Blips - A City Mapping Application

By
Cyrus Sadeghi-Emamchaie (100934822)
Jonathan Chan (100936881)
Eliab Woldeyes (100937656)

Supervisor: Dr. Chung-Horng Lung

A report submitted in partial fulfillment of the requirements
of SYSC-4907 Engineering Project

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University

April 11, 2018

Abstract

Travelling as a tourist can sometimes be confusing. Many tourists face difficulties navigating in a new city which makes looking for an activity or point of interest quickly become a hassle. Demands for a point of interest can vary from cases of urgent need, to a more leisurely setting. Blips aims to make travel management easier for tourists by centralizing points of interest in an intuitive user interface; allowing users to easily explore points of interest in a desired destination with their mobile device. By servicing offline clients through an interactive Blips kiosk with network connectivity, and providing features that will notify the user of a nearby point of interest or suggest an attraction type based on their profile, Blips will give tourists an enhanced travelling experience.

The main components involved in implementing Blips are: a multi-platform (iOS and Android) mobile application, a remote server application, and a touchscreen kiosk station. The mobile application accomplished implementing features such as: a user system, suggestions, navigation, dynamic location display, and attraction information. The server was able to gather and cache information about points of interest and serve them back to the clients. The touchscreen kiosk was able to interact with the user by providing an infrared touch screen overlay to be placed on top of a monitor.

Acknowledgements

We would like to thank Professor Lung for helping us create the initial idea for the project as well as providing guidance and general support while we were working on our project and its deliverables. We would also like to thank Professor Liu for his feedback following our oral presentation. Finally, we would like to thank our friends and family for giving us suggestions on what they like to see in our final project.

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
ELB	Elastic Beanstalk
IDE	Integrated Development Environment
iOS	iPhone/iPad Operating System
JDK	Java Development Kit
JSON	JavaScript Object Notation

Chapter 1 - Introduction

When one is travelling to a foreign city or country, they may not know what attractions to visit, what restaurants to visit for a meal, or where to find somewhere to sleep (such as a hotel). Having a simple application on your phone or on a stationary kiosk which allows you to find attractions (or other locations, such as banks or stores) nearby would be very helpful. Being able to filter these attractions by cost to visit, minimum user rating, and travel time would ease a tourist's mind.

1.1 Problem Motivation

With regular mapping applications (such as Google Maps), finding specific points of interest may be difficult. One may not know exactly what they are looking for and may want some suggestions on what attractions to visit. A regular mapping application does not provide suggestions on attractions to visit or restaurants to visit for a meal. Having an application that can use your usage history to suggest attractions would solve this problem.

1.2 Problem Statement

This project solves the issues of attractions suggestions, attraction wishlisting, and attraction filtering. As discussed earlier, an attraction suggestion system would aid the user in finding major attractions (such as Parliament Hill or the CN Tower) near their location or in a different city.

The attraction suggestion system should track the user's behaviour to find the most relevant attractions for the user. The user behaviour tracking should span across all of the user's devices, allowing recommendations to be made no matter what device the user is currently using.

The attraction wishlisting system would allow a user to create a list of attractions that they would like to visit in the future. The user should be able to search for attractions in other cities and add them to this list. The user's list of saved attractions should also be synchronized across all of the user's device, allowing them to use any device to find their saved attractions.

The attraction filtering system would allow a user to filter the results for a search by a variety of parameters. For example, a user may not want to visit attractions that have a high cost of entrance or attractions that have low user ratings.

1.3 Proposed Solution

The final product should be comprised of three major components. These components are the web server, the mobile clients (iOS and Android implementations), and the stationary touch screen kiosk.

The web server must be the primary source of information for the clients. The web server should maintain information regarding attractions around the world. The web server should also be able to maintain a user database, this database should contain the user's preferences and sufficient information to suggest new attractions to users. Finally, in terms of non functional requirements, the web server should be responsive, secure, and have high availability (ideally, the server should always be available, the server should never be in an unavailable state).

The mobile clients should be the user's primary interaction with the system. The mobile clients should allow the user to manually find attractions in close proximity to the user. The mobile clients should automatically notify the user when they visit a city, providing the user with a list of suggested attractions. The mobile clients should maintain a user account system that tracks how the user uses the client application. This usage history should be used to automatically suggest new attractions to visit. The mobile clients should also be able to find attractions that are not necessarily points of interest (for example, hotels, restaurants, grocery stores). The mobile client's map interface should be easy to use and provide detailed information about

each attraction on the map (historic attractions, such as Parliament Hill or the CN Tower should also have historical information included). The mobile clients should be available on both iOS and Android, allowing the user to use the system no matter what mobile device they use.

The touch screen kiosk should broadly offer the same functionality as the mobile clients, keeping in mind that the kiosk is stationary. The kiosk should limit the displayed attractions to an area encompassing the city that the kiosk is placed. The touch screen kiosk should also use the web server as a source of information.

1.4 Accomplishments

1.4.1 Web Server

As of the end of the project, the web server implementation includes all original requirements described in the Proposed Solution section.

- Point of Interest lists are quickly retrieved and served to the user. Attractions can be found in any country and can be of any type.
- User accounts are managed by the server. The server uses account information to suggest new attractions to users and allows the user to transfer their account to another device.

These requirements (and their implementations) are explained in depth in chapter 4.1.

The Proposed Solution outlined a number of non-functional requirements for the web server. As with the regular requirements, the final implementation meets all non-functional requirements.

- The server is very responsive. Client queries are typically serviced within five seconds.
- The server handles user information in a secure manner, to prevent any confidential information (i.e. emails, passwords, names) from leaking.
- The server has high availability. High availability requires the server to be ready to service a client request at a moment's notice.

Non-functional requirements for the web server are discussed in detail in chapter 4.1.1.

1.4.2 iOS Client

At the end of the project, the iOS client meets most of the original requirements outlined in the Proposed Solution section.

- Points of Interest can be found manually by the user. Points of Interest can be filtered by type, rating, distance from user, and price range.
- Points of Interest are automatically suggested to the user, based on their usage of the application.
- A user account system allows the user's history to be used to recommend new attractions to the user. User information is synchronized with the web server and can be accessed from any of the user's devices.
- Points of Interest are presented in an easy to use map and attraction interface. Details for each attraction (such as location, rating, and price range) are displayed.

These accomplished requirements (and their implementations) are explained in depth in chapter 4.2 (figures of the final product are also provided).

The following proposed requirements were not completed due to time and resource constraints.

- The client does not actively track the user's location, preventing new points of interest from appearing on the device.
- Interactions between the iOS client and touchscreen kiosk were not implemented.

1.4.3 Android Client

At the end of the project, the Android client does not meet most of the requirements that were stated in the proposed solution. For those that were met, the technical solution is outlined later in section 4.3. The completed goals for the Android client are as follows and mainly revolve around allowing the user to lookup a point of interest:

- The client can successfully communicate with the server by querying for attraction types, and location information.
- Provides in-app navigation between views with toolbar functions.
- Generates a map as the main view page of the application.
- Generates custom page showing a list of attraction types received from the server.
- Allows search restriction for the custom page based on a search radius input field.
- Provides search result information to the main map view.

Since the team initially had little Android development experience, the work involved was underestimated. Much of the implementation focus was on the iOS client and the following features were not able to be accomplished:

- User account system (or on-device persistent storage of user data as an alternative, more on this later in section 4.3).
- Notification and suggestion features based on user data.
- Display of location data as icons on the map view.
- Display of attraction information.
- Integration with kiosk to allow for communication between the device and the kiosk.

1.4.4 Touchscreen Kiosk

The touch screen kiosk meets most of the hardware requirements, but does not achieve all of the software requirements. The hardware component of the system have been completed and are comprised of the overlay, which enables a touch interface for a set display. The hardware contains a column of 16 infrared emitters on one side and a column of 8 infrared phototransistors on the other, the space between the columns create the touch interface. The software to determine the position on the interface that was touched has been completed, it involved recording breaks in the emitter to sensor lines to determine the intercept points (i.e. the touched location). The designed circuits were constructed, and the final software implementation was able to determine touch intercepts. The specifics of this implementation are covered in section 4.4 of this report.

Regrettably, the current system does not accomplish most of the proposed requirements due to underestimation of time required, they are:

- Determining the user's action on the interface, for example, clicking or dragging.
- Applying the intercept to the Operating System as a mouse action.

Due to an unexpected technical limitation, the goal of presenting the iOS application on the Raspberry Pi was determined to be very difficult due to hardware compatibility and emulation issues. Emulating the Android client on the Raspberry Pi is much simpler, however due to time constraints, the Android client was not fully implemented (as explained in section 1.4.3).

1.5 Organization of the Report

This report includes the methods and description of how the solutions were implemented to create an effective solution for city mapping. The Technical Solution section of this report presents the significant accomplishments made in each component of the project which are the web server, touch screen kiosk, iOS client, and Android client to create a comprehensive solution.

Chapter 2 - The Engineering Project

2.1 Health and Safety

The Blips application runs on smartphones and poses no inherent health and safety risks to users.

2.2 Engineering Professionalism

The team members met their professional responsibilities by being responsible and accountable for their contributions to their own component and to the overall project. Each member had a clear understanding of the team's goals and was committed to achieving them. Significant decisions regarding project milestones were made as a group, no one group member contributed more to the project than any other group member. The team worked well together, because all group members trusted that all other members would get their work done on time.

2.3 Project Management

Development plans for each project component were created separately, because the project components were relatively loosely coupled. Each project component had a single team member assigned as the component lead, the component lead was tasked with creating a development plan for their project. This included breaking down each desired deliverable into a number of smaller tasks to complete and then integrate. For deliverables and major decisions that concerned the entire group, all team members became involved.

The overall project management structure was flat, allowing communication and information to be easily shared among all members. The team did not have a single leader, instead team members would step up and direct the group as required, emulating Facilitative Leadership. Each member was mindful of the deadlines, and when required, meetings were conducted to clarify and reinforce overall project scope and milestones.

2.4 Individual Contributions

2.4.1 Project Contributions

Cyrus Sadeghi helped design the Blips Web Server and Blips iOS Client, and was responsible for the majority of the implementations of both the server and iOS Client. He also had a hand in designing and building the Android client, helping with feature additions and bug fixes. He had a smaller part in helping with the touchscreen kiosk.

Jonathan Chan designed, implemented, and debugged the touch screen kiosk for the application. He also attempted to determine a method to present the the iOS client onto the Raspberry Pi for the kiosk.

Eliab Woldeyes mainly designed and implemented the majority of the Android client. He also helped in the design of the Web Server as well as the overall layout of both Android and iOS clients.

2.4.2 Report Contributions

Report Section	**Written By**
Abstract	Eliab Woldeyes
Chapter 1 - Introduction	-
1.1 Problem Motivation	Cyrus Sadeghi
1.2 Problem Statement	Cyrus Sadeghi
1.3 Proposed Solution	Cyrus Sadeghi
1.4 Accomplishments	-
1.4.1 Web Server	Cyrus Sadeghi
1.4.2 iOS Client	Cyrus Sadeghi
1.4.3 Android Client	Eliab Woldeyes
1.4.4 Touchscreen Kiosk	Jonathan Chan
1.5 Organization of the Report	Jonathan Chan
Chapter 2 - The Engineering Project	-
2.1 Health and Safety	Eliab Woldeyes
2.2 Engineering Professionalism	Jonathan Chan
2.3 Project Management	Jonathan Chan
2.4 Individual Contributions	All team members
Chapter 3 - Background Literature Review	Eliab Woldeyes
Chapter 4 - Technical Solution	-
4.1 Web Server	Cyrus Sadeghi
4.2 iOS Client	Cyrus Sadeghi
4.3 Android Client	Eliab Woldeyes
4.4 Touchscreen Kiosk	Jonathan Chan
Chapter 5 - Conclusions	Eliab Woldeyes

Chapter 3 - Background Literature Review

Intuition would tell one that the tourism and travelling industry is driven by people's need for a getaway. However, travelling uninformed or lacking information is potentially the biggest challenge faced by tourists. Due to this, facilitating simple tasks can prove to be difficult and ends up deterring travelers from revisiting a destination or even traveling in the first place.

