Design and Implementation of a Location Based Data Tracking and Visualization Tool for Hardcore Traveller on iOS Platform

By

Lei Mingyu
Wang Jinghan

Department of Computer Science
School of Computing
National University of Singapore
2016/2017
B.Comp. Dissertation

Design and Implementation of a Location Based Data Tracking and Visualization Tool for Hardcore Traveller on iOS Platform

By

Lei Mingyu
Wang Jinghan

Department of Computer Science
School of Computing
National University of Singapore
2016/2017

<table>
<thead>
<tr>
<th>Project No:</th>
<th>H114470, H114500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervisor:</td>
<td>Prof. Wong Lim Soon</td>
</tr>
<tr>
<td>Deliverables:</td>
<td>Report: 1 Volume</td>
</tr>
<tr>
<td></td>
<td>Program: 1 ZIP File</td>
</tr>
</tbody>
</table>
Abstract

Nowadays, mobile devices play an indispensable role in modern people’s life. With the help of the technology, many of the daily tasks can be greatly facilitated. To be more specific, keeping a travel diary and writing journals are quite common among the hodophile. This project aims to provide them some assistance to keep the precious travel moments.

Leifr is an iOS application based on MapKit framework and backed by SpatiaLite/Realm database. It allows the user to record, view, share and play back the travelling paths. This project is completed jointly by Lei Mingyu and Wang Jinghan in fulfilment for CP4101 B.Comp. Dissertation.

Subject Descriptors:

D.1.5 Object-oriented Programming
D.2 Software Engineering
H.2.8 Database Applications
H.5.1 Multimedia Information Systems
H.5.2 User Interfaces

Keywords:

iOS, mobile application, geographic data visualization, spatial database

Implementation Software and Hardware:

macOS 10.12 Sierra, Xcode 8.3, Instruments 8.3, Swift 3.1, iOS 10, Realm Browser 2.1.7, iPhone, MacBook Pro
Acknowledgment

First of all, we want to give a special gratitude to our supervisor, Professor Wong Lim Soon. It is his considerate suggestions, information, feedback and supervision in the past eight months that has helped us progress this far. We would also like to express our thanks to our main evaluator, Professor Wang Ye, for the feedback and comments he gave during the CA presentation. These feedback were really important for us to move forward in the correct direction.

We would like to thank Professor Wong Lim Soon and Professor Wang Ye again, together with our moderators, Professor Chan Chee Yong and Professor Yap Hock Chuan, Roland. Thank you for the time and efforts you are about to put when grading our project, this report and presentations in the coming weeks.

Another acknowledgement is to Professor Roger Zimmermann. Although we have not contacted or communicated with him directly in person, yet his inspiring research in the “GeoVid” project provided us a brand new vision into the integration of geo-location data and photos.

Last but absolutely not least, we would like to express our sincerest appreciation to every professor who have taught us in the four-year undergraduate study in National University of Singapore, to every friend who helped and supported us in this project, and to everyone who is reading this report. It is your knowledge and interests that keep us always progressing, motivated and cheerful about what we do.
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Google Maps “Your Timeline” feature</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Nike+ Run Club</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Fog of World</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>View aggregated records</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>Record a new path</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>View the path lists</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>Share or delete a path</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>Select a path from the path lists</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>Select a path from the calendar picker</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>Start the animation</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>Pause and resume the animation</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>Stop the animation</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>Locate a photo</td>
<td>15</td>
</tr>
<tr>
<td>14</td>
<td>Check statistics of visited Countries</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>Component diagram for front-end architecture</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>Component diagram for back-end architecture</td>
<td>19</td>
</tr>
<tr>
<td>17</td>
<td>Customized tab bar at different snapping levels</td>
<td>23</td>
</tr>
<tr>
<td>18</td>
<td>Different map type</td>
<td>24</td>
</tr>
<tr>
<td>19</td>
<td>MapKit with annotation views</td>
<td>27</td>
</tr>
<tr>
<td>20</td>
<td>MapKit with overlay and overlay renderer</td>
<td>27</td>
</tr>
<tr>
<td>21</td>
<td>Prototype screenshots with MapKit framework</td>
<td>28</td>
</tr>
<tr>
<td>22</td>
<td>Mapbox Studio</td>
<td>29</td>
</tr>
<tr>
<td>23</td>
<td>Mapbox with annotation views</td>
<td>30</td>
</tr>
<tr>
<td>24</td>
<td>CPU usage for rendering 13,000 markers in Mapbox with annotation images</td>
<td>31</td>
</tr>
<tr>
<td>25</td>
<td>CPU usage for rendering 13,000 markers in Mapbox with annotation views</td>
<td>31</td>
</tr>
<tr>
<td>26</td>
<td>Mapbox with overlay</td>
<td>32</td>
</tr>
<tr>
<td>27</td>
<td>Mapbox with MGLGeoJSONSource</td>
<td>32</td>
</tr>
<tr>
<td>28</td>
<td>Rendering geo-location data with Mapbox framework</td>
<td>34</td>
</tr>
</tbody>
</table>
List of Tables

1. Feature correspondence for front-end architecture components
2. Map coordinate system conversion routines
3. Map frameworks comparison
4. RAM capacity of different iPhones
5. SpatiaLite SQL functions used in the project
Contents

Abstract i

Acknowledgement ii

List of Figures iii

List of Tables iv

1 Introduction 1

1.1 Background ................................................. 1
1.2 Overview .................................................. 2
1.3 Comparison to Existing Products ......................... 2
  1.3.1 Google Maps ......................................... 3
  1.3.2 Nike+ Run Club ...................................... 3
  1.3.3 Fog of World ........................................ 4

2 Use Cases and Demonstration 6

2.1 View Aggregated Records ............................... 6
2.2 Record a New Path ...................................... 7
2.3 View All Recorded Paths ................................. 8
  2.3.1 View the path lists .................................. 8
  2.3.2 Share or delete a path ................................. 9
2.4 Replay Paths .............................................. 10
  2.4.1 Select Paths .......................................... 10
  2.4.2 Playback Control ................................... 12
2.5 Locate a Photo ............................................ 15
2.6 Check Statistics of Visited Countries ................. 16

3 Application Architecture 17

3.1 Front-end Architecture ................................. 17
3.2 Back-end Architecture ................................ 19
4 Development Details

4.1 Front-End Details
4.1.1 Fundamentals
4.1.2 Map Framework
4.1.3 Geo-Location Information Recording
4.1.4 History Overview
4.1.5 Path Lists
4.1.6 Path Playback
4.1.7 Photos
4.1.8 Statistics
4.1.9 Settings

4.2 Back-End Details
4.2.1 Fundamentals
4.2.2 SpatiaLite
4.2.3 Realm
4.2.4 Photos
4.2.5 Direct Files

5 Conclusion

5.1 Summary
5.2 Difficulties and Limitations
5.3 Future Plans

Project Contributions

References

Appendix A - Program Listing

Appendix B - Installation Guide
1 Introduction

1.1 Background

“Well begun is half done”, the ancient Greek philosopher Aristotle once wrote this in his work *Politics*. Seventeen centuries passed, this famous proverb still applies in every single bits of people’s life. As for travellers, the *well begun* often comes from the thorough planning of their journey.

One of the most effective way to plan the journey is by reading others’ journals. Nowadays, travellers can easily find a huge amount of such information on travelling websites. Usually there are three categories of journals:

- the ones with lots of photos and sight recommendations
- the ones with lots of personal thoughts and understanding of different cultures
- the ones with detailed schedules and routes

We travelled a lot during the exchange period in Europe in 2016, our personal pain point is that when planning the journey, we can find tons of information of the first category listed above, but what we need most is actually the third one. After the travellers have some places in mind, the problem they need to solve is to organize them properly so that they have sufficient time spent in each of the places without wasting too much time travelling back and forth in between.

There are various reasons why we can rarely find information of the third category. One possible cause might be the difficulties of keeping track of everything regarding the schedules and routes: as travellers, they may clearly remember the different sights they have visited, but seldom can they tell what time they left the hotel on the second day and how long did they spend on the bus from the hotel to the restaurant that they had lunch.

To facilitate travellers to keep the information of their detailed travelling schedule, we come up with the idea of building an application using mobile phones with Global Po-
sitioning System (GPS) hardware to record and visualize the users’ geo-location change within a period of time.

Section 1.1 is written by Mingyu.

1.2 Overview

Named after the Icelandic explorer Leifr Eiríksson, Leifr is an iOS application to address the problem mentioned above. The core functions of this application includes:

- record a path of user’s geo-location information (longitude, latitude and altitude) with the exact time that this information is recorded
- visualize the visited places on a map and color them in a “heat map” manner to indicate how frequent the user visits a specify location
- view/share/replay a recorded path

There are also some advanced features implemented:

- replay all the paths sequentially starting from a selected date
- estimate the location where a chosen photo is taken
- display the statistics about all the countries the user visited

Some of the extra features are planned to be implemented in the future. These will be discussed in the Section 5.3 Future Plans.

Section 1.2 is written by Mingyu.

1.3 Comparison to Existing Products

During the project ideation process, we studied some existing mobile applications that partially fulfil our requirements. In general, they can be categorized into three different types, and a typical example will be chosen from each of them and compared with our proposed applications.

The three categories are:
- Map applications with the capability to keep track of the location history (e.g. Google Maps by Google, Inc.)
- Activity tracking app with the capability to share a path with others (e.g. Nike+ Run Club by Nike, Inc.)
- Visited locations visualization app (e.g. Fog of World by Ollix)

1.3.1 Google Maps

Google Maps is probably one of the most popular and successful map application in the world, it provides a completed set of functionalities coming along with the map services like navigation, address finding and location sharing. In 2015, Google released a new feature in Google Maps called “Your Timeline” (Sanz (2015)). It provides a way to track the user’s location history as long as the user turn “places you go” on. The visualization result of the location history is shown in Figure 1.

Google Maps effectively identifies the places the user visited recently, and predicts the mode of transportation based on the time spent in between. The result is like a normal navigation routes, all the critical places are identified and highlighted, then the paths in between are simply connecting those critical places. One strength of this timeline feature is the powerful map data support from Google; for example, in Figure 1, some of the photos are displayed for one of the place that the user visited, this definitely makes the history more informative. Nevertheless, this feature still lacks of some crucial functionalities we are looking for, like path sharing and full control of the start and end of recording a path. It’s more like a auxiliary feature that provides some useful information feedback while the user is using the Google Maps app.

1.3.2 Nike+ Run Club

Nike+ Run Club is an application developed by Nike to assist the runners to manage their running plan, history and progress. The special feature we are particularly interested in is the GPS tracking and sharing.
An example of the tracking result is shown in Figure 2. Despite the fact that the app is using color hue as the visual variable to visualize the speed in different parts of the overall track (which is quantitative), it is a good visualization result as to mark the running path on the map. This result can be conveniently shared together with some statistics about the running experience like the duration and average pace. This is also what we want to achieve in the proposed application. Although Nike+ Run Club meets the minimal core features we want to have for the traveller app, it also introduces lots of unnecessary details regarding the running statistics (after all it is a running app). Furthermore, it will be very unlikely to see some more advanced features for traveller like photo location finder to be integrated with this kind of app.

