Trip Diary

Software Engineering (14:332:452)

Report 3

Team 5

Samuel Zahner
Kinjal Patel
Vincent Chan
Nisha Bhat
Gaurav Sethi
Samuel Minkin
Yash Shah
Jonathan Banks

Submission Date

May 7, 2020

Table of Contents

Contribution Breakdown

	Sam Z	Kinjal	Vincent	Nisha	Gaurav	Sam M	Yash	Jon
Project Management	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Summary of Changes	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Customer Statement	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Glossary of Terms	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
System Requirements	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Functional Requirements	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Effort Estimation	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Domain Analysis	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Interaction Diagrams	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Class Diagram + Interface	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Systems Arch + Design	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Algo + Data Structures	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
UI + Implementation	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Design of Tests	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
History of Work	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
References	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Total Points	12.5	12.5	12.5	12.5	12.5	12.5	12.5	12.5

0. Summary of Changes

A. Feedback Implementations

1. Feedback for Report 1:

- Condense use cases (done)
- Section 1: Customer Problem Statement
    - Change to first person: “The only small issue I have is: you don’t write a report in the first person. Instead of ‘I’, you introduce ‘The app’ or ‘we’ or ‘team’”
- Functional requirements specification (section 4 in this document):
    - Actor relationships to use cases are not shown
- User Interface Design & Implementation (section 11 in this document):
    - Worst case scenarios for estimating the user effort are not considered

2. Feedback for Demo:

- Condense use cases + make them more sophisticated
- Novelty - how it’s different from other apps. Needs standout features.
- Lack of creativity with user data + user does too much work - more automation (discussed in group call)
- How actors are differentiated

3. Feedback for Report 2:

- Section 7 in this report (Interaction Diagrams):
    - Fully dressed description of use cases missing
    - Alternate scenarios and their descriptions missing

B. Enumerated Edits

Note:

1. To ensure a better user-experience within this report, there are quick links on each section header in this chart. When clicked, it will lead you directly within the report to where changes were made.
2. If you would like to return to these edits, the 'Enumerated Edits' link in the header will bring you back to this table from any page.

Section		Changes	Description
1	Customer Statement of Requirements	1. Rephrased and made clearer	1. The customer problem statement has been rephrased to meet the first person point-of-view requirement, as well as adjusted to become easier to read.
2	Glossary of Terms	No changes	N/A
3	System Requirements	1. Update requirements 2. Update UI pictures 3. Update relationship between old stake-holders and new use cases.	1. Reviewing our requirements from Report 1, we recognized that there were some requirements that we changed or removed all together as we continued to design and implement our project through and after Report 2. As a result, our number of requirements dropped from 68 to 63. 2. Since we have a functioning project, we were able to replace our designed user-interface pictures with actual screenshots of our application. 3. The "Stake-holders" section was updated to provide the reader with further information about how the stake-holders relate to (or in other words, make use of) our updated use cases. 4. New UI pictures were replaced to represent current application look
4	Functional Requirement Specification	1. Update use-cases 2. Update use-case diagram 3. Update traceability matrix 4. Update fully	1. We greatly decreased the number of use cases from over 20 use cases to just 8. This was based off of the feedback we received from our first demo. Previously, we had use cases that were too specific such as "add journal", "edit journal", and "delete journal", which were later

		dressed description (with alternate scenarios) 5. Update system sequence diagram	combined into one use case called “journaling”. By broadening the scope of other use cases we were able to decrease the count to 8 cases. The fully dressed descriptions, traceability matrix, and sequence diagrams of the changed use cases had to be appropriately updated and generalized.
5	Effort Estimation (Use Case Points)	Edited effort estimation from new use-cases	We needed to adjust the UUCW based on the updated use cases. After doing so our calculated UUCW went from 230 to 105. Our UAW and TCF remained the same. As a result, our UCP went from 231 to 112. With this new UCP, we calculated a project duration of 3136.
6	Domain Analysis	1. Make new domain model 2. Updated association definitions 3. Update traceability matrix	1. The new domain model accounts for further sophistication from the application, including cross-functionality and additional automation for the user. This was important from the feedback received from the demo, as to reduce the manual burden on the user. Such interactions included (1) journal receiving locations from the map to use for autofill (2) map receiving photos from the photos database to associate with pins, etc. Additionally, we would like to make it evident that all of these features are used by 1 user: the general traveller. But, there are several different types of travellers who cater to each feature, which are subcategories of our main user. 2. Association definitions were updated to include cross-functionality 3. The traceability matrix was updated to reflect the changes in the new domain model
7	Interaction Diagrams	1. Display new functionality 2. Update with new design principles 3. Add alternate scenarios to interaction diagrams	1. Our interaction diagrams need to display the additional functionality added to our updated use cases. The additional functionality comes from both consolidating many use cases into one and also making the use cases more sophisticated. Hence, the interaction diagrams in this report are much more complex than those in report 2. 2. We added some of the following design patterns that were taught after report 2: Command, Decorator, State, and Proxy, if they were