Consider a scenario where a tourist is unfamiliar with the popular historical sites and attractions in their visited city. They are not sure what they would be interested in but do not want to see an attraction that is too far from their lodging. They are given a brochure at the front desk of their lodging showing main historical attractions, but there is no information about the attractions themselves. Once the tourist reaches the attractions, they find it was not very interesting and wished they could have previewed the site beforehand or even went to a attraction further away that would have been a better use of their time. This could have been avoided with a minimal, informative, guide of historical sites and attractions.

Now consider another scenario where a tourist is on a road trip. They are driving back home from their destination but it's getting late and they need to find lodging for the night. They have not pre-booked a hotel since they were sure that they could find suitable lodging off the highway. By chance, the first hotel they find is booked due to an event nearby. They go to the nearest hotel they can find but the prices aren't suitable to their budget. This process continues for a while before they eventually find lodging. This could have been avoided if the tourist could

have been notified of the event taking place or if they could remotely search for nearby hotels by price and availability.

Our original proposed solution aimed to resolve these issues through a simple mobile application.

Chapter 4 - Technical Solution

4.1 Web Server

As explained in the Accomplishments section, the web server's main goal is to be the primary source of information for all client applications (mobile clients and touchscreen kiosk).

The web server must be able to use the client's location and other querying attributes (such as attraction type, attraction user average rating, etc.) to return a list of attraction that match the client's request (for example, if a client asks for banks within 10 km of Ottawa's Downtown with a minimum user rating of 4.5 out of 5 stars, the server must return a list of banks that have a user average rating of at least 4.5 stars).

The final web server implementation is a Linux virtual machine hosted by Amazon's AWS. The Linux machine itself can be expanded to a number of machines as more client requests are made (this is automatically handled by Amazon's ELB system), however for the purposes of our project, we limited the number of server machines to 1. The Linux machine runs the server application, primarily written with Node.JS, with other performance critical components written with Python.

Incoming HTTP requests are received by Node.JS and then delegated to a module that was written to handle the specific request. As seen in Figure 4.1, the main Node module delegates

the client's request to either the attraction querying module or the client synchronization module.

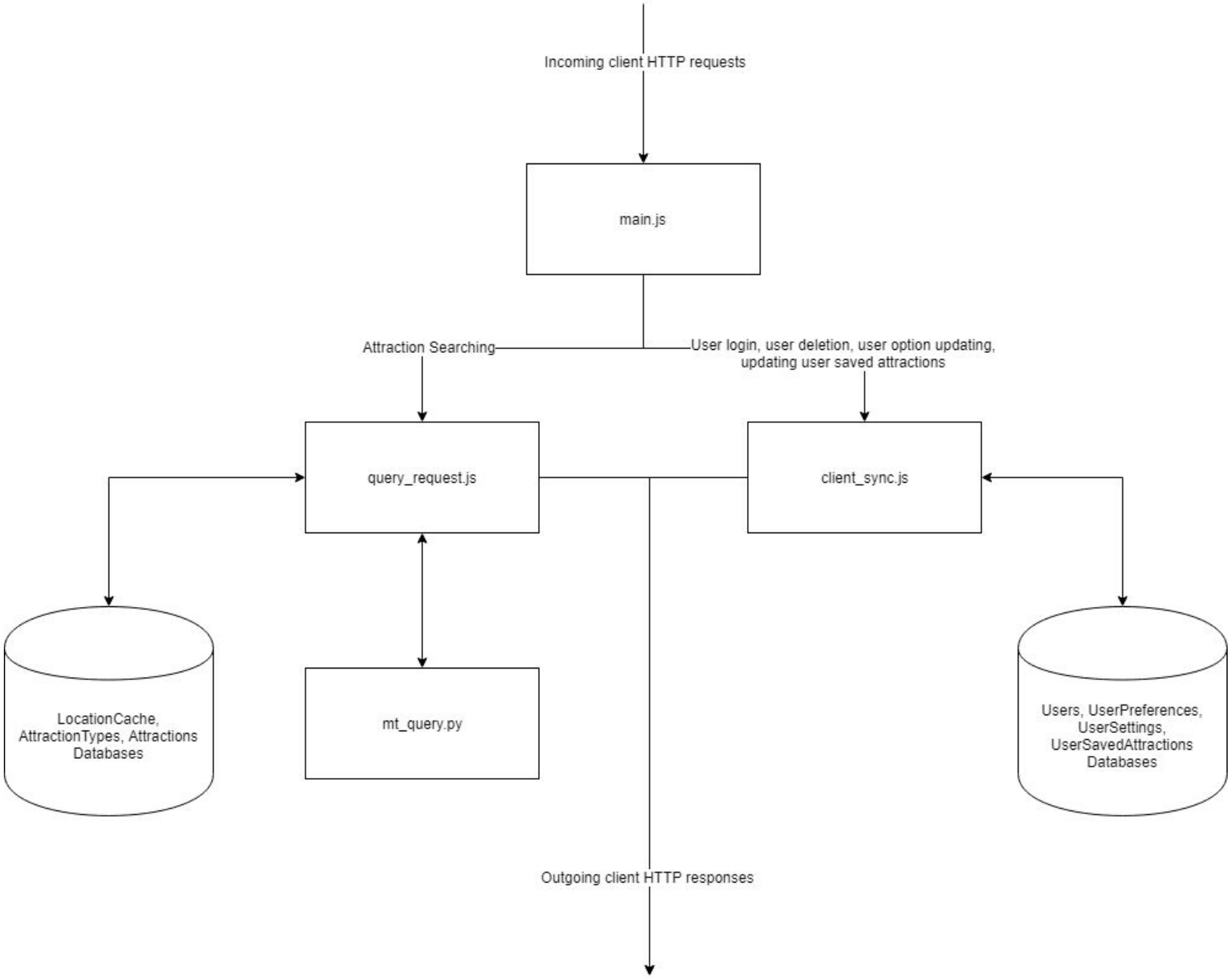

Figure 4.1: Server HTTP Request Delegation and HTTP Responses.

Incoming client HTTP requests are delegated to the proper module, where a response is formulated and returned to the client.

As seen in Figure 4.1, client attraction queries/searches are delegated to query_request.js. In first few versions of the server application, query_request handled the entire querying process. This process has seven steps, as seen in Figure 4.2.

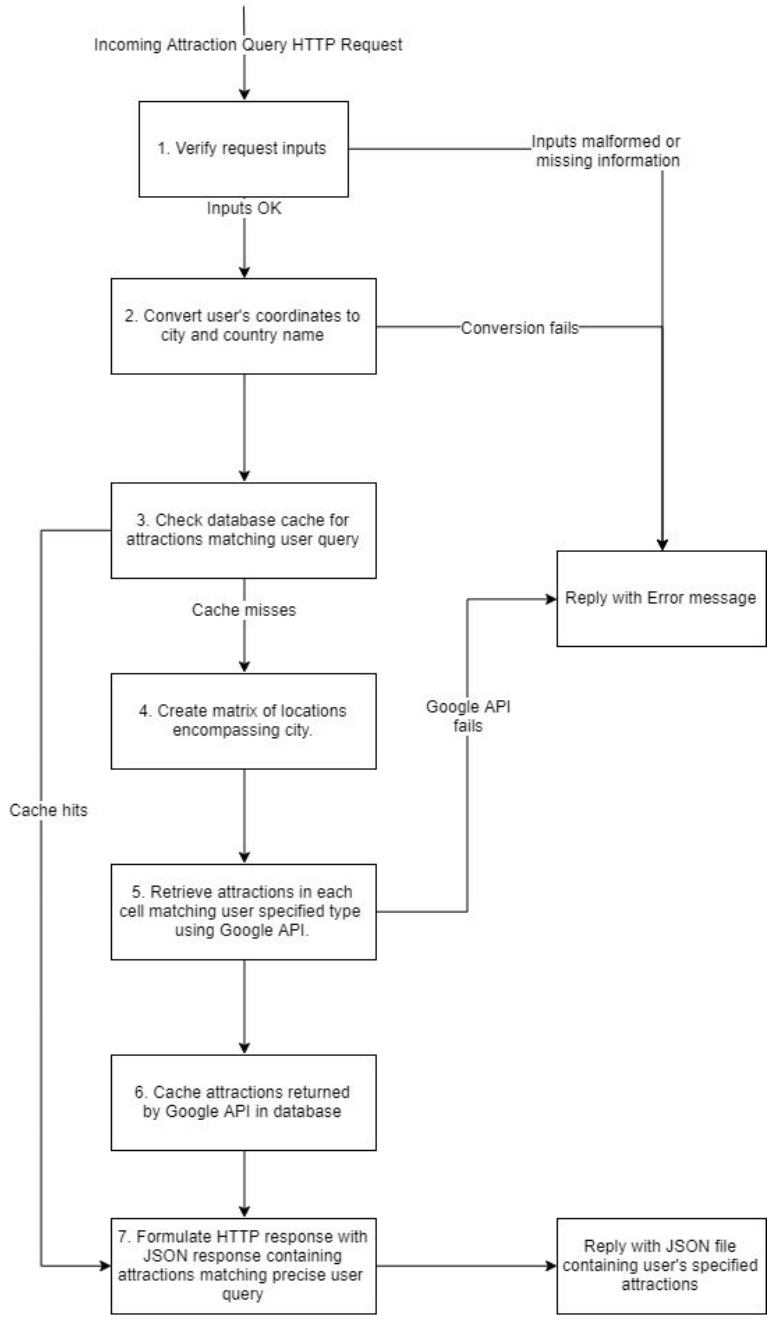

Figure 4.2: Seven step attraction querying process.

The first step requires validating the client request for all required information (i.e., client location, attractions to search for, search radius, etc.), and checking that no extra JSON tags were included in the request. If the client request is missing any information or is malformed in any way, the server replies with an error response (this error response translates to an error message that appears on the client device, this is detailed in section 4.3, iOS Client Technical Solution).

The second step is to convert the client's location (specified by the client as latitudinal and longitudinal coordinates) to a city name, country name, and coordinates for the city bounds (i.e., the four corners of the city required to create a virtual rectangle encompassing the city). This format is required for the next step. If the location conversion fails for any reason, the server replies with an error response, as in step one.

In the third step, the server checks its databases for the existence of a cache of attractions corresponding to the client's request. The cache system is setup to link each city and attraction type to a list of attractions (for example, Ottawa and Grocery stores is one cache entry, Ottawa and Banks is another cache entry, Toronto and Grocery stores is another cache entry). If a previous attraction query was made for the same cache entry (i.e. the same city and attraction type), the server skips to step seven, as the cache hit.

In the fourth step, the corners found in the second step are used to create a 2D matrix of smaller rectangles that encompass the entire city, as seen in Figure 4.3.

Cell (0, 0)	Cell (0, 1)	Cell (0, 2)	Cell (0, 3)	Cell (0, 4)
Cell (1, 0)	Cell (1, 1)	Cell (1, 2)	Cell (1, 3)	Cell (1, 4)
Cell (2, 0)	Cell (2, 1)	Cell (2, 2)	Cell (2, 3)	Cell (2, 4)
Cell (3, 0)	Cell (3, 1)	Cell (3, 2)	Cell (3, 3)	Cell (3, 4)
Cell (4, 0)	Cell (4, 1)	Cell (4, 2)	Cell (4, 3)	Cell (4, 4)

Figure 4.3: 2D matrix of cells encompassing an example city

Each cell has the same size, and together all the cells represent the entire city. The purpose of this matrix construction is explained in step five.