1.3.3 Fog of World

Fog of World is gamified path history tracking tool that allows the user to clear the “fog” on the map while exploring the world. It clearly shows the path that the user visited and introduces lots of social element like leveling system, Apple Game Center integration
and record sharing. Amongst the several existing apps mentioned in this section, Fog of World is the one closest to our proposed idea: designed for the travellers, support path history overview, cloud backup and import of past history. However, there is a key component from our proposal missing in this app: time. It will be much more interesting to add the time information on top of the geo-location data collected; for example, users can replay the path that they recorded a long time ago. This makes more sense when concerning about the travellers: date, time and even the season can be an important factor that they want to take into account when writing the journal and sharing with other travellers.

Section 1.3 is written by Mingyu.
2 Use Cases and Demonstration

2.1 View Aggregated Records

Aggregated records refer to all the places the user have been to. To visualize them on the map:

1. Select the first tab
2. Scroll, zoom in or out to find the location of your interests

As shown in Figure 4, the detail level will change as the user zooms in and out.

Figure 4: View aggregated records

Section 2.1 is written by Jinghan.
2.2 Record a New Path

To record a new path, the user will need to:

1. Select the first tab
2. Drag the tab bar out and snap it at the first level
3. Tap on the round button in red

As shown in Figure 4, the round button will transform to a square. The square record button indicates the application is now in the recording state. The user location button on the left is in dark blue. This means the application is in user tracking mode, in which the region that the map view shows will be updated so that the user location marker is always at the center of the screen.

Section 2.2 is written by Jinghan.
2.3 View All Recorded Paths

2.3.1 View the path lists

To view the lists of all available paths, the user will need to:

1. Select the first tab
2. Drag the tab bar out and snap it at the first level
3. Tap on path icon on the right hand side of the control panel
4. Select either the “My Track” tab or the “Inbox” tab
5. Tap the “Load More” button if more paths are desired

As shown in Figure 6, a new view will be presented after tapping on the path icon. In the presented view, the user’s own paths will be shown with a green background while the paths he or she received from his or her friends will be shown in a red background.
2.3.2 Share or delete a path

To share a path, the user will need to:

1. Go to the path list view as in section 2.3.1
2. Drag the cell which contains the target path to the left
3. Tap the blue “Share” button
4. Select the application to send the path

After the application is selected, an file with an extension of .leifr will be delegated to that particular application.

To delete a path, the user will need to:

1. Repeat the first two steps as in “To share a path”
2. Tap the red “Delete” button. Please note that this action cannot be recovered

Section 2.3 is written by Jinghan.
2.4 Replay Paths

Playing back a path animation can be divided into two sub-tasks: path selection and path playback control, where the former sub-task is the pre-condition of the latter one.

2.4.1 Select Paths

In the application, two different ways are provided to complete the path(s) selection: one is from the path lists in the history overview page and the other is from the calendar in path playback page.

2.4.1.1 Select a Path From Path Lists  To select a path from the path lists, the user will need to:

1. Go to the path list view as in Section 2.3.1.

2. Switch to the correct page depending on whether the user wants to select a path from the recorded paths list or the received paths list.
3. Tap on the path to be played back.

After the path is selected, the user will be directed to the path playback view, where he or she can control the path playing as describe in Section 2.4.2.

![Figure 9: Select a path from the calendar picker](image)

**2.4.1.2 Select Paths Starting From Some Date** To select a list of paths starting from some date, the user will need to:

1. Select the second tab
2. Drag the tab bar out and snap it at the first level
3. Tap on the calendar button
4. Scroll to the correct month of the starting date
5. Select and highlight the starting date (the date with white background in Figure 9)
6. Tap on the “Show” button at the top-right corner
After starting date is selected, the user will be directed back to the path playback view, where he or she can control the path playing as describe in Section 2.4.2.

2.4.2 Playback Control

After the user successfully selects the path(s) as describe in Section 2.4.1, the user can start to control the playback animation. In general, there are four different operations the user can perform, namely start, pause, resume and stop.

2.4.2.1 Start the Animation To start the playback animation, the user will need to:

1. Select the path(s) as describe in Section 2.4.1.
2. Tap on the play button

After the animation starts to play, the hover tab will be collapsed and a new header view showing the time information for the current annotation point will appear (see Figure 10: Start the animation).
10, middle). If the user select multiple paths from the calendar, after the animation of one path finished, the system will pause for 0.5 second before continuing to play the next path (see Figure 10, right). If all the paths have been played, the animation will be stopped.

![Figure 11: Pause and resume the animation](image)

**2.4.2.2 Pause and Resume the Animation** To pause the playback animation, the user will need to:

1. Drag the tab bar out and snap it at the first level, since the tab bar will be collapse once the user starts the animation. The result of this step is shown in Figure 11, left.

2. Tap on the pause button.

To resume the playback animation, the user will need to:

1. Tap on the resume button, which is the same as the play button.

The user has the freedom to pause and resume the path animation by any number of
times before the animation ends. The user can also drag the map or zoom in/out while the animation is being played.

![Animation Interface](image)

**Figure 12: Stop the animation**

### 2.4.2.3 Stop the Animation

To stop the playback animation, the user will need to:

1. Drag the tab bar out and snap it at the first level, since the tab bar will be collapse once the user starts the animation. The result of this step is shown in Figure 12, left.

2. Tap on the stop button.

After the user stops the animation, the path together with the annotation marker will be removed from the map, the header view showing the time information will also disappear. The user can tap on the start button again to start the animation from the beginning just like in Section 2.4.2.1.

Section 2.4 is written by Mingyu.
2.5 Locate a Photo

To check the location/estimation of a photo, the user will need to:

1. Select the third tab
2. Browse through the photo list
3. Tap on the photo he or she is interested in
4. Drag the photo up in the new view presented
5. Zoom in or out as he or she likes

As shown in Figure 13, if the location of the selected photo is an estimation according to the geo-location data stored in the database, the error will be shown in a cyan circle.

Section 2.5 is written by Jinghan.
2.6 Check Statistics of Visited Countries

To check visited countries and the paths in a specific country, the user will need to:

1. Select the fourth tab
2. Select world or any of the continents
3. Browse the country list
4. Tap on the flag of the country he or she is interested in
5. Zoom in or out in the presented map view as he or she likes

As shown in Figure 14, the blacked out background in the map preview view is tappable. Tapping on it will dismiss the map preview.

Section 2.5 is written by Jinghan.
3 Application Architecture

3.1 Front-end Architecture

The front-end architecture is illustrated by Figure 15.

Since this application is following the standard Model-View-Controller architecture pattern, the main interaction in the front-end is between the view controllers and the views. The components in grayish blue corresponds to the basic hover tab structure and relevant protocols, which will elaborated in Section 4.1.1 Fundamentals. Each of the view controllers in this application is inherited from the base class LFViewController, and the view controllers for the four main pages should override the delegate method

![Component diagram for front-end architecture](image-url)
from LFHoverTabBarDataSource in order to provide the specific control view for each pages.

The view controller classes and view classes for different feature pages are labelled with different background color, the correspondence can be find with Table 1.

<table>
<thead>
<tr>
<th>Feature group</th>
<th>Section</th>
<th>Color in Figure 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fundamentals</td>
<td>4.1.1</td>
<td>• Grayish blue</td>
</tr>
<tr>
<td>Path history overview</td>
<td>4.1.4</td>
<td>• Pink</td>
</tr>
<tr>
<td>Path lists and path sharing</td>
<td>4.1.5</td>
<td>• Orange</td>
</tr>
<tr>
<td>Path playback</td>
<td>4.1.6</td>
<td>• Yellow</td>
</tr>
<tr>
<td>Photos</td>
<td>4.1.7</td>
<td>• Green</td>
</tr>
<tr>
<td>Statistics</td>
<td>4.1.8</td>
<td>• Cyan</td>
</tr>
<tr>
<td>Settings</td>
<td>4.1.9</td>
<td>• Blue</td>
</tr>
</tbody>
</table>

Table 1: Feature correspondence for front-end architecture components
3.2 Back-end Architecture

In the back-end of this application, two databases are working together to support the persistent data storage and caching. Therefore, two different sets of models are used when dealing with data from different databases.

In general, LFPoint and LFPath are the two fundamental models for the geo-location data used; wrapping LFPath, LFIncomingPath is the special path model used when handling the path shared by others. These are all subclasses of the basic NSObject class from iOS Foundation framework. Another set of models are the Realm Object used for caching purpose. There are three types of information we cached in the Realm database: zoom level (LFCachedLevel), coordinates (LFCachedPoint) and country information (LFCachedCountry).
Several managers (see Service component in Figure 16) are responsible for the communication between front-end components and the back-end database. The elaboration of each service can be found in relevant subsections of Section 4 Development Details.

Class diagrams in section 3 are drawn by Jinghan. Texts in section 3 are written by Mingyu.
4 Development Details

As the project title suggested, design and development are the two critical components of this project. In the previous section Application Architecture, the high level structure of this application has been discussed. In this section, we will go into the details to discuss about the specific design decisions and the final implementation approach for each individual feature. The whole section is categorized into front-end part and back-end part, and the features of each part are listed accordingly.

This section is written jointly by Jinghan and Mingyu. Please refer to the individual subsections for specific attributions.

4.1 Front-End Details

There are seven general features in terms of front-end development. The first subsection Fundamentals will discuss the overall view hierarchy, the second subsection Map Framework will compare the different map frameworks available and our rational behind choosing Apple MapKit as the map framework for this application, the third subsection Geo-Location Information Recording will discuss the details of geo-location data collection, while the rest will keep the focus on different views.

4.1.1 Fundamentals

Based on the well accepted Model-View-Controller pattern, Apple has provided several standard templates for developers to organize the views in their applications. In general, developers are recommended to use these template to take advantages of the built-in optimizations. Amongst the various templates provided, we picked out the ones we may potentially make use of:

- master-detail application template
- page-based application template
- tabbed application template
The master-detail application template provides a convenient starting points for data driven applications where only brief data are shown in a list. Users will only view the details when they click on a specific list item. UITableViewDataSource as the list(master) view, is the perfect fit when showing texts or simple images, but when the geo-location data come in, the list introduces too much cognitive load to the viewer.

In the page-based and tabbed application templates, there are no cognitive hassle as in master-detail template when displaying geo-location data. However, in neither of these two, there allows native accommodations for the controls. A naive solution may be in each essential view (i.e. each page in a page-based template or each tab in a tabbed template), we put the relevant controls directly on top. This will certain solve the problem, but at the cost of messing up the user’s working space. Furthermore, there are possibly cases where one single control needed to be shared among different pages or tabs, but with this naive solution we will have to duplicate the codes here and there.

Due to the fact that none of the templates suits our needs, we decided to implement our own view hierarchy. The customized scheme we proposed and implemented can be considered as an extension of the tabbed application template. The main difference is that in our hierarchy, the tab bar can be pulled up and reveal the control panel, where various controls are placed. The controller of each essential view will inherit a common parent view controller, which implements a protocol to delegate the data sources of the controls. When a control needs to be shared among different views, it is defined in the parent view controller. When a control is only needed in one single particular view, it is defined in the specific child view controller. In this way, we avoid the potential duplicated code. Furthermore, the customized tab bar is able to be snapped into several code-defined levels so that a control may only be shown when it is really needed by the user. Figure 17 shows the customized tab bar in different snapping levels.