			applicable to the design of the use case. 3. In report 2, we only included the main success scenarios for the fully dressed use cases. This did not give the full picture as there were alternate scenarios that occurred depending on failure or different circumstances. We included these alternate scenarios to give a fuller picture of how our system works.
8	Class Diagram and Interface Specification	Section re-done	1. In the traceability matrix section there were several changes made from report 2 to our actual design because we hadn't started building out our app yet our vision of how it would be structured was very different from how it ended up becoming. We ended up using all all of this basic architecture to setup the views and controllers for the app but we also built out several other classes to organize how the data of the app was stored such as Pin which stored fields relevant to the location of the pin, Budget which stores details about budget, Retrieve which stores information about selected attractions, Journal which stores information relevant to journal entries, and many more custom classes for the actual storage of information. Information of other small changes were also included below this section.
9	Systems Arch + System Design	No changes	N/A
10	Algorithms + Data Structures	Minor revisions	Since we altered our design of how we are organizing the calendar displaying a specific day's information, the description of the data structures under the calendar section is updated. Instead of having day objects, we are utilizing a day field in the database.
11	User Interface Design and Implementation	Section re-done	1. Worst case scenarios are considered due to alternate scenarios for the UI diagrams 2. New UI pictures were replaced to represent current application look
12	Design of Tests	Section re-done	

13	History of Work	Added Report 1/2 comparisons, current work, future work	1. Added current updates for final demo for the application

C. Demo 2 Questions

KARTIK:

1. What would be the three biggest changes/advancements that you made from Demo1?

   The three features which have been changed the most have been photos, journals, and calendar/dailysummary. In general, we have strived to make our features more automated and sophisticated, and these three features best reflect our progress in doing so. In the first demo, some of the questions we had for the photos was how photos were going to be selected for the trip - could the system add photos in itself or would the user have to upload photos one by one. In order to make the system more automated we decided to fetch the photos taken during the duration of the trip and have the user decide which of those photos to upload. A major benefit is that it is much more efficient to upload a photo by now simply pressing on the picture rather than having to load the camera roll and select from there. In the Journals feature, we have added tags which autocomplete based on the names of pins on the map and the names of attractions the user has saved. When a tag appears in a journal, the system will fetch the photo url stored in the pin and attractions objects and gives a custom display for that journal of tags, photos, date, user photo, and journal title as seen in the demo. Lastly, for the calendar/dailysummary we have significantly improved the UI, added a weather display, and implemented the daily summary feature, which displays photos, journals, and attractions the user has stored the day that is currently selected on the calendar. Also, for these features it should be noted that working on the UI itself takes a significant amount of effort as well.

2. How do you make sure that the phone fetches (automatic upload) relevant pictures in between the dates mentioned?

   Our research found that when our camera's take a picture, they store a lot of interchange information in the EXIF data format as part of the image file, and we are able to parse and check for this set of data in each photo. Therefore, when our system loads the photos for a trip, it checks which photos have the relevant EXIF data (creation time, location) and only if it has this data, and the creation time parameter is between the duration of the trip, do the photos get loaded. Additionally, there's a photo identifier that's stored within the file which is a culmination of the deviceId and the timestamp of the photo taken that is generated. By storing this ID, we are able to verify whether a file already has been uploaded or not; allowing the user with more granular control of which photos stay on their device, and which are uploaded and available to every device where our application is installed and is logged in with the same account.

3. For the new attractions, are you given all the options or some selected top-recommended specifically to the user?: Later shown
    Currently the application shows the top 20 most highly rated businesses based on the user's current location.