In the fifth step, Google API calls are made to create a list of attractions with the parameters defined by the client. In earlier versions of our server software, we encountered an issue with the Google API. When calling the Google API to create a list of attractions for an entire city, the API was limited to 20 results. We decided that returning 20 results would not suffice, our server must be able to return every attraction for every city. This resulted in the city matrix idea. Working around the 20 result limitation, we discovered that if we split the city into dozens of cells, we could query for attractions in each cell. Each cell would contain fewer than 20 attractions, circumventing the 20 result limitation. Once attractions were returned from each cell, the lists of attractions from each cell are then merged together, giving the server the entire list of attractions for the entire city.

With the list of attractions in hand following step five, the sixth step requires the server to cache the list of attractions in the database. As explained in step 3, caching the results allows for future requests to be much faster, with experimental testing showing that subsequent requests to be 2-3 times faster than the initial request.

The final step in the server query process is to filter the large list of attractions by the user's query options. These options filtered the list of attractions by distance from the client, minimum user average rating, and maximum price. The filtered list of attractions is then formulated into a JSON file and returned to the client.

As mentioned earlier, the first few versions of the web server application took an unreasonable amount of time (between 20 and 30 seconds) to respond to a client request that missed the cache (i.e. the client's request was the first of its kind, in terms of city and attraction type combination).

After some investigation by our group, we realized that step five of the server query process (the Google API call step) was consuming the majority of the processing time. Our initial implementation was done using Node.JS, with each Google API call being done serially for each cell. We decided to switch to a multithreaded approach that performs Google API calls for each cell in parallel. Node.JS does not support multithreading in any form, so our group was forced to rewrite our implementation in Python (this module was named mt_query.py, as seen in Figure 4.1).

The multithreaded Python approach severely reduced initial query time, reducing the 30 second Node.JS only queries to 5 seconds with a mixed Node.JS and Python query.

The other major requirement for the web server is the ability to maintain a database of users (detailed in Figure 4.4), including their usage history, client options, and a list of saved attractions.

As seen in Figure 4.1, any requests pertaining to users are delegated to the client_sync module, which handles requirements related to user accounts.

When a user logs into a client, a request is made to the server, and the server responds with the user's saved attractions, the user's client options, and their usage history. The user can also delete their account, removing any information pertaining to the user from the server's databases.

Everytime the user makes a change to their settings in the client, searches for attractions, or saves an attraction for later, a request is made to the server to update its databases to reflect the user's change. Keeping the server's databases synchronized with the information stored on the client allows the user to change devices and have their information move with them.

Each entry of the users database is linked to entries in all other databases (i.e. usage history database, client options database, and saved attractions database.)

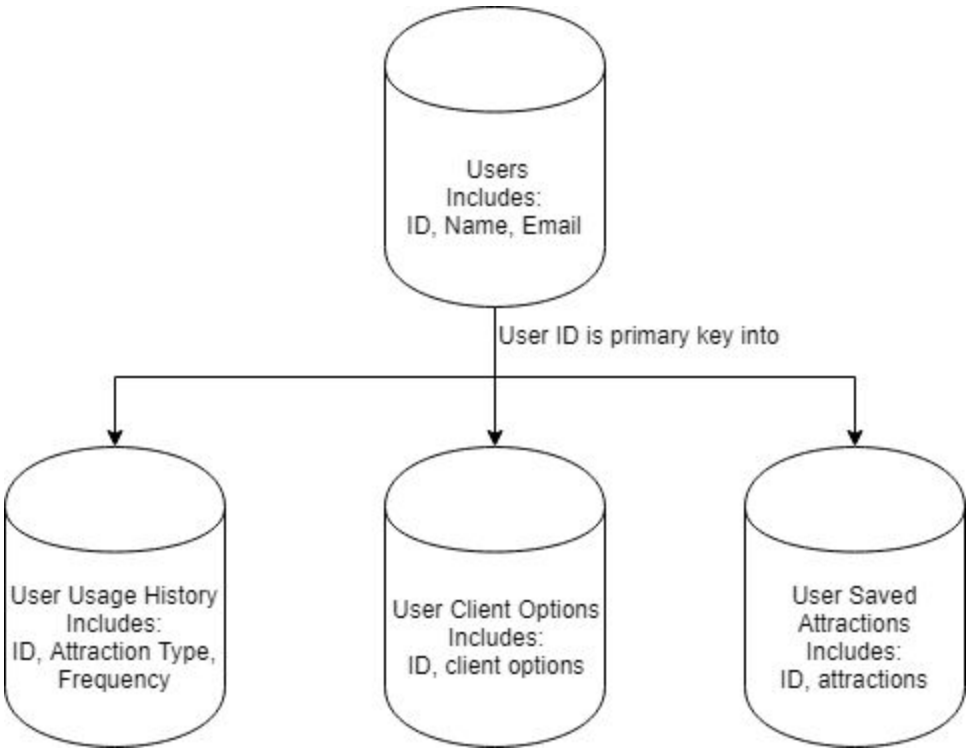

Figure 4.4: User database structure.

The primary database is the Users database, containing unique user IDs for each user, as well as the user's name and email address. The other databases use the unique user ID as the primary key for their entries.

4.1.1 Non-Functional Requirements

Responsiveness was the most important non-functional requirement for the web server. When initially developing the web server, response time for requests was unreasonably high (i.e., the server would take upwards of 20 seconds to formulate a response for the client). Through a number of re-designs, response time was reduced to approximately 2.5 seconds (as explained earlier, the server moved from a single threaded implementation to a multi threaded implementation).

Security was a big concern when creating the web server. Since the server saves user information (such as their name and email address), as well as user preferences (such as what types of attractions they visit most often), the server must prevent information from leaking to malicious actors. To solve this issue, our web server and mobile clients use Google's OAuth API to securely handle sensitive user information.

Availability was the final non-functional requirement for the web server. In our scenario, the server must always be available to handle a client request. The server should never be in a state that prevents client requests for being serviced (for example, the server should not be offline, the server application should not be in a crashed state, the server's database must also adhere to the same availability requirements). When first developing the web server, this requirement was not achievable (occasionally, the server would lose its connection to the database, or the

database itself would drop all tables containing attraction information). Achieving high availability required extensive testing and bug fixing to prevent the server from ever crashing.

4.2 iOS Client

As explained in the Accomplishments section, the iOS client's main goal is to provide the user with a simple interface containing attractions near their location. The application should also provide a method to manually search for attractions, allowing the user to specify specific attraction types, search radius and other options. The iOS client must work in tandem with the web server to provide the user with attractions.

The iOS client also needs to display attraction history for prominent locations, for example, the CN Tower or Parliament Hill must be displayed with additional information describing their importance. Other important details include pictures of the attraction, a method of getting directions to the attraction, and a method to save attractions for a later purpose (for example, saving attractions in a different city, allowing the user to recall attractions they wish to visit if they travel to the city).

The final iOS client implementation is exclusively written using Apple's new programming language, Swift. Swift is a multi-paradigm programming language (i.e., the language supports object-oriented programs, functional programs, etc.) primarily used to write software for Apple devices (for both iOS and macOS). When first developing the application, our team had no experience with developing iOS applications. Very soon after figuring out the design for the

application, we realized that Swift and iOS development in general follows the very popular MVC pattern of software development, where M stand for Model (model classes contain all logic and control flow of the application, all data manipulation and server interfacing is done through these classes), V stands for View (each visible section of the application is rendered through a view class. View classes contain user interface elements (such as buttons, text fields, text labels), and the C stands for controller. Controller classes control the content of view classes. Controller classes provide the link between model classes and view classes. They use the information provided by model classes to set the user interface elements defined in the view classes.

For example, the main map, as seen in Figure 4.6, is made up of numerous views, the menu, as seen in Figure 4.5, is made up of views, and the user account menu, as seen in Figure 4.7, is also made up of numerous views.

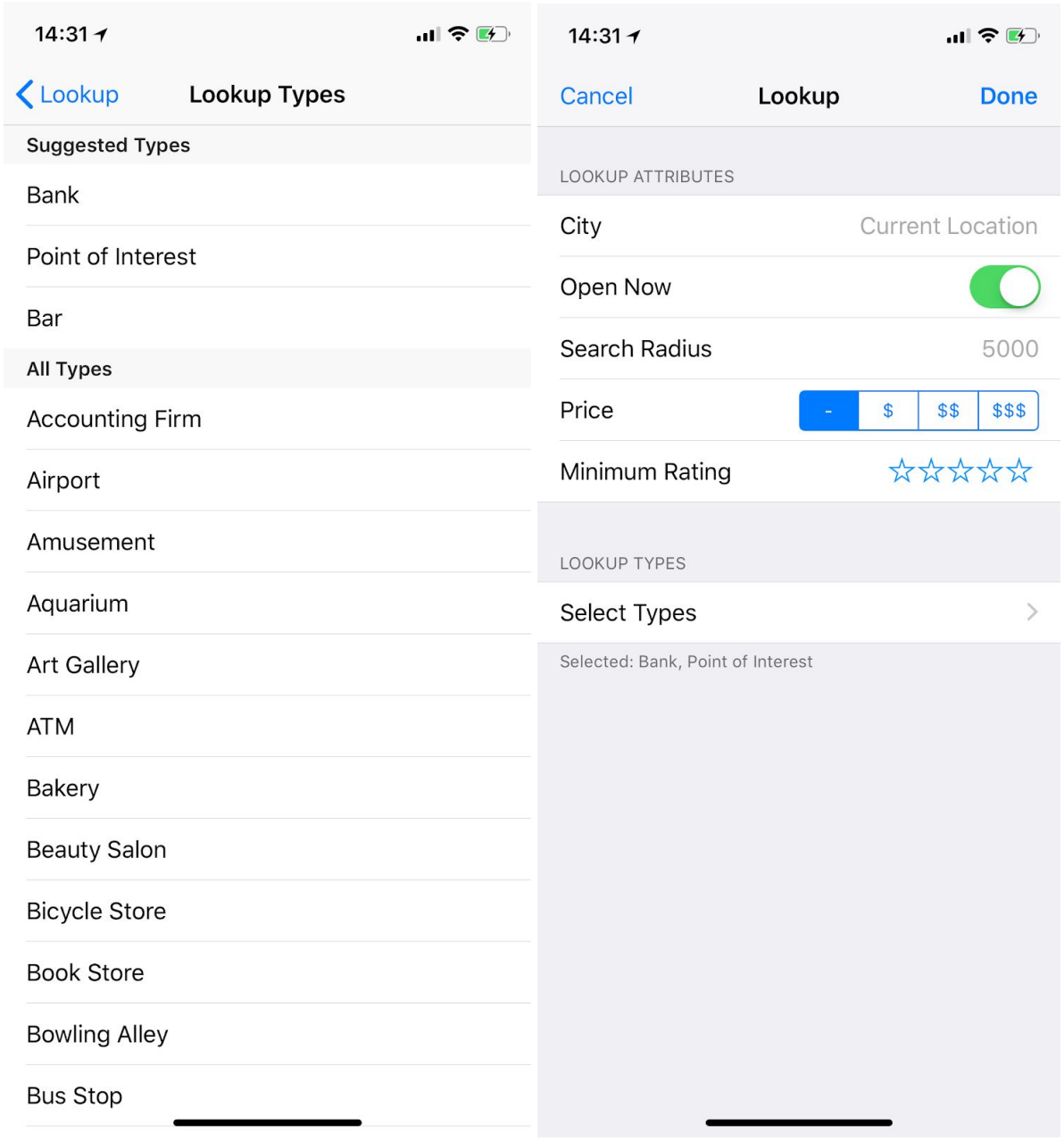

Figure 4.5: Custom Lookup Menu. Left: Attraction Type Selection Right: Lookup Options