Section 4.1.1 is written by Jinghan.
4.1.2 Map Framework

For any application dealing with geo-location data, map is always the inherent medium to present the location information. In our initial user interface prototypes, the map is also the most important element in the entire front-end view structure. For the two core functionality pages, history overview and path playback are simply one map view with different kinds of data visualization (see Section 4.1.4 History Overview and Section 4.1.6 Path Playback). For the other secondary feature pages like photos and statistics, map also plays an important role as an essential component (see Section 4.1.7 Photos and Section 4.1.8 Statistics).

In order to display and integrate a map in the application, an appropriate map framework is needed. A map framework provides the necessary geographic information on the map and supports the basic interaction between users and the map like panning, tapping, zooming in and out etc. Usually, the map displayed from a framework can be configured with different map types. Some of the common map types includes:

- Road map: this is the default map type for most of the map frameworks (see Figure
• Satellite map: the map is displayed as the imagery collected by the satellites operated by the governmental or non-governmental organizations (see Figure 18, middle)

• Hybrid map: this is the combination of the road map and satellite map. The map is displayed with the satellite imagery background with the labels of roads and buildings on top (see Figure 18, right)

Satellite map gives a sense of reality since the map is composed by the real photos. When the users is interacting with the satellite map, they will have an experience as if they were looking at the earth from some aerial platform. However, when visualizing the geolocation data, there will certainly be some markers displayed on the map of the satellite imagery. The whole user interface would look pretty messy, and the path rendered will lose the visual salience thus it defeats the purpose of showing visited track on the map view. Compared to the satellite map and hybrid map, the default road map is much simpler, most of the background is painted in light plain color and there are not many map elements interfering the path viewing. For the sake of clearness, we chose the road map as the map type for this application.
We have done some researches and prototypes with mainly two different map frameworks, namely Apple MapKit framework and the Mapbox framework. The rest of this section will briefly discuss these two different framework.

4.1.2.1 **Apple MapKit**  
MapKit is a native iOS SDK for displaying map in the application in the same way as the built-in map application. It has the best compatibility and integration with the other Apple framework like Core Location and Core Animation.

MapKit uses Mercator cylindrical map projection to display the map view. During the projection, the world coordinates are mapped onto the surface of a cylinder, and then unwrapped to generate a flat map. A simple conversion from longitude $\lambda$ and latitude $\phi$ to the map coordinate $x$ and $y$ can be calculated as:

\[
x = \lambda - \lambda_0 \\
y = \ln(\tan(\frac{\pi}{4} + \frac{\phi}{2}))
\]

\[
= \frac{1}{2} \ln\left(\frac{1 + \sin(\phi)}{1 - \sin(\phi)}\right)
\]

\[
= \sinh^{-1}(\tan(\phi))
\]

\[
= \tanh^{-1}(\sin(\phi))
\]

\[
= \ln(\tan(\phi) + \sec(\phi))
\]

According to Apple (2016b), using Mercator projection will benefit the general navigation since a straight line drawn between any two points on the map yields a course heading that can be used in actual navigation on the surface of the Earth.

After the Mercator projection, the world coordinates can be converted to map points; another conversion depending on the current zoom level from map points to the actual window points need to be done. MapKit provides a set of APIs to calculate these conversions, and they are listed in Table 2.

The main rendering process of MapKit is tile-based, which means when the user is zooming in or out, the map is rendered as different tiles in multiple threads. The number of tiles and the size of them are based on the current zoom level. Tile-based map rendering
From | To | Conversion routines
---|---|---
Coordinates | Points | `convertCoordinate:toPointToView:(MKMapView)`
Coordinates | Map points | `MKMapPointForCoordinate`
Map points | Coordinates | `MKCoordinateForMapPoint`
Map points | Points | `pointForMapPoint:(MKOverlayRenderer)`
Points | Coordinates | `convertPoint:toCoordinateFromView:(MKMapView)`
Points | Map Points | `mapPointForPoint:(MKOverlayRenderer)`

Table 2: Map coordinate system conversion routines

is efficient since the rendering processes for different tiles are taken place in different 
threads concurrently, but it would also makes the overall rendering process looks jaggy 
since the user might find the map is rendered tile by tile. This is even more obvious when 
the user is scrolling the map or zooming in and out rapidly.

While developing with MapKit, the developers can create their own renderer to display 
the map as desired. However, the developers have less freedom in terms of map style 
customization. Most likely they can only display the map in the same way as the native 
Apple maps; there is not even a way to hide the road label layer from the map view.

Two common architectures can be implemented with MapKit framework:

1. placing `MKAnnotationView` on `MKMapView` (Figure 19)
2. using `MKOverlay` and `MKOverlayRenderer` to manage the annotation overlay (Figure 20)

In the first case, for each of the geo-location coordinates fetched from database, we 
create an instance of `MKAnnotationView` with the longitude and latitude unwrapped 
from the `CLLocationCoordinate2D` before rendering them on the map, and then add the 
`MKAnnotationView` on to the `MKMapView` by calling the `addAnnotation(annotation:MKAnnotation)` 
method. This is a simple yet naive way to display the data points. As the number of 
points increases, the performance might suffer since many annotations are directly put 
on the map view hierarchy. A demonstration of one prototype with this architecture is 
shown in the left screenshot in Figure 21.
In the second case, after fetching the geo-location coordinates from database, instead of adding `MKAnnotationView` directly to the `MKMapView`, we can create a subclass of `MKOverlay` and `MKOverlayRenderer`, and then use the customized subclass we defined of `MKOverlayRenderer` to render the visualization result from the data. With this hierarchy, the performance problem with many `MKAnnotationView` stacking in map view hierarchy is addressed, since all the coordinates data are compacted to one `MKOverlay`. Moreover, with the customization of `MKOverlayRenderer`, the path can be drawn with more variation. For instance, in the demonstration of another prototype with customized overlay renderer (the right screenshot in Figure 21), the path can be rendered like the cleared fog in Fog of World (see Section 1.3.3 Fog of World).
4.1.2.2 Mapbox  Mapbox is another map framework developed by the map service provider with the same company name “Mapbox”. The framework is open-sourced with a productive development team. The most distinct feature of the map provided by this framework is the freedom to customize the map style. (Mapbox (2017))

Unlike the Apple MapKit framework which only allows the developer to adjust very limited customization options, Mapbox even provides a powerful map customization platform called Mapbox Studio. With the help of this platform, developers are able to make their map unique from others. Mapbox makes use of vector assets for tile rendering, therefore when the users are zooming the map in and out, they can expect the map content to be updated seamlessly (while in Apple MapKit the tiles need to be updated since they are not vector tiles).

Although Mapbox provides thorough customization freedom for map styling, it limits developers’ capability to customize the rendering process. That is to say, the developers will not be able to create their own map renderers as in MapKit. Instead, the rendering work will be delegated to the Mapbox framework to complete.
Three common architectures can be implemented with Mapbox framework:

1. placing `MGLAnnotation` on `MGLMapView` (Figure 23)
2. using `MGLOverlay` to manage the annotations (Figure 26)
3. using the “datasource-layer” model to manage the annotations with geo-location data (Figure 27)

The first approach is similar to the naive way of putting annotations on MapKit’s `MKMapView`, except the `MKAnnotationView` and `MKMapView` are replaced by `MGLAnnotation` and `MGLMapView`, and the method for adding `MGLAnnotation` is `addAnnotation(annotation:` `MGLAnnotation`). During prototyping, `MGLPointAnnotation`, which is a concrete class conforms to `MGLAnnotation` protocol, is used for rendering the geo-location points. The appearance of the `MGLAnnotation` can be configured by overriding the following two delegate methods from `MGLMapViewDelegate`:

1. `mapView(mapView:imageForAnnotation:)`
2. `mapView(mapView:viewForAnnotation:)`

These two methods manage the customized appearance of the `MGLAnnotation`. During the implementation of the overridden methods from `MGLMapViewDelegate`, if one of these
two methods is not implemented or the return value is set to `nil`, Mapbox renderer will use the other as the customization method. If neither of these two methods are implemented or they all return `nil`, Mapbox will use the default annotation as the annotation appearance.

There are a few differences between them which we need to consider this when choosing the appropriate configuration approach. Configuring annotation images will make the annotations shown as static images. This allows the developer to design the marker with other tools and save it as a separate image file. On the other hand, the second approach is returning a view object for the annotation to be rendered. In this case, the view will be drawn programmatically in the subclass of `MGLAnnotationView`. Rendering annotation markers as annotation view will make them more compatible with Apple’s native UIKit, Core Animation and some other Cocoa Touch frameworks, which enhances the interaction between the user and the system. However, the trade-off is higher CPU usage and longer drawing process than using static image. We performed a benchmarking experiment using Xcode Instruments tool, which adds 13,000 markers on the map, and then do some basic operation like panning and zooming. The result can be found in Figure 24 and 25. The conclusion of this experiment is pretty straightforward: it proves that using static images as annotation markers yields better performance.

Continuing the discussion of Mapbox integration approach, Figure 26 shows an improved
Figure 24: CPU usage for rendering 13,000 markers in Mapbox with annotation images

Figure 25: CPU usage for rendering 13,000 markers in Mapbox with annotation views
version which is similar to Figure 20. The ideas behind are the same: add one overlay on top of the map view and render the annotations in the overlay layer instead of adding too many annotation on the map view directly. Although the previous paragraph mentioned that using static image can alleviate some memory pressure, mobile devices are still not able to afford the great number of annotations displayed at the same time. Since our application is dealing with many geo-location points in one recorded path, it is better to have the overlay to manage different points instead.

In the previous approaches, the general flow of fetching and displaying the points are similar, which is querying database to get the data, converting the data to a list of CLLocationCoordinate2D, creating annotations with the coordinates information, and
finally displaying it on the map. Mapbox provides another way to achieve the same goal: using `MGLStyleLayer` to manage rendering and `MGLSource` to manage data source. More specifically, in order to render the visited path, we can encode the geographic information from database into GeoJSON format, create `MGLGeoJSONSource` with the encoded data, associate it with an `MGLCircleStyleLayer`, and finally add these two components to the map view together by calling `addLayer(layer:)` and `addSource(source:)` method from the `MGLMapView`'s `style` attribute. One advantage of this approach is that it simplifies the integration process between map framework and the backend data. With `MGLGeoJSONSource`, the GeoJSON data can be directly generated from the SpatiaLite query (see Section 4.2.2, SpatiaLite) with some format tuning from `LFGeoJSONManager`. It then can be bound to the map view and render the visualization outcome. Unfortunately, the Mapbox support for rendering with `MGLGeoJSONSource` was the new feature by the time we were doing the technology study and research. Until we finished the prototype and decision phase, it’s still under development and the feature was only partially available in the alpha version of 3.4.0, and some key features we required in the application, such as dynamic update of GeoJSON source when a new point is recorded, is not supported.

The screenshots of path rendering using Mapbox framework are shown in Figure 28. The one on the left is using default annotation marker, while the one on the right is using `MGLGeoJSONSource` with `MGLCircleStyleLayer`.