4. The budget feature is user-driven and not automated to recommend attractions?
    The budget is user driven because the costs for attractions can be greatly varying. In this case, if you wished to input a purchase, it is best to do it manually to take into account this factor. The budget, however, is not completely user driven, as it gives the user a spending limit they should try to stay below, in attempts to budget well. This limit is determined by the overall budget and is split equally amongst the days of the trip.

5. There is no range option in the price and rating? You can make other recommendations as previously visited and favorites.
    Currently there is no range filtering, but this would definitely be possible to implement. The only problem with filtering price is that we wouldn't be able to filter based on specific numbers, since the API does not provide numbers that characterize how expensive activities are at certain businesses. What we could do, however, is filter based on qualitative price rankings ($-$$$$$) because the API does provide this information. As for favorites and previous visits these are also recommendations that are good ideas. Implementing these require us to add properties to a user account instead of a trip, and comparing them to the set of attractions returned to the api. It is definitely an interesting problem worth potentially including in the final demo.

6. For the location tags in the journal, does it prompt only the locations present in the map or all the locations?
    Concerning location tags, it will only prompt locations that were pins in the map or saved attractions. The data for the autofill is taken from already present data within the system. So, autofill will take the collection of map pins and saved attractions to present an autofill set. This is due to the consideration that user will be likely writing about locations that they have already documented/visited.

SANYAM:

1. Is your application fully integrated?
    Yes. The app is integrated as follows:
    1. The user can login and be taken to the trips page or the user’s login has been saved from before and he/she is taken directly to the trips page

2. From the trips page, a trip may be selected and from there we are taken to the main view of the trip which has a bottom navigation bar that allows one to switch between all of the features - photos, map, journals, etc.
3. There is cross-functionality amongst features which use information from other features to enhance themselves

2. Does your application provide a reminder about upcoming events from the trip?
   There is currently no such feature. However, since we have used Firebase in our project, we can fairly easily utilize the Firebase Console, which allows us to display a notification to iOS and Android devices. Within the console we would be able to schedule notifications at a specific time so we could check at a given time each day if there will be a trip starting within 24 hours and if so we can display a notification. Hence, we will strongly consider implementing such a feature for our final submission.

3. What is the most unique feature of your application?
   The most unique feature of our application is that a user is able to log and plan their trip in one location. Currently, there are very few applications existing that have all of the functionality our application provides. If we go by feature, we have a calendar (with daily summaries), photos, journals, map, budget, and attractions. Each section has been fully developed, and when integrated together, they provide a cohesive experience for the user. The great thing about Trip Diary is that it provides for cross-functionality between features, and automation for the user.

AMOD:

1. Any reminder functionality for trip day once a future trip is added into the diary?
   Not as of yet - but, the journal entry chart (top of the main journal page) is centered around the trip start and end dates. The user will be able to see within that chart if they have added a diary after their start date, considering it acts as a journal calendar as well. As mentioned before, we can provide notifications using the Firebase console if we did want to notify the user!

2. Photo upload only shows photos of trip day? What if a friend sent in photos later?
   The photos tab actually shows photos from the entire duration of the trip, not just a single day. Additionally, as we mentioned in the response to Kartik's second question, the image file contains a set of data that gets carried over even if a friend sends you the file later. Once you receive this image file, our system will not check for the "file modification" timestamp, but rather the "file creation" timestamp, which persists across devices, and will store the time the photo was taken on a user's device. Once this occurs, our system will pick it up, and through

the generated photo Id mentioned earlier, the system can identify whether or not this photo has already been uploaded to the trip.

3. Which location has priority in map direction, for multiple selection?

   When the map renders we first fetch the attractions, translate them into pins, and then render them on the map. The API for the map, conveniently provides a fitToElements() function which we call to focus the map so that all of the pins are included in the view. In this sense, we don't give priority to some pins over another by focusing on specific pins. When the user has selected a pin and tries to select another, the view of the first pin will fade away and the view of the newly selected pin will appear. In principle, we don't allow multiple selection as only one pin view can be displayed at a time, but if the user tries to select another pin then the view of the most recently selected pin will appear.