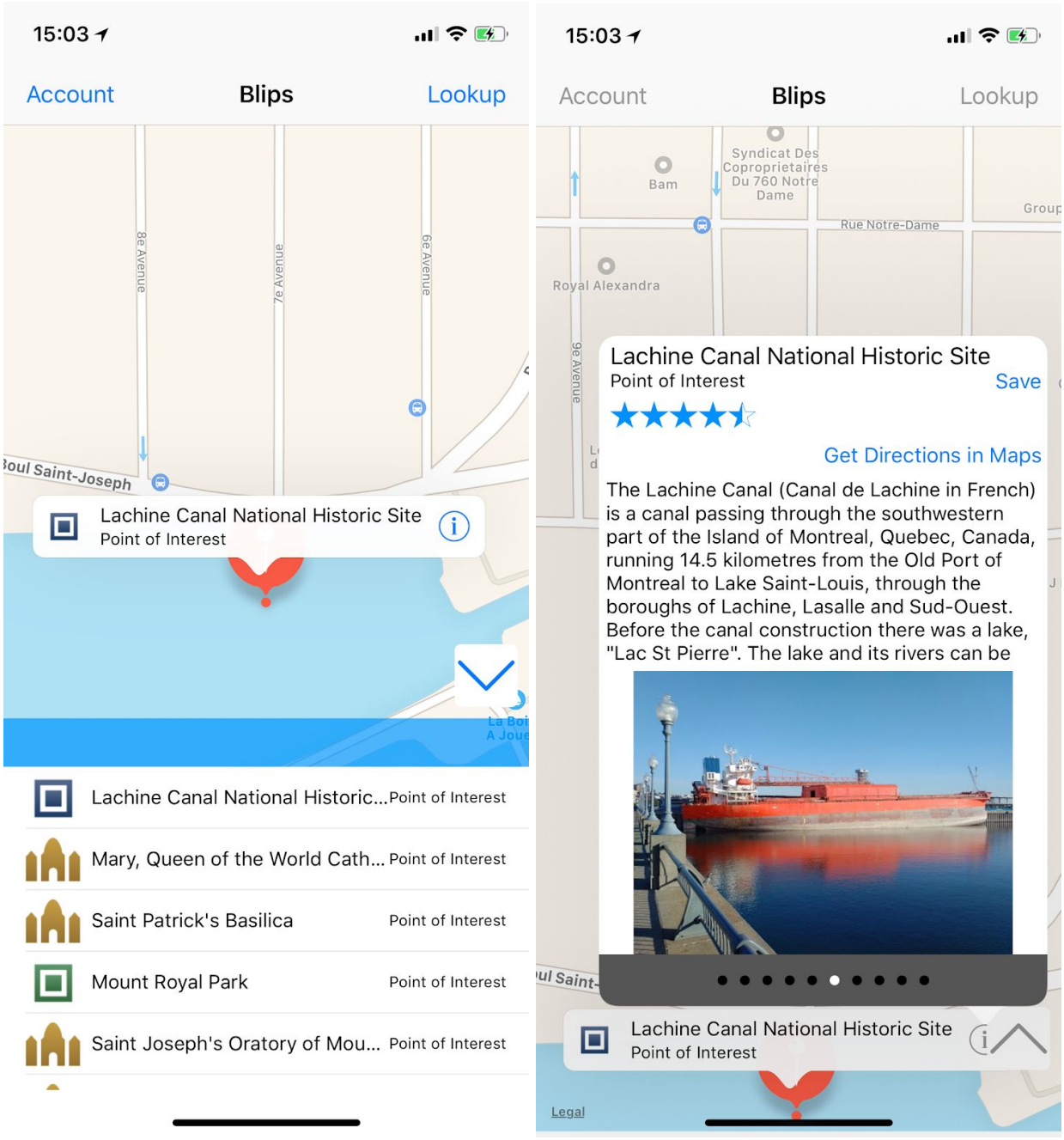

Figure 4.6: Map and Attraction Interface.

Left: Map interface after searching for Points of Interest in Montreal. Right: Map and Attraction Interface after selecting a specific attraction. The attraction's description, an album of photos, the average rating, a button to get directions, and a button to save the attraction are available.

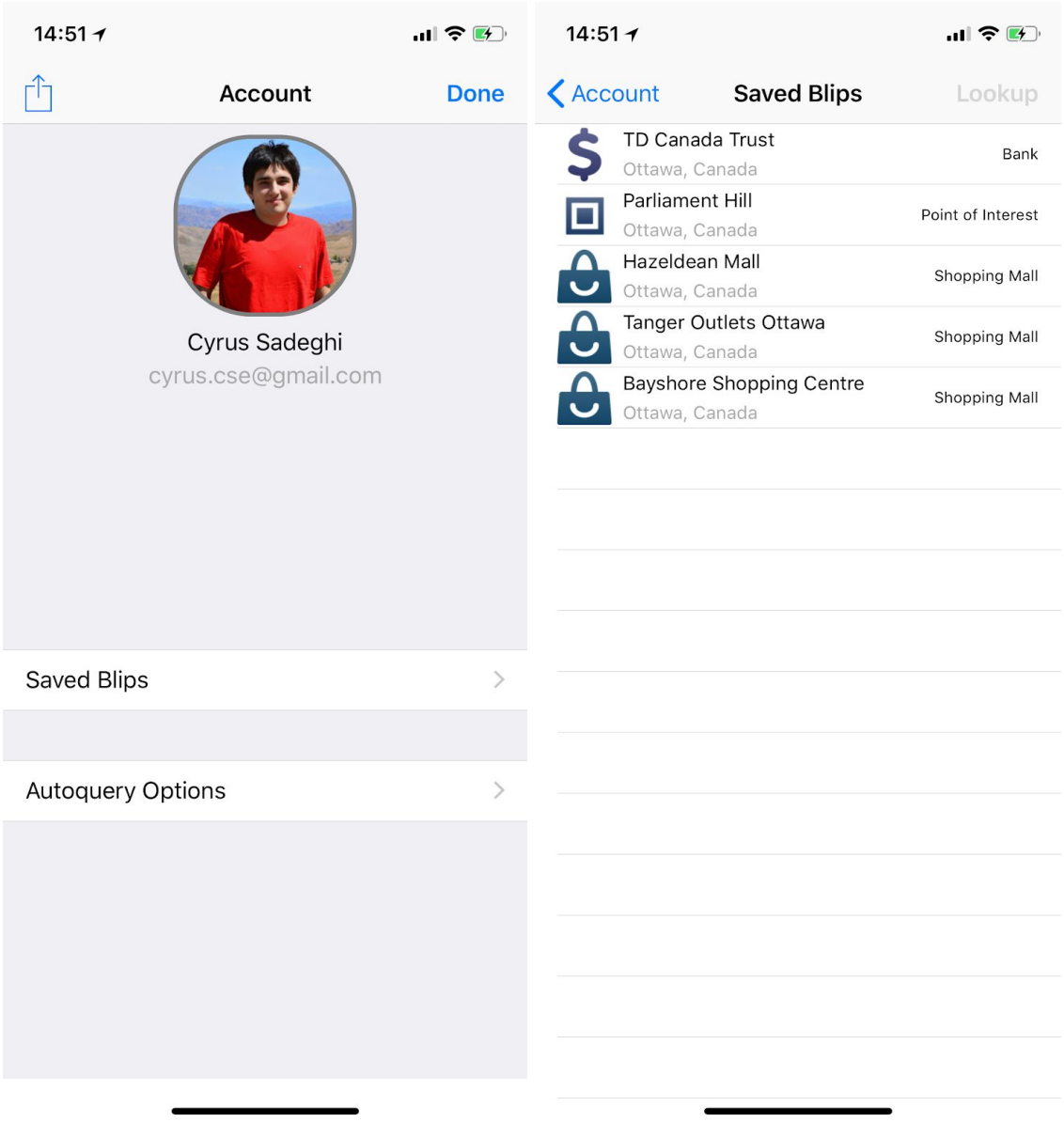

Figure 4.7: User Account Menu. Left: Account Summary Right: Saved Blips/Attractions

The iOS client includes four main functionalities. The first functionality our team implemented was the custom functionality (screenshots of the Custom Lookup menus are available in Figure 4.5).

In the first versions of the client, custom lookup was the only method of querying the server for attractions. The client required the user to select a number of attraction types (for example, points of interest, banks, book stores, etc.) and specify a search radius. The client would then use the user's location and inputs to formulate a request for the server. As explained in the Web Server Technical Solution section, the server would formulate a response containing a list of attractions which the client would show on a map. This process is shown in Figure 4.8.

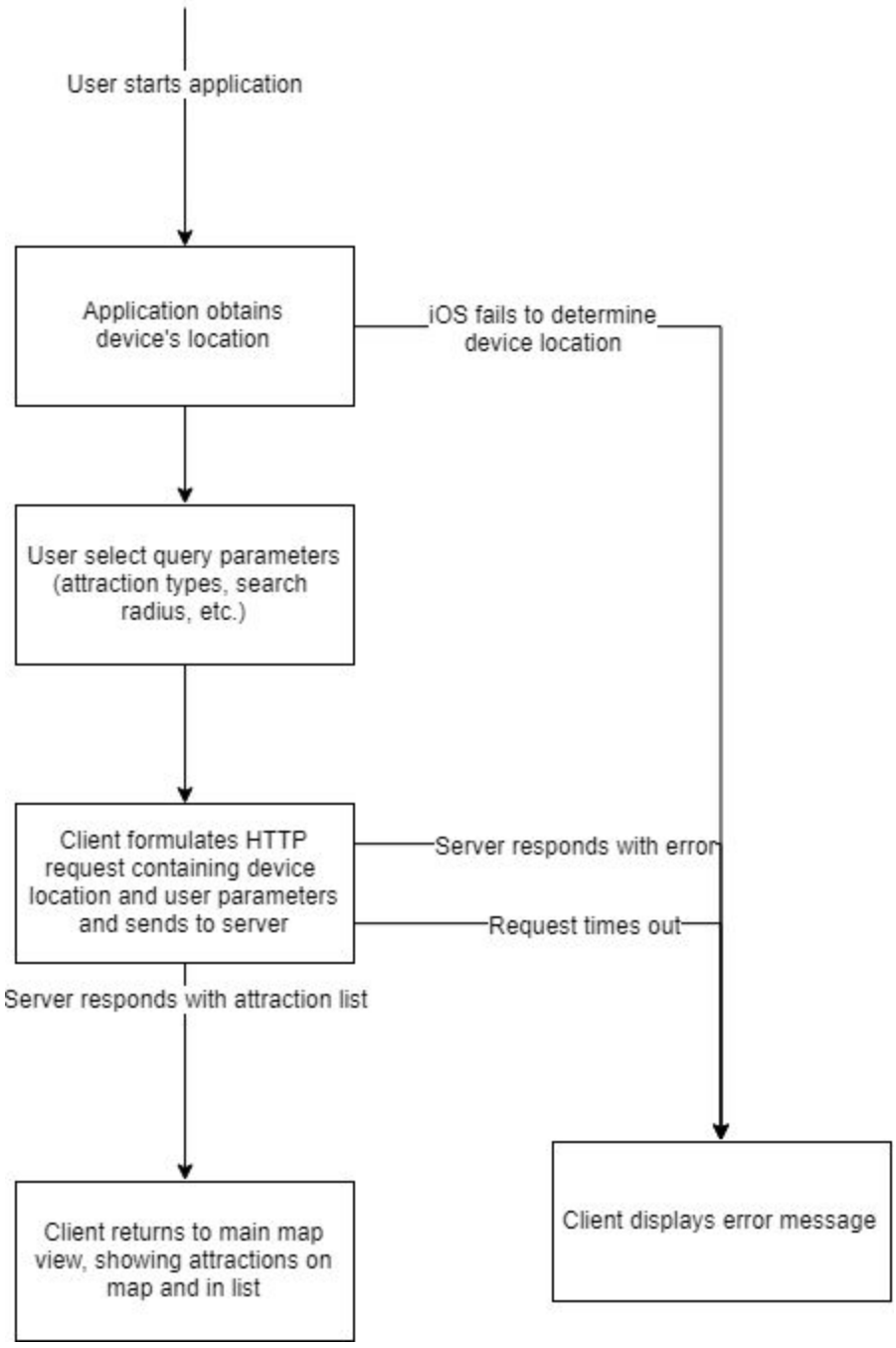

Figure 4.8: iOS Client Lookup Procedure.