### 4.1.2.3 Map Framework Decision

To conclude the discussion in the previous paragraphs, here is a table of all the advantages and disadvantages of MapKit and Mapbox frameworks.

To fully support our proposed features for this application, there are several criteria we need to consider when choosing the appropriate framework:

- The map framework should be stable to use.
- The map framework should provide the rendering support for visualizing the geo-location data.
Advantages

<table>
<thead>
<tr>
<th>MapKit</th>
<th>Mapbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well documented</td>
<td>Well documented</td>
</tr>
<tr>
<td>Native stable SDK</td>
<td>Active community</td>
</tr>
<tr>
<td>Best compatibility</td>
<td>GeoJSON source support</td>
</tr>
<tr>
<td>No extra library needed</td>
<td>Open-sourced, fast-faced</td>
</tr>
<tr>
<td>Customizable rendering process</td>
<td>Highly customizable style</td>
</tr>
<tr>
<td>Efficient tile-based rendering</td>
<td>Vector tile, smooth loading</td>
</tr>
</tbody>
</table>

Disadvantages

<table>
<thead>
<tr>
<th>MapKit</th>
<th>Mapbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jagged tile loading</td>
<td>No control over renderer</td>
</tr>
<tr>
<td>Less customizable style</td>
<td>Unofficial, compatibility problems</td>
</tr>
<tr>
<td></td>
<td>Less flexible to manage dynamic data</td>
</tr>
</tbody>
</table>

Table 3: Map frameworks comparison

Figure 28: Rendering geo-location data with Mapbox framework
• The map framework should allow the implementation of annotation animation for the path playback feature.

• The map displayed should be simple, clear, and clean without too much interference with the rendered path.

Judging from the advantages and disadvantages of these two frameworks and the how well the framework suits our features, we have decided to use Apple MapKit as the map framework in this application. However, since Mapbox is under such a productive and aggressive development, we will keep following the updates from this framework. We might consider supporting both frameworks in the future if the compatibility problems between Mapbox and our desired features are solved in the future.

Section 4.1.2 is written by Mingyu.

4.1.3 Geo-Location Information Recording

The core features of this application are based on the data recorded by user. In the application, a singleton manager LFGeoRecordManager is dedicated to all the services involving geo-location recording. When the user switches on the recording, the flag `isRecording` in LFGeoRecordManager will be set to `true`. This flag is checked in several places like the LFHistoryViewController (see Section 4.1.4) and AppDelegate. In the latter case, the flag checking happens when `CLLocationManager` detects an update of the current location with the delegate method `locationManager(manager: didUpdateLocations locations:)`: if the flag is on, LFGeoRecordManager encapsulates the coordinate of the new location and the time information into an abstracted model LFPoint and then appends it to a buffer path. The rational behind is that it takes significantly longer time to access the database than manipulating a in-memory variable. By buffering the incoming geo-location information and flushing the path into database regularly, frequent database accesses are avoided. The interval between each flush is set to 600 seconds. The user can also perform manual flushing in the settings page (see 4.1.9 Settings). Once the recording is stopped by the user, the current buffer path will be flushed as well.

In LFGeoRecordManager, a mutex is applied to ensure that appending a point and flushing
the existing points won’t happen simultaneously.

Section 4.1.3 is written by Mingyu.

4.1.4 History Overview

History Overview is the first view the user will see after they open the application. It shows aggregated records of all places the user have been to in a pixel-style overlay. While the user zooming in and out, different levels of details will be loaded dynamically according the zoom level of the map. This is shown in Figure 29.

As mentioned in section 4.1.2.3, we will take the combination of MapKit, MKOverlay and MKOverlayRenderer to render our geo-location data. When the map renders a tile for a specific region, the rendering process include six steps:

1. Get all the overlays of the MKMapView.

2. Test the target region against the boundingMapRect property of the overlay, and only keep the overlays that lie partially or completely in the target region.
3. The `MKMapView` call its delegate to get the renderers for the shortlisted overlays.

4. The method `canDraw(mapRect:zoomScale:in:)` of the renderer is called. This method returns a boolean indicates whether the renderer is ready for the drawing.

5. If the renderer is ready, the method `draw(mapRect:zoomScale:in:)` of the renderer is called and the renderer draws the customized contents onto a provided `CGContext` instance.

6. The `CGContext` instance is used to render the drawing on screen.

The key step in this process is step 5, where our customized renderer draws contents onto the `CGContext` according to the `mapRect` and `zoomScale` passed in. The `mapRect` property represents the rectangular region of the tile on the 2D map (i.e. the outcome of Mercator projection as mentioned in section 4.1.2). `zoomScale` represents the degree of details the map is in. It is a `CGFloat`, ranges from $2^{-17}$ (least detailed level) to $2^{1}$ (most detailed level).

A more popular and intuitive descriptor for the degree of details is zoom level, which is a non-zero integer. Zoom level 0 gives the world in a single tile and every time the level is incremented by 1, each of the tiles is split into 4 ($2 \times 2$). The correlation between zoom level ($l$) and zoom scale ($z$) is:

$$ l = 20 + \log(z) $$

Across the project, zoom level and zoom scale are used interchangeably according to the situation.

After getting the parameters from the caller, the `draw(mapRect:zoomScale:in:)` method will draw the overlay in the following steps:

1. Query `LFCachedDatabaseManager` with the specific `mapRect` and `zoomScale`, and get a list of `LFCachedPoints`.
2. For each `LFCachedPoint`, construct a `MKMapPoint` instance with the `x` and `y` value stored inside.
3. Convert the `MKMapPoint` to a `CGPoint` using the method `point(mapPoint:)`. 
4. Calculate the drawing size for each point according to current `zoomScale`.

5. Calculate the drawing rect with the size from step 4 and the point from step 3.

6. Calculate color of the point according to its `count` property. `count` of a point represents the number of times the point has been visited. A larger `count` will lead to a higher alpha value.

7. Fill the rect from step 5 with the color from step 6

One thing that is worth noting is that for each point, the alpha value of the color is not necessarily in a linear function with the `count` of that point. This is based on the consideration that people tend to stay at one place (his or her home, company or school) most of the time. If the alpha calculation follows a linear function strictly, the outcome will be that several points are completely opaque, while the others are almost completely transparent and can hardly be seen. To address this issue, we set a lower (0.2) and an upper bound (0.8) for the alpha value as well as a “count cap” for the alpha change. Then `count` values from 0 to the count cap maps to the possible alpha range. In this way, a relatively aesthetic appearance of the overlay is always guaranteed.

Section 4.1.4 is written by Jinghan.

### 4.1.5 Path Lists

In path list views, users may view, delete and share the paths of their own and the paths from their friends, as shown in Figure 30. There are two view controllers presenting similar views but fetching data from different back-end components. `LFMyTrackViewController` shows the user’s own paths, and it fetches paths from the main database through the facade of `LFDatabaseManager`. On the other hand, `LFInboxViewController` shows the paths from friends, and fetches the data directly from local file systems through the facade of `LFLocalFileManager`. The differences between these two back-end components and the rational behind will be discussed in section 4.2.5.

In `LFMyTrackViewController`, paths are loaded on batches instead of being loaded all at once to reduce the latency when the view is presented. There is no such mechanism in
LFInboxViewController because the incoming paths from friends are considered sparse and manageable.

In either view, sliding a cell to the left will reveal two buttons, as shown in the middle picture of Figure 30. This behavior is achieved through implementing the protocol method `tableView(tableView:editActionsForRowAt:)` in `UITableViewDelegate`. In this method, we can define the buttons and specify the corresponding actions when a user taps on it.

When the sharing button is tapped, the system file sharing prompt will be popped out, and the user will then be able to select the application to which he or she wants Leifr to export. In Figure 31, the left and middle picture show the export process. In another application (probably on another device), as the right picture of Figure 31 shows, Leifr will appear in the sharing prompt as the first option to handle the `.leifr` files.

The path exporting described above is achieved in three steps:

1. Archive a path into a file. This is done through the implementation of `NSCoding` protocol, the details of which will be discussed in section 4.2.5.
2. Initialize an instance of UIDocumentInteractionController with the file’s URL.

3. Present the UIDocumentInteractionController to pop the system sharing prompt.

After implementing export, we still need to associate our app with the file extension in the system so that it can be recognized as a handler of that particular type of files (.leifr files). This is done by adding in the CFBundleDocumentTypes entry in the application specification file info.plist.

Section 4.1.5 is written by Jinghan.

### 4.1.6 Path Playback

In the path playback page, there are two possible states: the state when there are some paths selected to play and the state when there isn’t any path selected. Before the user selects any path, the control panel for this page will only contain a calendar icon (see Figure 32, left). If some paths have been selected, but the animation has not started, there will be an extra play button in the control panel (see Figure 32, middle). If the
user initiates the animation, the play button will be replaced by a pause/resume button. Additionally, a stop button will also be added into the control panel to support the full playback control (see Figure 32, right).

There are two ways of selecting paths: on the one hand, the path can be selected from the path lists. When a certain row in those two list is tapped, \texttt{LFPathsPlayingManger} will create an \texttt{LFPath} array with only one path inside. On the other hand, if the user selected multiple paths from the calendar by specifying the starting date, \texttt{LFPathsPlayingManger} will load an \texttt{LFPath} array with only many paths inside. For each \texttt{LFPath} array, a \texttt{LFPathPlayingManager} will be created and be in charge of the playing management of that particular array of \texttt{LFPath}. The detailed demonstration of path selection can be found in Section 2.4.1.

Currently the property \texttt{paths} in \texttt{LFPathsPlayingManger} (which is of the type \texttt{[[LFPath]]}, array of array of \texttt{LFPath}) will only have one element inside; that is to say, no matter how many paths are selected for animation, they will all be put within one \texttt{LFPath} array. This is because all the paths are to be play in sequence. In the future, if multi-path playback
animation is implemented, there will be more than one LFPathPlayingManager active at
the same time; in this case, the singleton LFPathsPlayingManager will coordinate the
animation of all the LFPathPlayingManagers.

Back to the path selection process, selecting path from the path lists is straightforward. The relevant view controller will simply interact with LFPathsPlayingManager and pass the only path associated with the tapped row. Another way is a bit more complex: when the user taps on the calendar button, a new calendar view controlled by LFPlaybackCalendarViewController will be presented, and a calendar, a subclass of FSCalendar, will appear for the user to select the starting date. Before the user enters the calendar page, all dates with valid paths will be loaded by the method loadDateData in LFPlaybackViewController. The loadDateData method will perform the following tasks:

1. Fetch all the valid paths from SpatiaLite database by calling getAllPaths method.
2. Set the time of the start date timestamp to 00:00 and time of the end date timestamp
to 23:59 because we only care about the date here.
3. Append the (startDate, endDate) tuple to the availableDates array.
4. Call mergeDates method to merge all the date with overlaps. The algorithm used in
mergeDates is quite simple, just sort the availableDates according to the start
date, then do a linear scan to merge all the overlapped time range. The whole
merge step should have a time complexity of $\Theta(N \log(N))$.

