, and stored, with the username in the database. Both the username and password values in this case are encoded with the String `utf8` format, as opposed to JSON, because they are basic String data types.

User Profile
The User Profile displays the user’s first and last name, and their date of birth under. The profile picture, which they can choose to change, is cropped circularly by making use of the cornerRadius variable. The one used in the above figure is obtained from the app’s own photo library, using the ‘Change Picture’ button’s functionality. All non-login/signup views have a quick navigator at the bottom. If the +E is pressed, the user would be segued to the CreateEventViewController. If the user presses GO, they are segued the global list of events, sorted by proximity, in the EventsViewController. The P at the bottom segues to the MainProfileViewController; the user’s profile being discussed. The user can also click on the ‘My events’ button, which would also segue them to the EventsViewController class, but would display events they’d RSVPed to instead.

FIGURE 8
Create Event; allowing user to pick from drop-down list of addresses

FIGURE 9

All prompted details in this view are mandatory, with the exception of the maximum age restriction. Event titles can be up to 30 chars long. Upon choosing the date of the event, a DatePickerView similar to the one in the Sign Up view is presented, which the user needs to scroll through and pick the date from. However, since the date value is not the only one being prompted in this view, a toolbar is added on top of it, like the one shown in the figure above,
with a ‘done’ button which would hide the date picker and use the highlighted date for the
textfield’s value. The time requested the hours and minutes for the ‘start time’ of the event. Both
the date textfield’s and the time textfields’ values are stored in the StartTime variable. The
duration textfields prompt the user for the same hour and minute types, with the difference being
that these hours and minutes now represent the length/duration of the event, and are stored in the
Duration variable for that Event class’ initialization. The above figure captures the gender
restriction setting process, with an implemented custom UIPickerView object being presented.
The minimum age textfields would present a numeric keyboard to be used, which is an option
that can easily be enabled from the interface builder, the minimum age also has to be greater than
10, and the maximum age less than 120. The description text view is very similar to that of a text
field, but enabled for more text to be inputted. The location-setting has already been thoroughly
discussed, and segues the user to a a ViewController, which acts as a full-screen sized drop down
list of addresses with a search box at the top. When the CreateEventViewController is
instantiated (segued to), its member eventinfo variable is initialized, and aids in the initialization
of an Event object

✓ Event Profile; that is accessible from Event List
In this profile view, as opposed to the user’s, the photo cannot be changed, specifically because once an event is created, it cannot be edited; if the user wishes to edit they would have to cancel and make a new event. The ‘Cancel Event’ button appears because the owner of the event is the one accessing the view, as shown on left in the above figure. The screenshot on the right is taken from another user’s perspective, who is not the owner, and has not rsvp’d to the event. The ‘here’ button is disabled, and is not implemented although its implementation would only need to check the event’s Start Time against the present date, as well as a distance checking query, similar to the
proximity one used in the EventsListViewController, to check if the user is within, say, a distance of 10 meters?

✓ Event List View; sorted and listed events in order of proximity to user.

![Event List View](image.png)

FIGURE 11

The yellow ‘GO’ button can be pressed to directly rsvp to the event from the list, where it would get replaced with the green check mark displayed above. The check mark can in turn be pressed to un-rsvp, and would accordingly be replaced the yellow ‘GO’ button. Up to 10 circular attendee images can be displayed. The textfield above the quick navigator can be used to search
for desired events, which the search is concerned with finding corresponding event title character values to match the search value inputted. It is also worthy to note that the textfield is surrounded by an artificially generated shadow, which was specified in the EventsViewController’s code. The event owners cannot click on the check mark to rsvp.

4.2 Further work and conclusion

The rating variable was not integrated into the project app though all the necessary tools for its calculation were implemented. It was also unnecessary to implement for the purposes of this project, as the adoption of its formula would include research into other fields, which can be considered as further work that can be contributed to this project. The I’m here button should also be implemented, who implementation tools were also provided. An essential component for a final version of this app would be a ‘chat’ feature, which would allow all users rsyping to the same event to communicate. Direct user communication or ‘chatting’ should also be considered. A market strategy-based feature to expand the user-base rapidly would be to allows users to invite their Facebook/Instagram/Snapchat etc. accounts, and provide an option where the user can log-in using their said apps’ log-in credentials.

In conclusion, the project was successful in its goal of providing a scalable framework, which provided the necessary features and implementations that would allow future features and implementations to be added. All the testing done was user-based manual testing, which was deemed to be the most efficient. Even after the project was thought to have been fully implemented, the final testing brought to attention a major bug, where the app would rarely crash when seguing the events list view. The problem was that the app was trying to unwrap the location object when determining the user’s location, before the object’s value had been set. It’s crashing likelihood would depend on the internet speed of the connection being used. The problem was solved by forcing the dispatcher to wait until the location (or CLLocation to be more precise) object’s value was set. In the end, although the project app’s implementation was underestimate in terms of time consumption and effort-required, the desired implementation was realized.
References