The client utilizes the device's location and user's input parameters to display a list of attractions on the client's map.

In later revisions of the iOS client, our team added a field allowing the user to select a specific city to query in. This allows the user to either search for attractions in close proximity to their current location, or search for attractions in other cities. Having this feature also allowed the application to be used on devices that do not support location services or on devices that have location services blocked by the user.

The second feature our team implemented was the client side user account system. This account system complements and works with the user account system implemented in the web server (as detailed in section 4.1).

As our team was gathering feedback and further requirements from friends, family and industry experts, we received requests for a few new features. Specifically, the ability for the mobile application to suggest attractions to the user based on their usage of the application. For example, if a user tends to frequently search for restaurants, the application should suggest restaurants.

Our team implemented this feature by tying the user's search history to the user account. Whenever the user queried for any type of attraction, the client would keep track of what attraction types they searched for. This information is then stored in a table on the client (which is also synchronized to the server), with an example seen in table 4.1.

Attraction Type	Request Frequency
Point of Interest	43
Bank	32
Bar	25
Shopping Mall	15
Restaurant	13

Table 4.1: An example user's usage history.

The client (and optionally server) keep track of what kinds of queries a user makes, and how often they make those queries. In table 4.1's example, the user queried for points of interest 43 times, banks 32 times, etc.

The next time the user makes a custom query, the application then suggests the types that the user queried the most frequently. The attraction types are sorted by request frequency, as seen in Figure 4.9.

Should the user be logged into the user account system, this information is also synchronized with the web server (as detailed in section 4.1). This allows the user to switch devices and have their recommendations follow them.

Later, when asking for additional requirements for the application, a friend noted that there was no method of clearing a user's history, limiting the user's privacy. Our team then implemented a method of clearing a user's history from both the client and server.

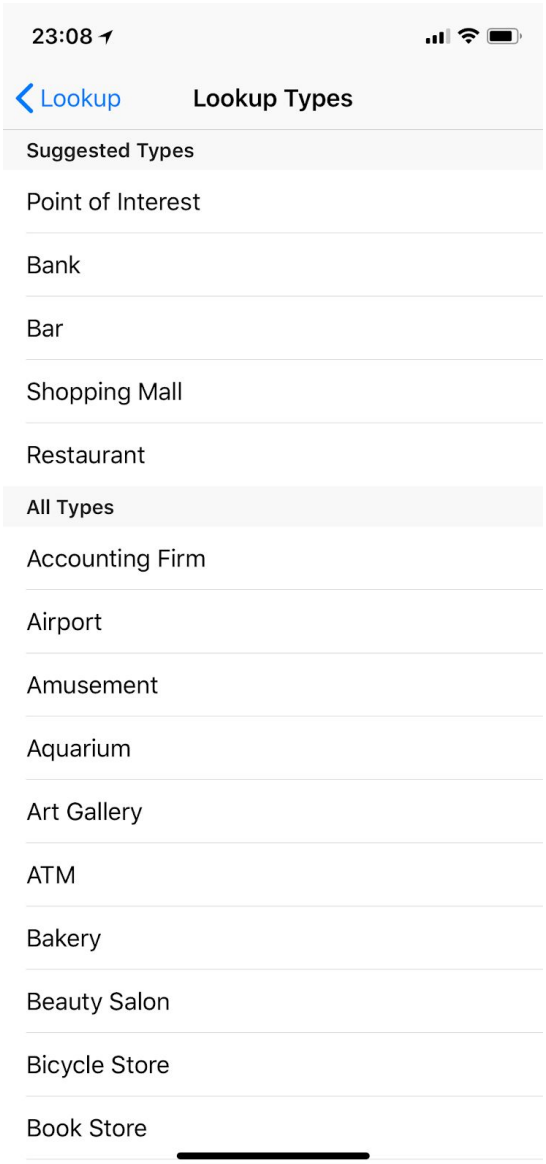

Figure 4.9: iOS Client Attraction Type Suggestions

The application suggests attraction types, corresponding to the example user's usage history from table 4.1.

The third feature our team implemented was the automatic querying functionality. In earlier versions of the iOS client, the only method to search for attractions was to perform a custom lookup, as seen in figure 4.5. After asking friends and family for advice on how to improve our application, they suggested adding a second method of querying. This second method shouldn't require any user input and should provide the user with some suggested attractions when they start the application.

Our team decided to use the usage history gathered from the custom lookup functionality to provide automatic querying. When the user launches the application, the client checks if the user has previously used the application (i.e., if the user has any of the information presented in table 4.1). If they do, the client uses the top 3 attraction types (this is configurable, the user can change how many attraction types the automatic query process uses, as seen in figure 4.10) and automatically queries the server for attractions. These attractions are then placed on the map, in the same fashion as a regular custom lookup.

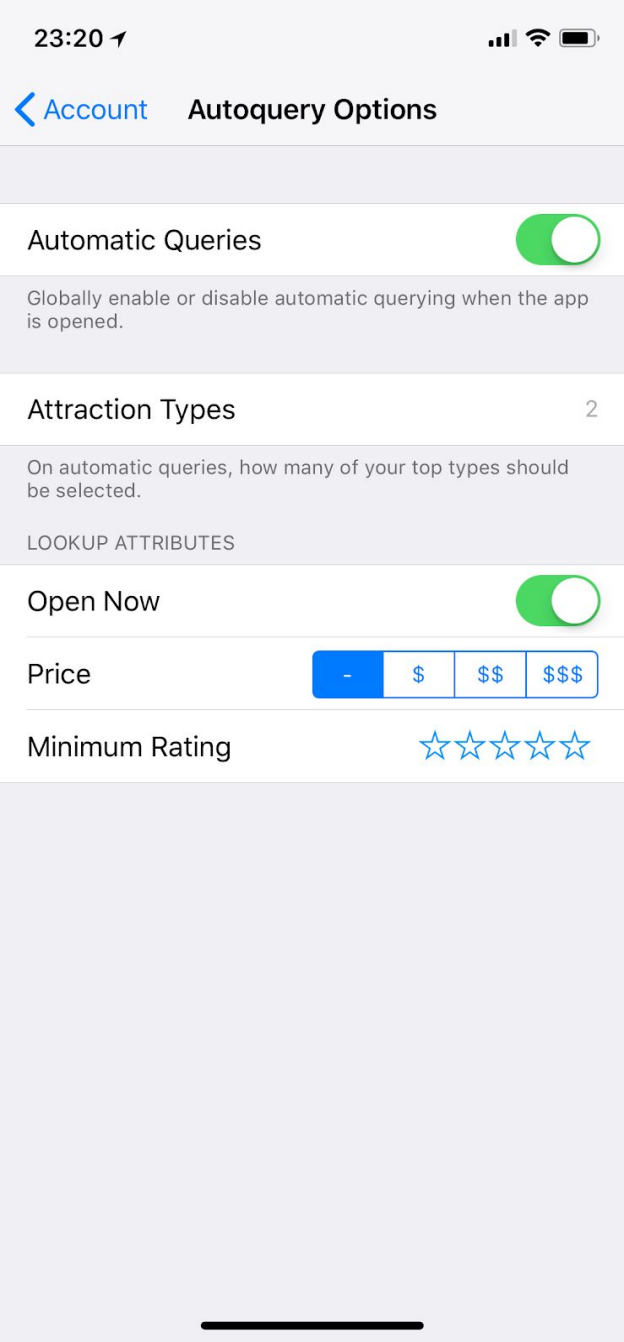

Figure 4.10: iOS client automatic query options.

The user can globally disable automatic querying, change the number of top attraction types to query for, and specify automatic lookup filters.

The final main feature our team implemented in the iOS client was the list of saved attractions. This feature was also added as a result of requirements gathered from friends, family, and industry experts. The saved attraction list allows the user to save specific attractions to their user account for retrieval later. As noted by friends and family, they wanted a way to search for and save attractions in a city that they wish to visit in the future, allowing them to have a list of attractions to visit once they arrive in the city (as seen in figure 4.11).

When a user makes a query, each specific attraction can be selected and the attraction details contain a button allowing the user to save the attraction (as seen in figure 4.6, right). When the attraction is saved, it is added to the list of the user's saved attractions (as seen in figure 4.7, right).

Attractions in the saved Blips list can be selected and displayed on the map, allowing the user to visually see the attraction's location, as well as the photos of the attraction, and the attraction's average user rating. This is seen in detail in figure 4.6, right.

Should the user be logged into a Google Account, their list of saved attractions is also synchronized to the server. If they log into the application on another device, their list of saved attractions is downloaded from the server. The user can also easily clear their list of saved attractions, should they be worried about privacy concerns.