With the available dates, we configure the FSCalendar appearance by overriding the
method calendar(calendar:appearance:fillDefaultColorFor:), and we use different background colors to highlight the dates with paths in the calendar view. (see Figure 33, left, the dates with blue background are the dates with paths). Moreover, we disable the selection of dates before the first available date and the dates after the last available date by the method calendar(calendar:shouldSelect:at:). Now the invalid starting date will not be selectable (see Figure 33, middle). After the user successfully select a starting date, the background color will be changed to white, and now the “show” button appears, allowing the user to go back to the playback page to view the animation with
There are three different states for path replay: stopped, paused and playing, which are specified in the PlaybackState enumeration type. Initially, the user is in stop state. In this state the control panel view will be like the middle screenshot of Figure 32. Once the user taps on the play button, it goes into the playing state, and then the control panel will be like the right screenshot of Figure 32. The user can control the state of the playing. The state transition diagram is show in Figure 34. Notice that it is impossible to go to “paused” state from the “stopped” state directly.

Once the user starts the animation, LFPathPlayingManager will compute the map region that covers all the points in the path with MKMapRectMake and MKMapRectUnion, and a camera transition will be performed to make the whole unioned region visible on the map view. Then an MKPointAnnotation with the first coordinate from the path is added to the map. Together with the MKPointAnnotation, an MKPolyline consists of all the coordinates from the playing path will be drawn on the map. Initially, we did not plan to add this preview path. After some rounds of user testing and consultation with our
supervisor, we discovered that showing a path offers the user a better understanding of the current status. Additionally it helps the user to follow so that he or she will not be confused by the animated annotation that randomly moving here and there.

To animate the annotation, we have tried two different approaches:

1. Fix the annotation and move the map.
2. Fix the map and move the annotation.

The first way can be found in navigation applications like Google Maps. When the navigation starts, the user location annotation will be fixed to the center of the screen, and the map will move relative to the annotation. However, this is not very suitable for our application. During the playback, unlike when people are walking or driving with the navigation applications, the animated annotation may move very fast (it makes more sense to spend 5 seconds to play back the path recorded for 1 hour than spending 1 hour to play back the path recorded for 5 seconds). If we keep the annotation fixed and animate the map, the user will probably see a lot of information moving quickly on the screen but could not follow what is going on.

The second way is quite common in general animation applications. Because of the incom-
compatibility of the first approach with our application discussed in the previous paragraph, we have decided to use this strategy for our animation implementation.

During the animation, we iterate through all the points in the path to be played in the LFPathPlayingManager, and keep track of delay variable that gets increased by a certain animation interval for each point. Then the manager will schedule a swift Timer with the corresponding delay to render the UIView animation. The outcome of this will be that the point annotation is moving along the path and the duration of animation will depend on the length of the path.

Besides the delay variable, we also keep four indices: playingPathIndex, playingPointIndex, pausePathIndex and pausePointIndex. These indices are useful for the pause/resume management.

Section 4.1.6 is written by Mingyu.

### 4.1.7 Photos

The photos section (the third tab) is where users can find out the locations of the photos in their iOS photo library. If the EXIF information of the photo contains GPS data, the GPS data will be used directly. Otherwise, the application will make and present the best guess according the the photo’s creation time and the time indexed data stored in our database.

There are two view controllers involved in this section:

- LFPhotoViewController manages the image picker view (Figure 35, left).
- LFPhotoDetailViewController manages the image detail view (Figure 35, middle) and the image location map view (Figure 35, right).

#### 4.1.7.1 LFPhotoViewController  
Due to the possibilities that a user may store thousands of photos in their photo library, how to present them all in a single view becomes a challenging issue. A naive solution may be fetching all of the photos from the secondary storage into the main memory first before presenting the view to the user.
Suppose each photo has a size of 2 megabytes and there are 2,000 photos stored inside the user’s photo library, a conservative estimate of the memory usage is:

$$2\text{ MB} \times 2000 = 4\text{ GB}$$

A 4Gb usage might not seem very large in today’s context. However, as we compare it to the RAM size of iPhones (Table 4), we find that it is already way beyond the physical memory limit, and we have not even considered the fact that kernel and other application processes would also need memories. Such a simple analysis has already opted out this naive solution.

The approach we took is keeping the reference to the image data, and loading the actual data only when the image is about to be visible on the screen. Additionally, in order to provide a smooth user experience, we also implemented a caching mechanism so that the
images will be preheated a little bit before it is actually needed. The image reference and the caching mechanism will be discussed in detail in section 4.2.4.

As a matter of fact, the memory constraints do not only come from the massive data we are dealing with, it also comes from the maintenance of view hierarchies. In the collection view, each image corresponds to a logical cell, thus 2000 images in the photo library means 2000 logical cells. Even if it requires much less memory to store the cell specifics than the image data, it is still considered a big waste to have an actual cell object allocated for each logical one because only around 30 out of the 2000 cells will be shown on the screen at the same time. The approach to avoid this waste is to reuse the cell objects, as promoted by Apple (2013).

In the cell reuse mechanism, each type of the cells are associated with a `reuseIdentifier`. When a logical cell is about to be presented, we will first check whether there are available physical cells in the reuse pool with the same identifier as we need. If there are, we will then reuse the original cell instead of creating a new one. When the physical cell is moved out of the visible area of the screen, it is recycled and put back to the reuse pool. In this way, the number of physical cells we need allocate will only relate to the cell and screen size, and the memory usage will not scale as the number of logical cells increases.

Another approach we took to guarantee the smooth scrolling is that the images are always loaded in a progressive manner. As you will see in section 4.2.4, our back-end supports the loading of different versions with different resolutions for the target images. For each image, we will attempt to load the lowest resolution first, perhaps only 10 by 10 pixels, and update the view when higher resolution images arrives. The update process is animated to lower the UI interference to the minimum. In fact, the actual case is that users can hardly perceive the updates while they are scrolling the collection view.

### 4.1.7.2 LFPhotoDetailViewController

As mentioned in the subsection 4.1.7.1, `LFPhotoViewController` keeps a list of references of the photos. The reference of a photo contains not only the pointer to the actual image data, but also other text based meta informations about the photo, which includes the `creationTime` we needed. When
the user tap on an image cell in the picker, an instance of LFPhotoDetailViewController is created, fed with the specific image reference, and presented on the screen.

The image view, as shown in the middle picture of Figure 35, is loaded using the same progressive manner as discussed in the previous section. The only difference is that it has a much higher resolution. This image view is draggable. Dragging it down will dismiss the view controller on touch ends while dragging it up will introduce a map view with a nice parallax animation.

The map view is for showing the recorded or estimated location of this particular photo. As mentioned previously, the EXIF information of a photo may contain the accurate GPS data already. In such a case, we do not make any estimation, but instead, visualize the GPS data directly on the map. From the user’s point of view, this appears as a pin, without the cyan circular range indicator.

However, there are cases where the EXIF information does not contain any location data. This may happen if the image is import from a digital camera without GPS units. In this case, we take the creation time of that image, and query our database to get the closest previous and subsequent data point, through the facade of LFDatabaseManager (which will be discussed in section 4.2.2). There are two different conditions required different actions regarding the result returned by the LFDatabaseManager:

- The result does not contain previous or subsequent (or both) point. This means the photo is taken before the first path or after the last path. There is no way for us to estimate the location, nor we can give a reasonable error range. Therefore, we do not add anything onto the map in this case.

- The result contains both previous and subsequent point. This means the photo is taken either in the duration of specific path or in the space between two recorded paths. In either case, we draw a pin at the middle of the previous and subsequent point, with an cyan circular range to represent the error.

For the second case, we do note that there are possibilities that the photo is taken between two paths, and the time interval between the two paths are too long for us to give a reasonable guess. We have not picked such cases out due to the time limit, but we
4.1.8 Statistics

The statistics section (the fourth tab) is one of the bells and whistles we added to the application in order to make it more interesting and motivating. The country data are gathered through an offline reverse-geocoding library and are stored in the Realm cache database. There are three views involved in this section:

- A overall view, which is controlled by `LFStatisticsCollectionViewController`, to show world or different continents (Figure 36, left)
- A detail view, which is controlled by `LFFlagViewController`, to show the exact visited countries in the selected domain (Figure 36, middle)
- A preview view, which is controlled by `LFMapPreviewViewController`, to show the exact places you have been to in a selected country (Figure 36, right)
4.1.8.1 LFStatisticsCollectionViewController This controller, as a subclass of UICollectionView, serves as the root of the whole statistics section. It retrieves country data from the database and feed them to the related view controllers (including itself). There are seven tappable tiles in the view, representing the 6 continents (no Antarctica) and a world as a whole. When a tile is tapped, it presents an LFFlagViewController instance and brings the user to the country details of that domain. The number of visited countries and the total country number in the corresponding domain are shown in each tile.

A standard paged UIScrollView (the superclass of UICollectionView) are only able to be paginated according to the width of its frame. This means if the tile width does not equal to the frame width, no tiles except the first one can be centered correctly. To address this issue, we subclassed the standard UICollectionViewFlowLayout, and implemented our own desired behavior by overriding the method targetContentOffset(forProposedContentOffset:withScrollingVelocity:).

4.1.8.2 LFFlagViewController As mentioned previously, LFFlagViewController controls the view which presents a collection of flags. It is designed to maintain a 2D list of LFCachedCountries, and to render the flags of these countries on the screen. Each row of the 2D list corresponds to a section in the collection view. The country in the list may or may not have been visited. A visited country will be lighted up while an unvisited country will be grayed out.

Every flag shown in the collection view is tappable. Tapping on them will lead to the presentation of a map preview, which will be discussed in the next subsection.

4.1.8.3 LFMapPreviewViewController This is a relatively simple view controller to presents a map view according to the given region. The showing of history records are achieved through the same way as in section 4.1.4.

Section 4.1.8 is written by Jinghan.
4.1.9 Settings

In Section 4.1.1 Fundamentals, it’s mentioned that the customized tab bar has different snapping levels. The first snapping level is for the specific controls for each of the views, and the second one is the homogeneous entry point for the settings page. Two features are implemented in the settings page:

- Reconstruct database
- Flush buffer path

4.1.9.1 Reconstruct Database Occasionally, some corrupted data might be introduced in the cached database, since the rendering of path history view is based on the data stored in the cached database, corrupted cached data could cause incorrect rendering result. Therefore, the user is provided a way to recover the cached data from the main SpatiaLite database.

Since the reconstruction process is iterated zoom level by zoom level, it is not efficient to make the reconstruction back-end method to interact with the front-end progress view after every single point is migrated; the progress view should be updated level by level as well. There are two possible simple progress percentage calculation algorithm can be adopted here:

- Calculate the percentage from the completed levels and total levels (21 in this application map settings):

  \[ p = \frac{n_c}{21} \times 100\% \]

  where \( p \) is the progress percentage and \( n_c \) is the completed levels.

- Calculate the percentage from the estimated completed points and the total completed points; since the rendered grid size is calculated by:

  \[ s = \frac{1}{2^n} \cdot \frac{w}{10240000} \]

  where \( s \) is the grid size, \( n \) is the zoom level and \( w \) is the width for MapKit world size.
It can be assumed that estimated number of points in level \( n \) will have four times of points in level \( n + 1 \).