4. Calender summary would show weather for that day only? What if a previous day is selected?

   Since we are using a free API from OpenWeatherMap, it is only possible to show the current weather data. This means that the calendar only shows the weather that is happening currently(real-time) at the user's location. This is separate from selecting days and will always display this current real-time weather. If we decided to upgrade the API, we could display the weather data from the selected day(previous or future); however, this would require us to pay a subscription fee.

5. How does the journal entry get pictures? Default is a picture of a cup?

   The journal gets pictures due to 3 scenarios:

   1. **User's uploaded photos (corresponds to map pins)**

      A user has the option to include location tags from map pins. If a user has uploaded photos on the photos tab, we are able to grab the location of the picture. If the picture is within 5 miles of the location tag, then the user will get that picture as their journal photo. As for in the demo, the cup was a picture that the user uploaded, associated with 'Home' from a map pin. This allowed for the cup picture (which was taken at 'home' for the user) to appear on the journal card.

   2. **Google photo (corresponds to saved attractions)**

      A user has the option to include location tags from saved attractions. If a user has selected a saved attraction as their location tag, the cover picture from Google for that location (when searched in Google search) will be used for the cover picture of the journal card.

   3. **Default photo**

      If the user has not included any location tags corresponding to a

map pin or a saved attraction (i.e. a custom location tag), a default picture of the 'Pink Golden Gate Bridge' will appear.

1. Customer Statement of Requirements

1. Actor - General Traveller

Only a frequent traveller would understand that there are many things that compromise going on a trip, these include planning in advance, doing what was planned, and then reflecting on the journey. It is easy to spread travels on many platforms -- one can book a flight on an airline's website, plan onna trip planning platform such as TripAdvisor or a local application, and store photos in an album.

A. Concern 1: Daily Trip Summary

It has become more and more frustrating to understand expenses from mismanaged credit card statements, knowing what plans are made each day, and reflections upon each day (since it is sometimes difficult to remember to journal, which is essential when doing day-to-day reflections).

What would be ideal is a platform where one is able to see everything together. Everything includes what they did each day (including the transportation they took), all of their journals (which reflect on what they did), and what they spent on different excursions. This would allow them to be able to reflect on their trip without having to scour different platforms, and essentially places everything related to their vacations together

Trip Diary Solution: Trip Diary allows for an environment which aggregates all information related to a vacation. It provides a calendar in which people are able to view the dates of their trip. On that same page, they are able to view a daily summary, which includes locations they've visited (through a map visual), their budget (with interactive charts to organize information), pictures, and journal entries. To further organize this, if the 'general traveller' wants to see a further breakdown, they are able to interact with the calendar.

2. Actor - Journaller

Many people find journaling as an outlet for expression and reflection. These same people would find journaling especially useful whenever they travel. According to this travel blog which references the benefits of keeping a travel journal, a journaler essentially identifies with being able to

1. Gathering information for different needs and wants during the trip
2. Documenting thoughts provides a meditative therapy through the hassle of travelling
3. Provides a creative outlet to one's texts, drawings and non-text scribbles

4. An alternative when passing time during a dead period on a trip (ex. Sitting at the airport, waiting for a train, etc.)
5. It is a personal souvenir of one's trip, with several sentimental entries reflecting on their thoughts from time away
6. Provides for richer memories, and an opportunity to look back at how one experiences each day

A. Concern 1: Digital Platform

Although journalling is a great pursuit in a physical book, it's quite frustrating having to find a place for my diary, especially when there is limited space to fit things. Also, it is very possible for one to lose their journal book, which will absolutely demotivate them to continue journaling for the remainder of their travels. Additionally, sometimes a journaller will be so tired and exhausted on their trip that physically writing is the last thing they want to do. There should be a means to type their thoughts out on something, instead of always having to physically write.

What there should be is a digital platform that is accessible off of the phone or tablet. For the majority of people in the world, life is held up on the phone or digital device. It would be helpful if one could journal within an app that was related to their travels. Additionally, it would be awesome if there were several options -- doodling, typing, and adding other emoticons.

Trip Diary Solution: Trip Diary offers a journaling feature, where the 'journaller' is able to document his/her experiences on a digital platform. This feature is not solely a text sheet, but provides prompts of the user to answer ("What was your favorite memory from today?", etc.), allows for typing and doodling, photo, and sticker features. This provides the user with an array of options when documenting their thoughts.