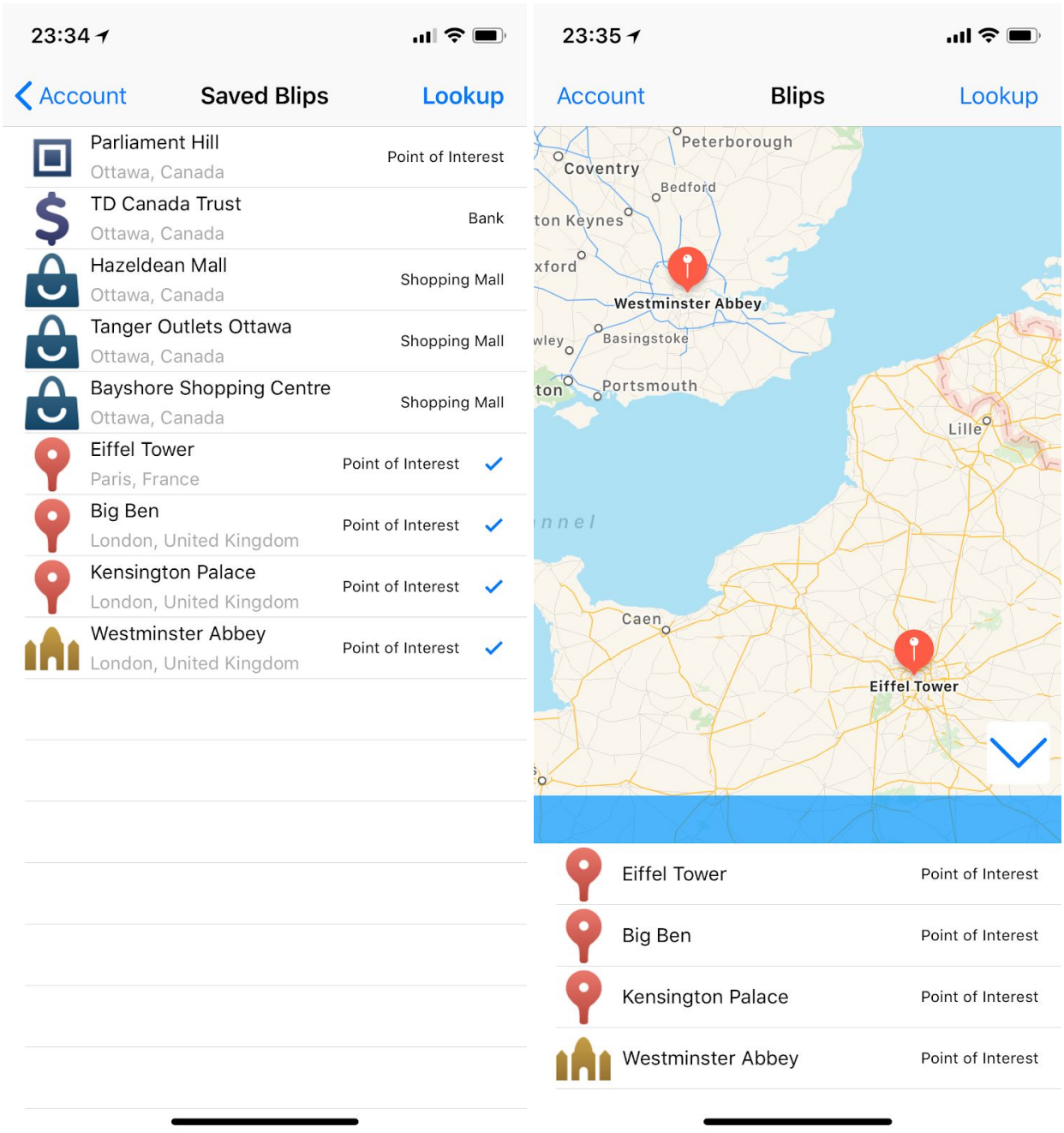

Figure 4.11: iOS Client Saved Blips Example.

The user has already saved attractions to their list (as seen on the left), and after looking them up (as seen on the left), they can see the attractions on the map.

4.3 Android Client

Similar to the iOS client, the Android client is one of the main points of interaction with the system, only targeted for a different platform. From the user's perspective there should be little to no difference in functionality between iOS and Android clients. Therefore, of those that were completed, the functional requirements are the same. The implementation of the design meeting these requirements was done using Android Studio IDE, which follows an MVC architecture. Additional information on Android Studio is outlined in Appendix B. The code for the implementation of the Android client can be found in Appendix A.

The core of the solution from the mobile client perspective, is to allow the user to search for their desired attraction types. The Android client should let the user easily navigate to a manual lookup page and see the available attraction types to pick from. To support this requirement, the main accomplishment of the Android client is allowing the user to make a custom lookup of attraction types along with optional search parameters. The functional process of a lookup was designed following the flowchart in Figure 4.12 (similar to the iOS procedure in Figure 4.8). The entry point of the lookup is through a toolbar action button from the main page of the client, which is the map view (denoted by the black circle). When a failure occurs, the lookup ends (denoted by the black and red circle). This failure can be due to a network error, or a user input error. Depending on which, a suitable message is shown and the user can then try again or cancel the lookup. Once the search result is returned from the server, that location data is passed back to the map view.

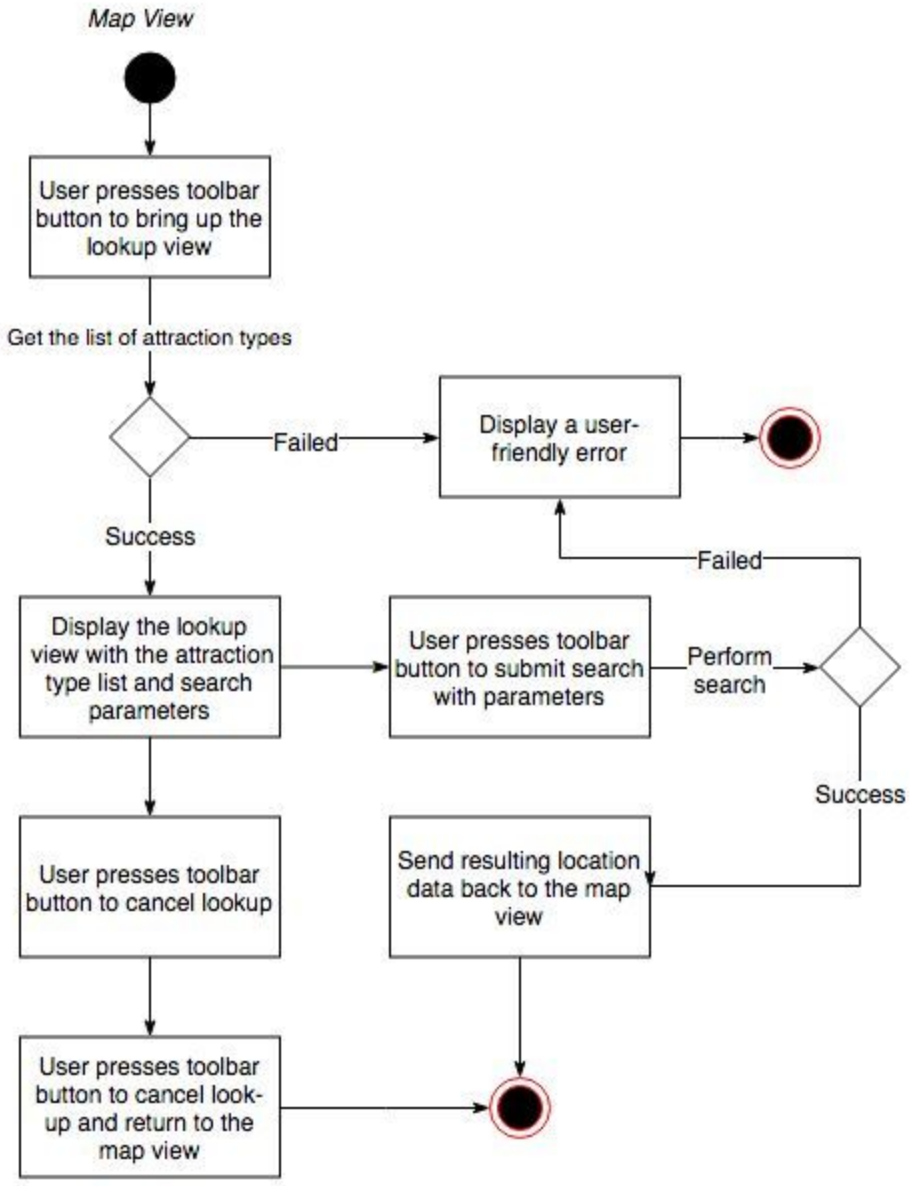

Figure 4.12 - Flowchart design of a lookup.

The implementation of this design involves two key elements, the map and lookup pages. The base of the implementation is the map. The map activity class (BlipsMapActivity.java) is the model and controller of the map view. It links to the map activity resource file (activity_blips_map.xml) that contains all the view components and their layout. Whenever the map view is displayed, the map activity calls its onCreateFuction() that assigns functionality to view components; thus assuming the MVC model and controller roles while the resource file assumes the view role. The lookup page follows the same implementation structure (defined in LookUp.java and activity_look_up.xml respectively). The map view is shown in Figure 4.13 while the lookup view is shown in Figure 4.14.

Figure 4.13: Android Map View

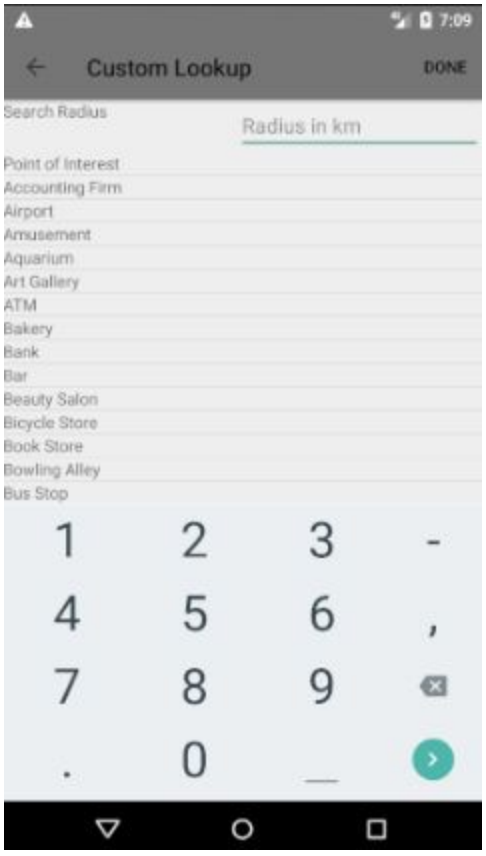

Figure 4.14: Android Lookup View

The main method of interaction between the activities is through toolbar actions. The main map activity allows the user to perform the search with the “Lookup” button (shown in Figure 4.13 above). Upon pressing it, the lookup view is generated; sending a request for a list of attraction types and displaying the result. After pushing the “Done” button (shown in Figure 4.14 above), the lookup activity formulates the user’s request based on the search radius parameter and the selected attraction types (current implementation simply displays the selection to the user). There is also a back navigation arrow that cancels the user’s search and

returns to the initial map view without regenerating. This implementation is shown in the sequence diagram of Figure 4.15 below.

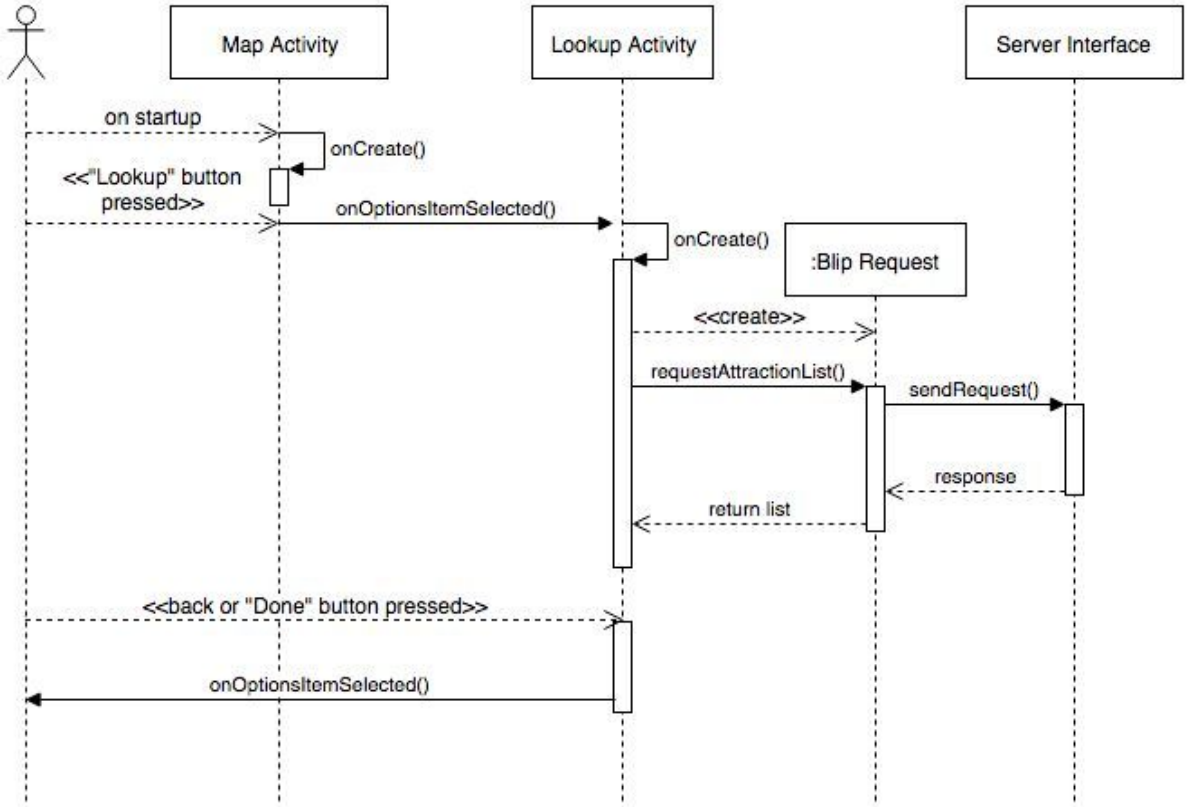

Figure 4.15: Lookup Sequence Diagram

The Android client interacts with the web server through asynchronous network requests. The team initially attempted networking capability using a task thread for every request. But the Android JDK requires a very convoluted process to send and receive asynchronous calls. This is due to the strict limitations on multithreading imposed by the Android OS since the user interface thread can easily be affected; hindering responsiveness. In response to this challenge, the group decided to use the open-source library, Volley, to handle networking functionality.