Thus the estimated progress percentage can be calculated as:

\[
p = \frac{4^n c}{4^{21}} \times 100\%
\]

where \( p \) is the progress percentage and \( n_c \) is the completed levels.

Since fewer points are associated with smaller zoom level, if the progress percentage is calculated following the first algorithm above, the progress view will go much faster at the beginning, then slow down as the reconstructing zoom level becomes larger. This gives the user a false hope that the reconstruction will finish shortly but the outcome is that it goes slower and slower. It is not a good user experience and the feedback information from the progress view is not accurate. The problem can be addressed with the second algorithm; although the number of points is still under estimation, it is more accurate since the different numbers of points in each zoom level are taken into consideration.

For more information regarding the cached database and database reconstruction, see relevant section 4.2.1 \textit{Fundamentals} and 4.2.3 \textit{Realm}.

\subsection*{4.1.9.2 Flush Buffer Path}

This is where the user can flush the buffer path manually as discussed in Section 4.1.3 \textit{Geo-Location Information Recording}.

Section 4.1.9 is written by Mingyu.

\section*{4.2 Back-End Details}

There are five subsections regarding the back-end development details. The first subsection \textit{Fundamentals} will critique the available options and discuss how we ended up with our current back-end architecture. The rest will concentrate on the discussion of how we applied different technologies to implement the design.
4.2.1 Fundamentals

Before we diving into the discussion on our choices among the available database technologies and the rational behind those choices, a restatement of the requirements would be helpful for making clear of what are the desired features. The following are the features that require data persistence support and their corresponding requirements for the data persistence module:

- showing the aggregated path records:
  - requires massive aggregation operations through the entire database
  - requires querying according to geological regions (longitude and latitude)
  - requires outputs to be bounded (otherwise front end may be overwhelmed)
  - compromise on precision may be acceptable depending on zoom scales

- showing path records according to time
  - requires querying according to time
  - requires loss-less precision on a selected portion of the data
• showing local photos
  – requires querying according to time (locations)
  – requires fast loading of massive data (photos)
  – requires querying on different resolutions (photos)

• importing and exporting for path sharing
  – requires convenient access to portable files
  – requires secure serialization
  – requires easy separation between shared paths and regular paths

According to the requirements specified above, the application overall requires 3 different
data persistent components:

• a component as the essential component to persist users’ regular records
• a component to manage the photos
• a component to manage the incoming and outgoing path files

Apple has provided some well-established, native solutions for component 2 and 3 and they can fit into our case very well. Therefore, the design choices are clear regarding these two and we will discuss them in detail in subsection 4.2.4 and 4.2.5. However, for component 1, there is no clear solution to address all the requirements listed. The following four are the available technologies we have researched, namely Core Data, SQLite, Realm and SpatiaLite.

Core Data is one of the most popular data persistent approach on iOS. It is provided and maintained by Apple and integrated into the operating system natively. The large user base of Core Data results in a very comprehensive community to seek help from. However, Core Data is not designed as a database, but an object graph manager. Therefore, a lot of functions in Core Data are aiming to make the object relationships accessible and organized rather than pursuing high performances. In our case, the only entities are point, path and country, a complicated object graph manager is therefore not required.

SQLite is the base of Core Data, but without all the object graph management overhead.
So naturally, if Core Data was not suitable enough, the lower level SQLite turns to be the second consideration. SQLite is also supported natively by iOS and it is faster than Core Data, but as we taking out the Core Data wrapper, raw SQLite is much harder to setup and use. There will have to be vast number of manipulations on C pointers in Swift, which are often tedious, unintuitive and messy. Moreover, SQLite by default does not support multi-threading. In our use case, when the map is loading, each tile will load from a different thread. If the database module can only be accessed from one single thread, all database requests would then have to be lined up in a queue, which defeats the original purpose of the map’s multi-thread loading. Furthermore, with SQLite, we would have to write the SQL statements in the form of strings, which is error-prone and sometimes maybe hard to debug. All these hassles made us wonder whether there’s a better choice.

Realm is the next one we examined. This is a much younger framework comparing to SQLite. In general, there are three characteristics of Realm to make it particularly attractive to us. Firstly, Realm operates in a synchronous, but lazy manner. This is important to us because when loading the map tiles, the life time of each loading thread is controlled by the OS rather than ourselves. Asynchronous loading may result in a condition that the thread is deallocated before the data arrive. Secondly, in Realm, data are exposed as objects and thus it can fit into our objective oriented codes very easily. This appears even more appealing when comparing to the complicated setup and querying steps of SQLite. Thirdly and most importantly, Realm is fast. Figure 38 shows the performance comparison between Realm and other popular data persistence solutions including SQLite and Core Data. (Realm (2014))

Realm manages to address most of our requirements; however, there is still no way to perform appropriate sampling operations. As mentioned in the requirement, we would like the database omits some of the records based on the zoom scale on which the query is performed. With the consideration that this should be a common requirement for every application that deals with massive geo-location data, we started searching for specialized spatial databases to fulfill our needs.

SpatiaLite is the only spatial database we found which can be integrated into an iOS
Figure 38: Performance comparison between Realm and other solutions

application without too much hassles. It supports all the required spatial functions
to perform the sampling task, such as simplifying paths and snapping points to grid.
However, as an extension of SQLite, SpatiaLite suffers from the same sets of problems,
and most of the time, the problems tend to be more severe due to its small user base and
poor documentation.

There is no single perfect solution here to fully address all requirements at once, but
fortunately, we don’t have to stick with only one of these options. A combinatorial scheme
is to use SpatiaLite as the base-level database for storing and processing every detail of
the recorded user data, and then to use Realm as a cache database in between to allow
fast querying on the processed data when loading up the map. When new data entries
arrives, it will be first stored into the base database, then the base database perform the
aggregation operations and update the cache database with the new result. In this way,
computations are kept decentralized and thus the querying performance is improved. In
the extreme cases where some inconsistencies occur between the two databases, it is also
possible to reconstruct the entire cache database from scratch. The combinatorial scheme
finally fulfilled all the requirements, and thus, it is chosen as our final solution on the
data persistence component design.

Section 4.2.1 is written by Jinghan.
CREATE TABLE tracks (  
  track_id INTEGER PRIMARY KEY AUTOINCREMENT,  
  track_geometry LINESTRING NOT NULL  
)

Figure 39: SQL schema for table tracks

4.2.2 SpatiaLite

All I/O operations with SpatiaLite database are handled by the singleton instance of the class LFDatabaseManager.

As mentioned in section 4.2.1, SpatiaLite is used as the base database that stores and processes all recorded geo-location and time data. The only table here is table tracks, the schema of which is shown in Figure 39.

4.2.2.1 Spatial Functions and Point Sampling  SpatiaLite allows eight geometry types in the table, including

- GEOMETRY
- POINT
- LINESTRING
- POLYGON
- MULTIPOINT
- MULTILINESTRING
- MULTIPOLYGON
- GEOMCOLLECTION

Among the eight types, POINT and LINESTRING are considered as the most relevant ones. As you might have already noticed from Figure 39, we store the geo-location data as tracks. Each track is made up of a LINESTRING and an identifier. Each LINESTRING consists of one or more four-dimensional POINTs, and the four dimensions of every single POINT are: longitude, latitude, altitude and time.

The supporting website of SpatiaLite (2017) has provided a lengthy list of all supported SQL functions. The ones we are actually using in the project (13 in total) are shown
in Table 5. Among the 13 functions, MbrContains, MbrOverlaps, DissolvePoints and SnapToGrid together make up the core of our point sampling logic.

When the cache database (which we will discuss in detail in section 4.2.3) requests a point sample by passing in the desired grid size and region, LFDatabaseManager will follow the following steps to produce the results:

1. Create a reference Polygon object according the given region
2. Use MbrOverlaps and MbrContains to find out every geometry stored which is either partially overlapped with or entirely contained in the reference polygon.
3. Perform the first SnapToGrid operation on the output of step 2 according to the given grid size.
4. Get the intersection part of the reference polygon and the output of step 3.
5. Perform an GUnion operation to group the output of step 4 as a single geometry.
6. Perform the second SnapToGrid operation to reduce the possible duplicated points.
7. Perform the DissolvePoints operation to break the geometry and get all the points.

Please note that SnapToGrid is performed twice with different purposes. The one in step 3 is for reducing the points amount as the intersection operation in step 4 is a costly operations. This SnapToGrid is performed on each individual geometries and guarantees that there is no duplicated points within the same geometry. However, the duplicates may still exist among different geometries. Therefore, we need to perform the operation again after merge all geometries together. The end product of these steps will be a set of points, snapped to a grid of the specified size and without any duplicates.

4.2.2.2 WKB, WBT and GeoJSON  The Well Known Binary, Well Known Texts and GeoJSON are the three standard formats we used to represent geometry information. They are serving as the bridges between the the geometry models that SpatiaLite can understand and our own model which are used across the application. Each of three types has its own strengths and weaknesses and thus they are used in different places in the application.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsBinary</td>
<td>Returns the Well Known Binary representation</td>
</tr>
<tr>
<td>AsGeoJSON</td>
<td>Returns the GeoJSON representation of the geometry</td>
</tr>
<tr>
<td>DissolvePoints</td>
<td>Returns a Geometry. The input Geometry is arbitrary: any POINT will remain unaffected, but any LINESTRING or RING will be dissolved into elementary Points corresponding to each Vertex.</td>
</tr>
<tr>
<td>EndPoint</td>
<td>Return a Point containing the last point of the input curve</td>
</tr>
<tr>
<td>GeomFromText</td>
<td>Constructs a geometric object given the provided Well Known text Representation</td>
</tr>
<tr>
<td>GUnion</td>
<td>Returns a geometric object that is the set union of the two input geometries.</td>
</tr>
<tr>
<td>Intersection</td>
<td>Returns a geometric object that is the intersection of the two input geometries.</td>
</tr>
<tr>
<td>M</td>
<td>Returns the M (in our case, the time) value of the point</td>
</tr>
<tr>
<td>MbrContains</td>
<td>Returns TRUE if the second geometry’s MBR is completely contained in the first geometry’s MBR</td>
</tr>
<tr>
<td>MbrOverlaps</td>
<td>Returns TRUE if the intersection of the two input geometries’ minimum bounding rectangles (MBR) results in a value of the same dimension as the two geometries that is different from both of them</td>
</tr>
<tr>
<td>Simplify</td>
<td>Returns a geometric object representing a simplified version of the curve by applying the Douglas-Peukert algorithm with given tolerance.</td>
</tr>
<tr>
<td>SnapToGrid</td>
<td>Returns a new Geometry corresponding to the input Geometry; all points and vertices will be snapped to the grid defined by its origin and size(s).</td>
</tr>
<tr>
<td>StartPoint</td>
<td>Returns a Point containing the first point of the input curve</td>
</tr>
</tbody>
</table>

Table 5: SpatiaLite SQL functions used in the project
The inputs for SpatiaLite are in WKT formats. WKT is an simple string based format, which can be easily generated using regular Swift string manipulation routines. String manipulations are slow, but more intuitive. Since we are going to segment a long track into several short ones before storing it to SpatiaLite, the data size for each conversion will not be very large and thus the performance differences are tiny or even imperceptible. Therefore, we chose WKT as bridging format for SpatiaLite inputs.