With this design alternative, all requests can be self-managed by Volley and the Android client can take full advantage of Volley's out-of-the-box network accessing. Now when a network request is being made, the activity formulates the request object with a callback function to handle the response. The server interface then utilises Volley's capabilities to access the remote server and asynchronously receive the response in the callback function. Depending on the implementation of the assigned callback function, the response is parsed and packaged in a form appropriate to the activity. This general process is demonstrated in Figure 4.16, below. For more information on the Volley library, refer to Appendix B.

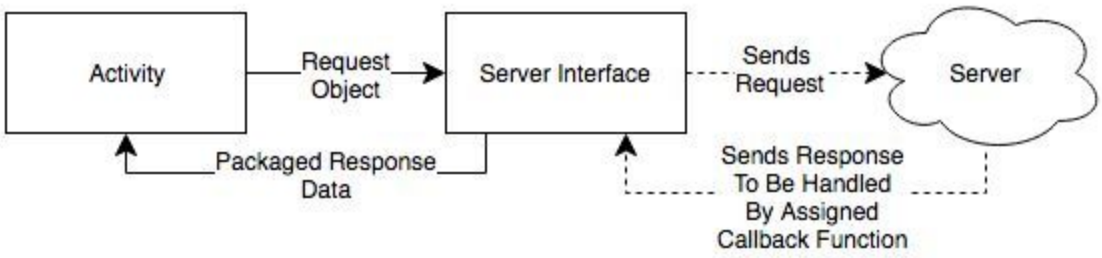

Figure 4.16: General Networking Functionality Process with Volley

4.4 Touchscreen Kiosk

To implement a touch screen kiosk for the application, the development of a touch screen apparatus was required to enable touch on the monitor where it would be displayed. The engineering requirements can be seen in table 1 of link “Kiosk supplementary document” in appendix A.

The technical solution for the touch screen apparatus included 2 parts, a vertical array of infrared emitters on one side and a vertical array of infrared transistors on the other, while the area in-between the columns is the effective touch screen area, this is illustrated in a simulation that can be seen in figure 1 of link “Kiosk supplementary document”. Figure 1.1 of the same document illustrates the general operation of the emitters.

The main idea is to have only one emitter ON at any given time, allowing the sensors to determine if they can detect the current emitter. If a sensor does not detect the current emitter then it can be interpreted as a blockage from an object. After cycling through all emitters back to the originating emitter where a sensor detected a blockage, the detector will attempt to determine an intercept if there are a minimum of 2 blocked sensors. For example, when emitter 2 was ON, some of the sensors had blockages, so the cycle will end back at emitter 2. If there are multiple different intercepts, since the first goal is single touch, the crude method used was to determine the average of all of them.

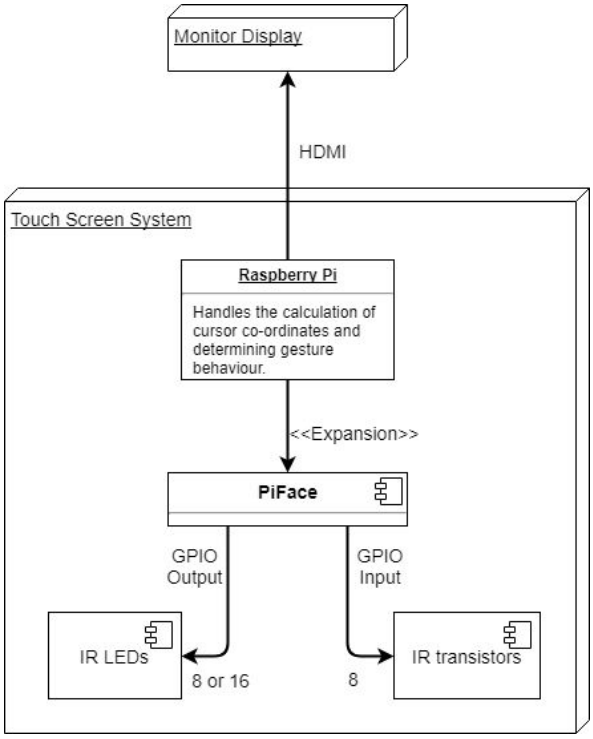

Figure 4.17: Touch Screen System Architecture

Figure 4.17 illustrates the system's architecture. A Raspberry Pi 3 was used as the computer for the touch screen system due to the device's ability to run its own Operating System. The Operating System contains a driver to control a mouse, and also has general purpose input/output, GPIO, readily available.

To isolate the inputs and outputs, while protecting the Raspberry Pi from unintentional damage, a PiFace digital 2 was used as an expansion over the Raspberry Pi's pins. The PiFace allows for 8 input pins and 8 output pins. The final configuration has each input connected to a sensor circuit containing 1 infrared transistor, making a total of 8 sensor circuits, and each output is connected to an emitter circuit containing 2 infrared emitters.

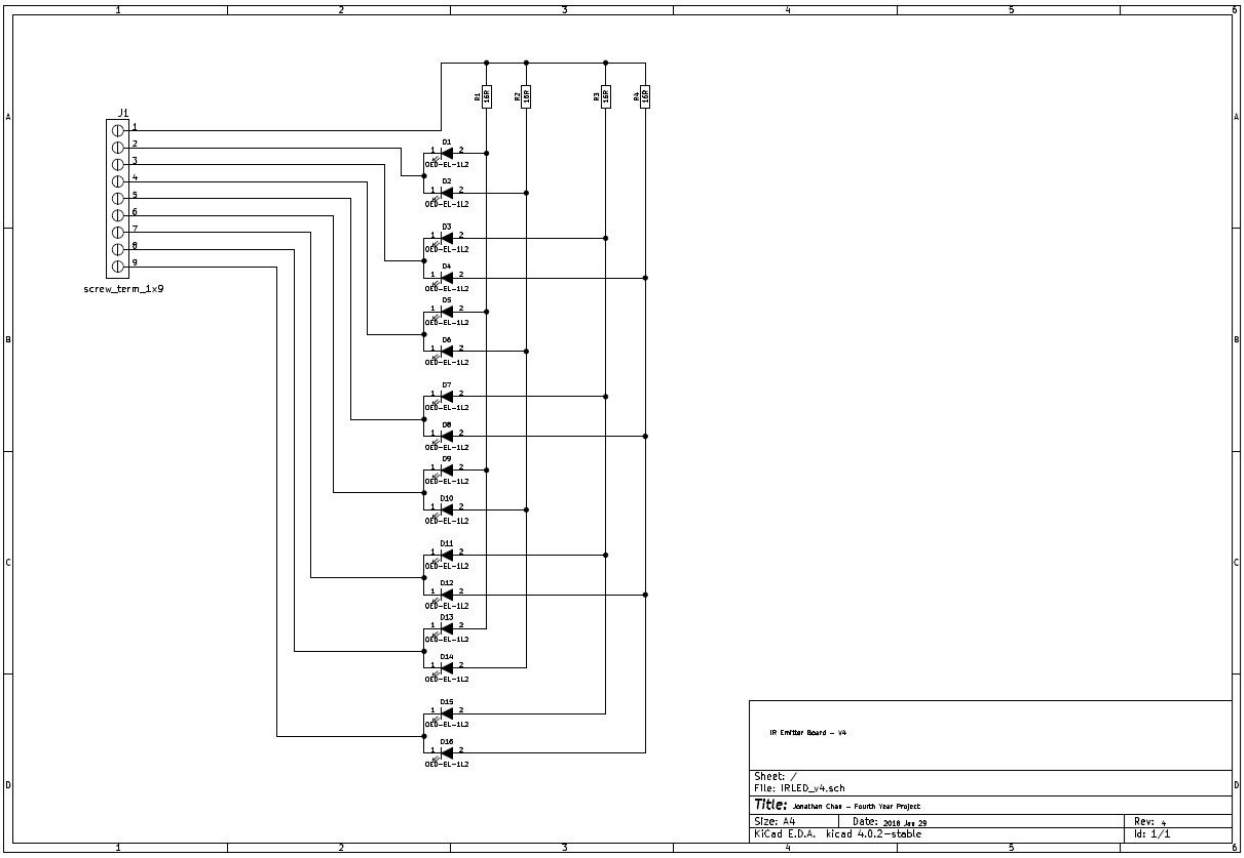

Figure 4.18: Emitter circuit schematic

Figure 4.18 contains the schematic of the emitter portion. Figure 2 of link “Kiosk supplementary document” contains the actual bread boarded circuit. It was the final result of multiple iterations due to empirical range limitations of the infrared emitter. Initially the effective range of the component calculated with square law was sufficient for this project, but when tested by applying max steady current, it was not the case. Using 2 infrared emitters in parallel proved to be the best solution and allowed for combined radiant strength to increase range.