However, the situations for outputs are different. There are possibilities that the result of a single request contains thousands or more data entries. The time penalty of the conversion between strings and different models when using WKT is unacceptable. In contrast, WKB is more compact than WKT in terms of byte usage, and there exists some handy libraries which make us capable to easily initialize Objective C objects directly from the binary. This avoids wasting the precious CPU cycles unnecessarily on string conversions and thus WKB is chosen as the main bridging formats for SpatiaLite outputs.

While WKB is the output formats in most cases, there are still some exceptions. The string based GeoJSON serves as another output format in some particular cases. GeoJSON is an established format used across different platforms and frameworks. Some of the frameworks, such as Mapbox mentioned in section 4.1.2, are able to read in a raw GeoJSON and render the data directly on a map, which is a one-step solution and avoids all conversions between different models directly. Although currently we are using MapKit as our map rendering framework and MapKit does not support GeoJSON, we are still open to other framework options in the future. Therefore, we decided to develop and preserve the GeoJSON related APIs.

Section 4.2.2 is written by Jinghan.

4.2.3 Realm

As described in Section 4.1.4, when the user is at different zoom level in the history overview page, the points fetched from database will be different. The zoom level information can be passed to the LFDatabaseManger with the method getSnappedPathsInRegion(region:gridSize:completion:).
Undoubtedly, this is an inefficient process, each time a new tile is to be rendered, the SpatiaLite database will need to handle a query scanning through the entire database of recorded points. Furthermore, before the paths eventually get returned, it need to be simplified and snapped to grid since we want to render the visited places in a pixelated style. Actually there are lots of repetitive calculation here, since the user may not record new path all the time (he should not be travelling all the time in all different places), and each time the user view the history overview, he will probably be querying for the same \texttt{LFP\textup{ath}} all the time. In order to get rid of the redundant queries and improve the general rendering efficiency, we create a Realm cache database between the SpatiaLite database and the front end component.

There are basically three types of Realm object stored in the cached database: \texttt{LFCachedPoint}, \texttt{LFCachedLevel} and \texttt{LFCachedCountry}.

\subsection*{4.2.3.1 \texttt{LFCachedPoint}} Unlike the point we store in the SpatiaLite database, the cached points are mainly for the purpose of history overview rendering. Hence, the time information is not important here. Other than the coordinate information of longitude and latitude, \texttt{LFCachedPoint} owns another property: count. The reason behind this count property is that we color the rendered points in history overview page by the frequency the user visit that location. Instead of storing all the duplicated points in the database and count the query result every time, the \texttt{LFCachedPoints} with the same coordinate information will only appear at most once in the cache database. For any new point recorded with the existing coordinates, we simply increment the count property by one.

\subsection*{4.2.3.2 \texttt{LFCachedLevel}} Different points are rendered in different zoom level, but it is also possible that for the same point to appear in different zoom level (see Figure 40, notice that the point with the same coordinate at the T intersection appears in all three different zoom level, with different grid size).

Therefore, \texttt{LFCachedLevel} and \texttt{LFCachedPoint} have a many-to-many relationship.
4.2.3.3 **LFCachedCountry**  The LFCachedCountry model keeps track of a list of different countries in the world where the application will check if the user has visited a specific country based on the geo-location information the user has in the database. The result is used in the statistics page (see more details in Section 4.1.8 *Statistics*). Every time a new LFCachedPoint with new coordinate is added, the LFRReverseGeocodingManager will find out the country that the new point is located in, and LFCachedDatabaseManager will update the visited flag for the LFCachedCountry model.

When the user starts the database reconstruction process from the settings page, the current Realm cached database will be removed, and the data of the cached points, cached level, and cached country will be re-generated from the SpatiaLite database.

Section 4.2.3 is written by Mingyu.

### 4.2.4 Photos

All I/O operations related to photos are handled by the singleton instance of the class LFPhotoManager.
Since one of the functions of our application is to help user to find the locations for the photos inside their system photo album, we need some back-end supports to get photos out, process and present them to the user. One of the native iOS frameworks, Photos, has provided a mature solution.

4.2.4.1 Fetching a Photo  We have two terminologies here: asset and image. In our context, an asset refers to a representation of an image, which contains the information related to that image. An image refers to the image data which is required by the GPU to render it on screen. An asset keeps a pointer to the image it represents rather than contains the data itself. When querying, we are just querying for the correct assets, and the image data for the assets will not be fetched until the photo manager is explicitly instructed to do so. This reduces the required I/O and increases query speed to a large extend. In this scheme, if we want to render an image from the Photos library on screen, two steps are required:

1. Fetch the asset according certain predicates
2. Fetch the image according to the asset

In the project, the first step is done through the static method fetchAssets(with:) of the class PHAsset. This methods accepts an PHFetchOptions object as the parameter. In the PHFetchOptions object, we can specify the desired sorting order and the predicates we want the results to fulfill. On calling, this method will return an PHFetchResult object synchronously. Subsequently, according to the formal documentation provided by Apple (2016a), the PHFetchResult object will be loaded dynamically on need to provide the optimal performance.

The second step is done through an instance method from the PHCachingImageManager class: requestImage(for:targetSize:contentMode:options:resultHandler:). This method will produce a UIImage instance with the specified requirements (e.g. size, etc.) and return it in an asynchronous manner. The UIImage can then be used directly in the rendering process.
4.2.4.2 Caching a Photo  As we can see from section 4.1.7, photo browsing involves thousands of images being loaded. We do not want to perform a fresh load from the disk every time when an image is required. Moreover, sometimes we also would like to preheat some of the images for better user experience. As a result, we decided to implement a cache system for photos. Fortunately, PHCachingImageManager provides native supports for such needs.

There are two related instance methods of PHCachingImageManager:

1. startCachingImages(for:targetSize:contentMode:options:)
2. stopCachingImages(for:targetSize:contentMode:options:)

These two methods accept the exact same sets of parameters to tell the image manager what to cache and how to cache. As the names suggested, the first methods indicates the need to start caching on the given assets while the second one indicates the need to stop. After calling these two methods, PHCachingImageManager will handle the caching process internally and the available image cache will be used in normal load requests directly so that the rendering component APIs are not affected by the caching unit.

Section 4.2.4 is written by Jinghan.

4.2.5 Direct Files

All direct I/O interactions with local file systems are handled by the singleton instance of the class LFLocalFileManager. These interactions are mainly occurred when the user shares a path to, or accepts a path from his or her friends. Due to the fact that most users will not want the incoming paths to be mixed with their own paths, we decided to keep these separately from the user’s main database. Since paths are serialized and sent as a single binary file, it is more convenient to just keep and manage them in the local file systems than creating a separate table in the SpatiaLite database. A different model, LFIncomingPath, is defined and used here. An LFIncomingPath instance serves as a wrapper of an LFPath instance, with some extra file related parameters such as fileName and url.
4.2.5.1 NSCoding  The serialization and deserialization are done by make the classes LFPath and LFPoint implement the NSCoding protocol. Comparing to the other options we had, such as JSON files or property lists, the main advantages of NSCoding are two:

- It is supported by iOS natively and can be used hassle free.
- Its child NSSecureCoding provides an easy but secure encryption.

There are two methods declared in this protocol, which are essential for the instances of a classed to be encoded and decoded.

- In the constructor init(coder:), we extract the properties from a given decoder, init a clean instance and assign the properties to that instance. In this way, deserialization is achieved.
- In the method encode(with:), we insert the properties of current instances into the given coder, which will be used to create the binary data or a file. In this way, serialization is achieved.

4.2.5.2 LFLocalFileManager  The singleton LFLocalFileManager manages incoming path events and local file hierarchies. There are three essential methods defined in LFLocalFileManager:

- The method getAllIncomingPaths reads all the files in current Documents/Inbox directory, serializes them and returns an array of LFIIncomingPath instances.
- The method removeFile(at:) takes in a file path string and removes the corresponding file at the path if it exists.
- The method handleIncomingFile(with:) is called when a new file is added to the Documents/Inbox directory. It will subsequently present the list view of to show all incoming paths.
Please note that there is no method to create a file in the Documents/Inbox directory. This is because the file will be added to the directory automatically by the operating system when it is passed from another application.

Section 4.2.5 is written by Jinghan.
5 Conclusion

5.1 Summary

After eight months of design and implementation, we successfully make Leifr a usable product with most of the proposed features completed including:

- recording a path of user’s geo-location information (longitude, latitude and altitude) with the exact time that this information is recorded
- visualize the visited places on the map and color it in a “heat map” manner to indicate how frequent the user visit a specify location
- view/share/replay a recorded path
- replay all the paths sequentially starting from a selected date
- predict the location where the chosen photo is taken
- display the statistics about all the countries the user visited

The implementation progress involves different forms of technology ranged from front-end (map frameworks, iOS design pattern, rendering and animation) to back-end (SpatiaLite, Realm, reverse geo-coding and files management). With the help of team project management tools like git and CocoaPods, this final year project also becomes a good practice of software engineering.

There are still some incomplete abandoned features due to certain limitations, more about the future of Leifr will be discussed in the following sections.

Section 5.1 is written by Mingyu.

5.2 Difficulties and Limitations

Although we are cheering about the progress we have made, we did encounter certain difficulties or limitations during the development process. We managed to overcome or find workarounds for some of them, but for others we had to make the trade-offs.
It was really hard to get started with the SpatiaLite database we are dealing with, as we had expected. There was almost no third party learning resources online to provide guidance on how to setup and initialize the database. Furthermore, the documentation is very poor and most of the time we had to do the trials-and-errors process again and again before get our desired behaviors. Fortunately, we did find a way out in the end.

Another difficulty comes with the nature of our projects. Since this is an application that aim to record, manage and visualize geo-location data, some testing had to be carried out in the field, which is very inconvenient and time consuming. The iOS simulators was considered as an alternative because of the location simulation it provides. However, the user experience to operate a simulator is still quite different from operating the actual device.

A third one is about the statistics section. We planned to add in much more statistics than what we have currently, such as visited cities, sights, etc. The main problem here is that we do not want this application depend on the Internet connection. For countries, the data set is relatively small, can be stored as compact as one single GeoJSON file and used as a offline resource for reverse- geocoding, but this does not apply to cities or sights due to the simple fact that there just too many things to store.

Some of the difficulties were tough, but we do appreciate their existence. We treat this project not only as a way to demonstrate what we have learnt during the four-year undergraduate study, but more importantly, a process for us to learn and apply new things. People said difficulties made them learn, and we did feel this way.

**About Testing**

Testings are not performed formally in this project because of several reasons. The most essential one is from the time and manpower constraints. A systematic testing usually requires non-negligible efforts from the entire team: in the industry there are even dedicated people whose job is solely to take care of the quality assurance part of a project. Given the eight-month time limit and the two-people constraint, we chose to strike for a tasty product with more interesting features instead of devoting ourselves into
pursuing the code coverages.

However, although formal testing was not performed, we did write some test cases during the development process. These are mostly unit tests on our back-end services to help us gain some confidence on each individual unit before we integrate them with each other. In some other cases, these tests are also served as the module drivers for us to examine the back-end execution before actually developing the corresponding front-end user interfaces.