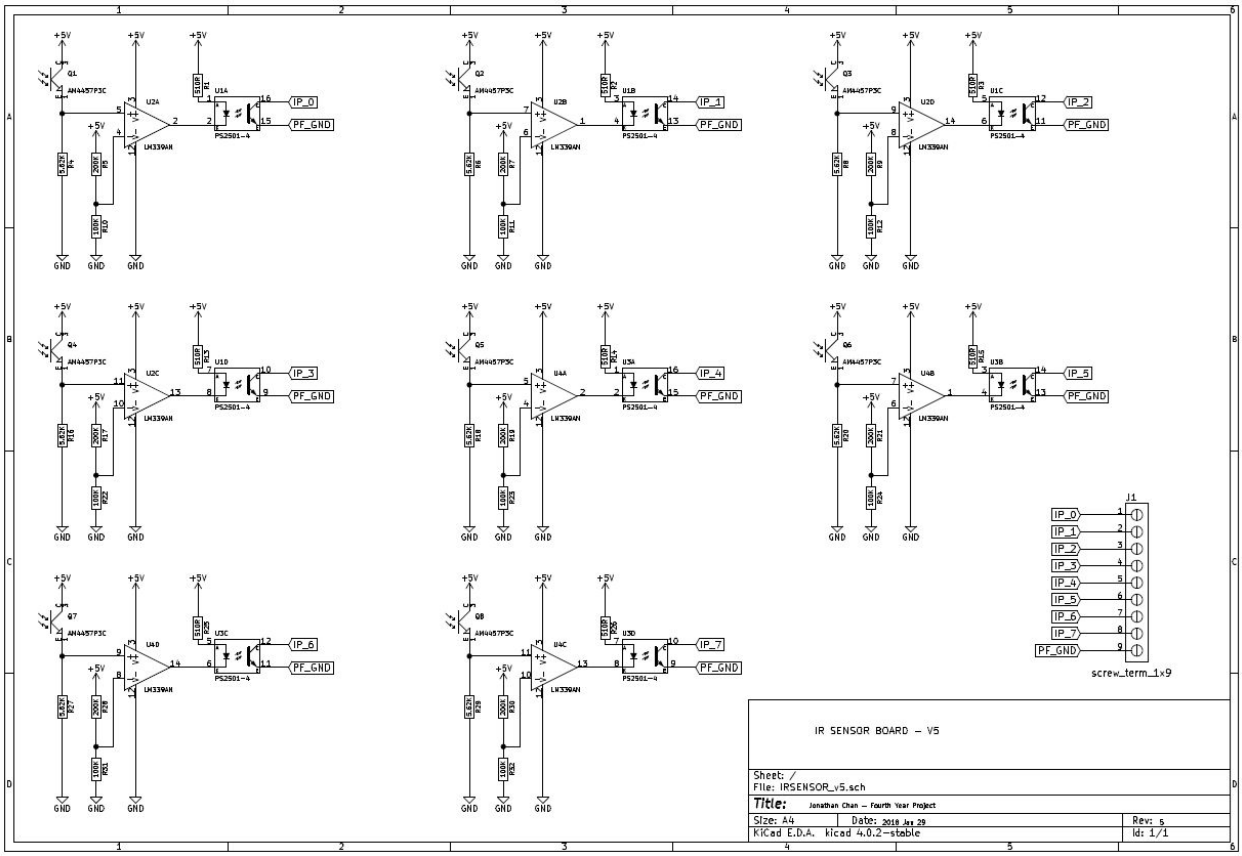

Figure 4.19: All 8 Sensor circuits schematic

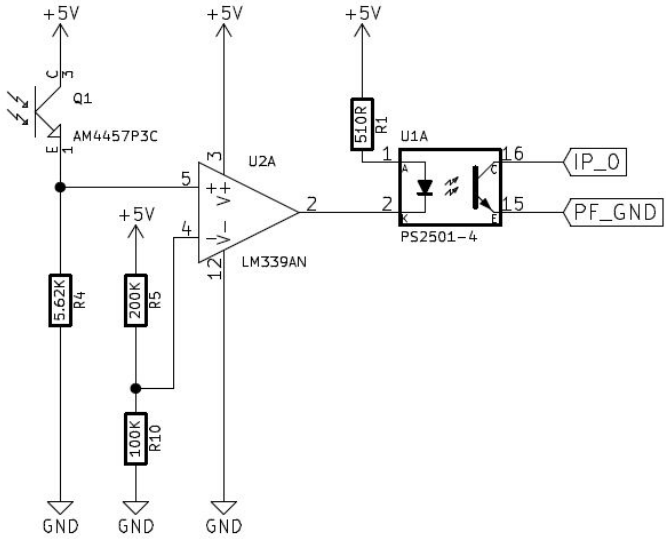

Figure 4.20: Single sensor circuit schematic

Figure 4.18 contains the schematic of the sensor portion. Figure 3 of link “Kiosk supplementary document” contains the actual breadboard circuit, figure 3.1 of the same document contains the overall touch screen overlay.

Its purpose was to detect the infrared, convert it to a digital value, then send the digital value to the board’s input through an optical isolator. An infrared phototransistor was used, because by construction it is less affected by visible light. The reason to convert the analog voltage signal from the phototransistor to a digital value was because there were two desired states, detected and blocked. To do this a voltage comparator was used and in this configuration it, a single threshold value was used where the digital output transitions; if the analog input was greater than the threshold voltage it was pulled to rail, in this case 5 V, and if the analog input was less than the threshold voltage was pulled to ground, 0 V.

An alternative method to determining the digital transitions was to use a hysteresis which used two threshold voltages to create a range where no change takes place, demonstrated in “Section A.1 Hysteresis add-on” of link “Kiosk supplementary document” in appendix A. It was actually implemented on the constructed circuit but did not add to the circuit’s performance because the analog signal was not very noisy or jittery around the threshold voltage to negatively impact the digital output. If the analog voltage signal was very noisy around the single threshold voltage, the output would frequently switch and an inconsistent message would be sent to the microcontroller. An optical isolator was used to send the signal to the piFace input and is a device that uses an optical transmission, visible light or infrared emitter

paired with a transistor, to transfer an electrical signal between separate circuits. This was mainly used to offer a safe interface between high voltage components and low voltage devices. In this case, the comparator will either output 0V or rail (which is 5V). The testing methods of the hardware portion are tabulated in table 2 of link “Kiosk supplementary document” in appendix A.

For the software components, the activity, sequence, and state diagrams to control the hardware is presented in figures 4, 5, and 6 respectively of link “Kiosk supplementary document” in the appendices. Controlling the emitters’ output and sensors’ input in general operation had to be sequential, where only one emitter is ON at a one time and then all the sensors’ inputs are checked. For example, emitter 1 turns ON and then all the sensors check if there any broken lines; emitter 1 turns off and emitter 2 turns ON and then all the sensors check if there any broken lines; etc. When a sensor did not detect the current emitter, it would record that line as a blockage and at the end of a cycle it calculated if there was a valid intercept with other records. The rationale of having the emitter control and sensor control managed by separate threads was due to a tested alternative solution that was software based to increase the range of the infrared emitter. The method was to apply peak forward current, 10 times more current than steady current, and apply a pulse width modulation with 1% duty cycle at high frequency, the result was a significant increase in emitter range but synchronization with the sensor checking became difficult because the sensor had to only check at that small interval or it will interpret it as a blockage when it really is not. With the successful solution of 2 parallel emitters to increase range, this synchronization did not have to be

implemented. The testing methods of the software portion are tabulated in table 2 of link "Kiosk supplementary document" in appendix A.

Chapter 5 - Conclusions

The initial problem that we intended to solve was the difficulty of finding attractions to visit when you are visiting a city. Often when visiting a new city, finding the best attractions to visit, the best restaurants to visit for a meal, or the best hotel to stay at can be difficult.

Our solution was to create a simple to use mobile application (that runs on iOS and Android), which solves many of the problems outlined earlier. The iOS application allows one to find attractions, restaurants, hotels (as well as many other categories of locations) very quickly and easily. The application provides filters, allowing the user to exclude locations with low ratings or high prices. We are aware that other, bigger solutions already exist for this problem (Google Maps, Apple Maps, etc.), but we believe that our solution is faster and simpler to use. For example, our solution suggests locations close to the user based on the user's preferences (explained in Chapter 4 - Technical Solution), allowing the user to quickly and easily find somewhere to visit.

In the future, our work can be extended in a number of ways. The most obvious extensions would be the completing the Android client and Touchscreen Kiosk. The Android implementation is missing most of the iOS client's functionality, the touchscreen kiosk needs to run the application to interface with the server. Other broader goals would be the inclusion of attraction occupancy. For example, having the ability to check if a hotel has any vacancies.

Appendices

Appendix A - Repository Links

Blips Server Github (https://github.com/cyruscse/blips-server)

Blips iOS Client Github (https://github.com/cyruscse/blips-iOS-client)

Blips Android Client Github (https://github.com/cyruscse/blips-android-client)

Kiosk Software Github (https://github.com/JonathanC13/Touch-Screen-System)

Kiosk Supplementary Document

(https://github.com/JonathanC13/Touch-Screen-System/blob/master/Kiosk%20Supplementary%20Document.docx)

Appendix B - Android Client Development Tools

Android Studio

The Android client is built using Java on the Android Studio development platform and, like iOS, Android development follows a general MVC architecture. The Android operating system makes use of a vast library of specialized classes, abstractions, and interfaces; appropriately taking advantage of the object-oriented paradigm, which is why Android development is mainly done in Java or C++. The application functionality is written in Java which corresponds directly to the controller component. The model and view components are composed of xml files. An activity is an xml file that represents the physical page viewed by the user. This activity file determines all view related components of the page like layout, text fields, button types, images, etc. and

these can be interpreted as the view component in the MVC design architecture. Resource files are xml files that hold various static content to be used by the application like user interface strings, layout definitions, and animation instructions. These resource files are prepared by classes modelling the data written in Java. The resource files with their accompanying model classes follow the model component in MVC design. All these components are built in Android Studios using the Gradle build system to create an Android package (APK) file that runs on a device or emulator. Refer to the official Android Studio user guide for more detailed information.

Volley Library

The Android Volley library is a framework by Google built to reduce the complexity of data communication over network. The usual method to perform an HTTP transaction and access a web API (in this case, the Blips server) is through the AsyncTask class from the Android JDK. This class requires you to override certain functions handling pre and post execution as well as a main method that runs in the background to handle all network access code. Volley on the other hand, handles all network accessing behind the scenes by leveraging the Android platform native network accessing functions. It also creates background threads for every request and manages them itself; maintaining the request for the whole life-cycle of the thread. By default, Volley implements features for a request such as automatic retry and request caching, while also supporting modular behaviour in request types between JSON objects, JSON arrays, strings, and images.

Volley mostly works by maintaining one request queue shared among all activities in an application. Every request in the queue is handled separately on its own thread. The request listens for its response and executes the assigned callback on success or handles the error appropriately. Figure B.1 below shows the make up of a request queue and its interaction between the application and a web API. For more information on Volley, visit the Android developer training page.

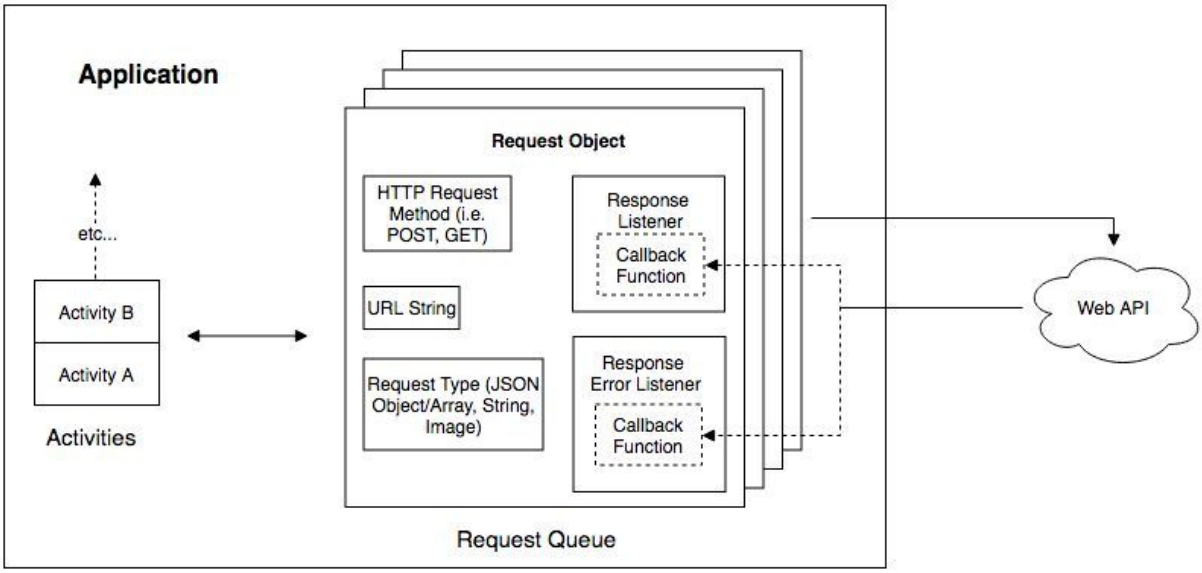

Figure B.1: Abstract Application Usage of Volley