We do understand the importance of testing, but this was a hard trade-off we had to make.

Section 5.2 is written by Jinghan.

5.3 Future Plans

By now, the application has been implemented as a viable product. The users can record paths and play around with the data. However, due to the time constraint and some limitations and difficulties discussed in Section 5.2 *Difficulties and Limitations*, the application developed so far is imperfect. We are going to continue the development in the future, other than fixing some existing defects and bugs, we have some plans in mind to work towards. Here is a list of some selected features and improvements we want to bring to this application in the near future:

- Support multi-path playback animation, which allow a user to compare the path shared by his friend with his own path at the same time period.
- Increase the accuracy of photo location estimation by taking the temporal gap between different points in a path into account.
- Follow the updates from Mapbox framework and try again to integrate with it in order to present a nicer map.
- Try to bring more interesting statistics by analysing the data collected from the recorded path.
- Support cloud backup of the geo-location information in case of the loss of data.
• Provide support for “path repairing” (e.g. remove the extra point caused by the inaccuracy of GPS data and concatenate several paths that are supposed to form a complete path)

• Implement more features specially designed for travellers (e.g. automatic journal generation), which corresponds with our initial goal for working on this project.

Section 5.3 is written by Mingyu.
Project Contribution

Since this is a complete software engineering implementation team project, the division of work might not be very clear-cut, there are many intersection between the contribution from Jinghan and Mingyu, and the report is also completed jointly.

Although the work is not strictly equally distributed, we tried to make it reasonable that each of the individuals has critical contributions to certain key components of the project. To have a glance of the rough project distribution, some statistics might be useful:

- In terms of project development, from Sep 11, 2016 to Apr 4, 2017, 187 commits are authored by Jinghan, and 146 commits are authored by Mingyu.
- In terms of report writing, there are 70 pages in the main report section with a surprisingly even cut: 35 pages are written by Jinghan, and 35 pages are written by Mingyu.

Detailed distribution of the work between Jinghan and Mingyu is shown in the following pages. In all tables, J stands for Jinghan and M stands for Mingyu.
Main Report

We have already added the author information for each individual section of the main report as inline foot notes. The table below is to provide an overall idea of how the writing work is distributed. The main report excludes the following sections:

- Abstract
- Acknowledgements
- List of Figures
- List of Tables
- Contents
- References
- Project Contribution
- Program Listing
- Installation Guide

The contribution for these sections are introduced separately in the table next page, Report Appendices.

<table>
<thead>
<tr>
<th></th>
<th>J</th>
<th>M</th>
<th></th>
<th>J</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>-</td>
<td>Major</td>
<td>4.1.3</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>1.2</td>
<td>-</td>
<td>Major</td>
<td>4.1.4</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>1.3</td>
<td>-</td>
<td>Major</td>
<td>4.1.5</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>2.1</td>
<td>Major</td>
<td>-</td>
<td>4.1.6</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>2.2</td>
<td>Major</td>
<td>-</td>
<td>4.1.7</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Major</td>
<td>-</td>
<td>4.1.8</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Major</td>
<td>-</td>
<td>4.1.9</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>2.4.1</td>
<td>-</td>
<td>Major</td>
<td>4.2.1</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>2.4.2</td>
<td>-</td>
<td>Major</td>
<td>4.2.2</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>2.5</td>
<td>Major</td>
<td>-</td>
<td>4.2.3</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>2.6</td>
<td>Major</td>
<td>-</td>
<td>4.2.4</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>3.1</td>
<td>Equal</td>
<td>Equal</td>
<td>4.2.5</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>3.2</td>
<td>Equal</td>
<td>Equal</td>
<td>5.1</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Major</td>
<td>-</td>
<td>5.2</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>4.1.2</td>
<td>-</td>
<td>Major</td>
<td>5.3</td>
<td>Major</td>
<td>Major</td>
</tr>
</tbody>
</table>
Report Appendices

<table>
<thead>
<tr>
<th></th>
<th>J</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>Acknowledgement</td>
<td>Major</td>
<td>Major</td>
</tr>
<tr>
<td>Contents</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>List of Figures</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>List of Tables</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>References</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>Project Contributions</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>Program Listing</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>Installation Guide</td>
<td>Major</td>
<td>-</td>
</tr>
</tbody>
</table>

Research and Experiment

<table>
<thead>
<tr>
<th></th>
<th>J</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Data</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>Realm</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>SpatiaLite</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>MapKit + Tiled Overlay Renderer</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>MapKit + Annotation Views</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>MapKit + Overlay Renderer</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>Mapbox + Annotation Images</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>Mapbox + Annotation Views</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>Mapbox + GeoJSON Real Time Rendering</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>Mapbox Development Progress Tracking</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>MapKit Playback Animation</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>Background Mode</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>Chinese Coordinates Conversion</td>
<td>Major</td>
<td>-</td>
</tr>
</tbody>
</table>
## Implementation

### Front-end

<table>
<thead>
<tr>
<th>Class Name</th>
<th>J</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>LFViewController</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFHoverTabBarDataSource</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFHoverTabBaseController</td>
<td>Major</td>
<td>Minor</td>
</tr>
<tr>
<td>LFHoverTabDelegate</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFHoverTabViewController</td>
<td>Equal</td>
<td>Equal</td>
</tr>
<tr>
<td>LFHoverTabDataSource</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFHistoryViewController</td>
<td>Major</td>
<td>Minor</td>
</tr>
<tr>
<td>LFTTrackTabViewController</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFMyTrackViewController</td>
<td>Major</td>
<td>Minor</td>
</tr>
<tr>
<td>LFIInboxViewController</td>
<td>Major</td>
<td>Minor</td>
</tr>
<tr>
<td>LFPlaybackViewController</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>LFPlaybackCalendarViewController</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>LFPhotoViewController</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFPhotoDetailViewController</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFStatisticsCollectionViewController</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFStatisticsViewLayout</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFFlagViewController</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFMapPreviewViewController</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFFSettingViewController</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFLoadingViewController</td>
<td>Major</td>
<td>Minor</td>
</tr>
<tr>
<td>LFOverallViewController (postponed)</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>LFTTableViewCell</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFCollectionTableViewCell</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFIInboxTableViewCell</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFTTrackTableViewCell</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFRecordButton</td>
<td>Major</td>
<td>Minor</td>
</tr>
<tr>
<td>Class</td>
<td>J</td>
<td>M</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td>LFHistoryControlView</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFPlaybackAnnotationView</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>LFPlaybackControlView</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>LFPlaybackCalendarView</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>LFGridViewCell</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFPhotoControlView</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFHorizontalCollectionViewCell</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFMapCell</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFI mageCell</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFS tatisticsCollectionViewCell</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFFlagCollectionViewCell</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFButtonCell</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFS ett ingAccessoryView</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFGeoPointsOverlayRenderer</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFCurrentPathOverlayRenderer</td>
<td>-</td>
<td>Major</td>
</tr>
</tbody>
</table>

**Assets**

<table>
<thead>
<tr>
<th>Class</th>
<th>J</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Icon</td>
<td>Major</td>
<td>Minor</td>
</tr>
<tr>
<td>Continent Maps</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>Country Flags</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>Material Icons</td>
<td>Major</td>
<td>Minor</td>
</tr>
<tr>
<td>Placeholders</td>
<td>Major</td>
<td>-</td>
</tr>
</tbody>
</table>
### Back-end

<table>
<thead>
<tr>
<th>Class</th>
<th>J</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>LFPoint</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFPath</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFIIncomingPoint</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFCachedPoint</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>LFCachedLevel</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>LFCachedCountry</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFPhotoManager</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFLocalFileManager</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>LFPathPlayManager</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>LFPathsPlayManager</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>LFGeoRecordManager</td>
<td>Equal</td>
<td>Equal</td>
</tr>
<tr>
<td>LFDatabaseManager</td>
<td>Major</td>
<td>Minor</td>
</tr>
<tr>
<td>LFGeoJSONManager</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>LFCachedDatabaseManager</td>
<td>Minor</td>
<td>Major</td>
</tr>
<tr>
<td>LFReverseGeocodingManager</td>
<td>Major</td>
<td>-</td>
</tr>
</tbody>
</table>

### Miscellaneous

<table>
<thead>
<tr>
<th>Class</th>
<th>J</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>LFDatabaseManagerTest</td>
<td>Equal</td>
<td>Equal</td>
</tr>
<tr>
<td>LFGeoRecordManagerTest</td>
<td>Equal</td>
<td>Equal</td>
</tr>
<tr>
<td>LFGeoJSONManagerTest</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>LFZoomLevelTest</td>
<td>-</td>
<td>Major</td>
</tr>
<tr>
<td>LFPhotoManagerTest</td>
<td>Major</td>
<td>-</td>
</tr>
<tr>
<td>AppDelegate</td>
<td>Equal</td>
<td>Equal</td>
</tr>
<tr>
<td>UIKitExtension</td>
<td>Equal</td>
<td>Equal</td>
</tr>
<tr>
<td>MapKitExtension</td>
<td>-</td>
<td>Major</td>
</tr>
</tbody>
</table>
References


Appendix A - Program Listing

LEIFR

Views
  - LaunchScreen.storyboard
  - Main.storyboard
  - LFHistoryControlView.xib
  - LFPlaybackControlView.xib
  - LFPlaybackCalendarView.xib
  - LFPhotoControlView.xib
  - LFRadioButton.swift
  - LFNavigationBar.swift
  - LFPlaybackAnnotationView.swift
  - LFSettingAccessoryView.xib

Cells
  - Table View Cells
    - LFTableViewController.swift
    - LFTableViewController.xib
    - LFButtonCell.swift
    - LFButtonCell.xib
    - LFImageCell.swift
    - LFImageCell.xib
    - LFMapCell.swift
    - LFMapCell.swift
    - LFMapCell.xib
    - LFTrackTableViewCell.swift
    - LFTrackTableViewCell.xib

[cont.]
LFMyTrackViewController.swift
LFInboxViewController.swift

Tab 2
LFPlaybackViewController.swift
LFPlaybackCalendarViewController.swift

Tab 3
LFPhotoViewController.swift
LFPhotoDetailViewController.swift

Tab 4
LFStatisticsViewController.swift
LFFlagViewController.swift
LFStatisticsCollectionViewController.swift
LFStatisticsViewLayout.swift
LFMapPreviewViewController.swift
Appendix B - Installation Guide

The source code is available on Github, at https://github.com/Echx/LEIFR. To compile and run the application on an iPhone, you will need:

1. A Macintosh with Xcode 8.3 or above installed.
2. A valid Apple Developer account
3. A working iPhone running iOS 10.0 or above which is registered with the account.

For more information about Apple Developer Account, please refer to Apple’s official website at https://developer.apple.com/support/compare-memberships/.

If you are already with the things mentioned above, please follow the steps below to open the project, compile and run the application:

1. Download the code from master branch.
2. Double click the file . / LEIFR / LEIFR.xcworkspace. (Xcode should be opened)
3. Connect your iPhone with your Mac, and select the iPhone in the device menu, which is besides the “Run” button at the top left corner.
4. Click the “Run” button to select the “Run” option in the “Product” menu.
5. The application should be installed and launched on the iPhone.