B. Concern 2: Adding Pictures to Entries

Another concern is wanting to put pictures in my journal. It's great being able to print pictures and placing them in the journal, but that has to be done after the trip. Sometimes, it is easy to completely forget to go and print pictures.

It would be nice to see a place that has pictures directly from the journal. Essentially, this would only be feasible by digital means, where it takes an instant to take/place a picture and place it within the text.

Trip Diary Solution: Trip Diary provides an option to add photos to accompany text. Essentially, users will be able to either choose from their camera roll or physically take a picture while adding entries. This will allow them to connect their photos (which are additionally incorporated into the application and journal entries, providing for interaction between features.

C. Concern 3: Remembering to Journal

It is easy to forget to do non-essential tasks, especially journaling during a trip. It is frustrating having journal entries for only select days of the trip, and it would be desired to have a full collection of entries when reflecting back on the trip.

It would be nice to see a reminder feature for my journals. This would remind one to journal at a (possibly preselected) time directly to their phone.

Trip Diary Solution: Trip Diary allows users to receive reminders to journal if they have not completed it already for that date. This would be directly on their home screen, and if they clicked on the notification, it would take them to the journal entry page on their Trip Diary application.

3. Actor - Photographer

A photographer would like to store all of the wonderful memories that he captures during his trips. Photos allow him to not just see a moment of his trip but it also serves as mental cues which instigate entire memories that he has associated with the photo. The motivation of trips would be to experience pieces of culture, history, and life which inspire, amaze, and transcend the life that one is used to. When looking back at trips and the experiences, one would want to be able to feel those same wonders so that they can truly appreciate traveling and learning about the world.

A. Concern 1: Photo Organization

There is an issue of how to best organize photos so that the user can get the most out of viewing them. There exist social media sites such as Facebook which allow one to store photos in the order that they are uploaded in an album. These photos typically have metadata such as date, location, and descriptions. However, when one wants to view an entire trip, viewing an album may not be the optimal way of re-experiencing a trip. As a whole, it is hard to tell how the pictures fit in together and one may struggle to understand the meaning of some pictures whose purpose may be hard to discern or are not well described.

Trip Diary Solution: We believe that users have an unmet desire to be able to organize their trips into categories and be able to write descriptions of individual photos and groups of photos. One of the major purposes of the trip diary is to encourage users to better document and manage their trips, and by doing so to their photos they will have a better and fuller understanding of a trip.

4. Actor - Mapper

A map is a visual tool which can help trace the way one traveled. Looking at a map gives me a better idea of what regions he visited, differences between those regions, and also specific points within a region. It is interesting to be able to see how places are organized within a particular location, it provides a story about what life is like in a city. For instance, if one were to visit a park, a broadway show, and a skyscraper, he would like to know how these parts are located relative to each other so that he can get a better idea of the city's structure.

A. Concern 1: Trip Ordering

An important aspect which photos may fail to convey is a specific ordering of how a day was spent as photos are taken sporadically or may not be taken at all at particular locations. Hence, there should be a way for users to understand how they were able to get from one place to another.

Trip Diary Solution: So, we believe there should be a mapping feature which uses geolocation to trace location. Doing so, one can curate a summary of places visited, restaurants that were stopped at, and roads traveled. This feature aggregates a lot of information for users and is convenient in not having to be bogged down in writing the same detailed information. Additionally, one can fetch information about the places visited that they may have not known, enlightening them about every aspect of their trip. This feature is meant to provide as much convenience as possible so that the trip diary acts as an enhancement and not as a distraction.

B. Concern 2: Privacy

However, with this convenience comes a tradeoff in privacy. Many users may be uncomfortable with having their location being tracked at all times. We understand their concerns and especially would not want data about the trips they have taken to be shared with third-party sources. As this is a diary, a high level of privacy is expected and should be implemented. The world is a complicated place and one may choose to travel to an area which may be experiencing a conflict with other regions. One might feel distressed if that information was shared without my permission and ended up offending someone else, and would much rather it be kept private.

Trip Diary Solution: We believe that the application should have the geolocation tracking in the background be optional. This will not severely hinder the mapping feature as one can employ additional ways of getting locations. For instance, one can use APIs to extract the location of a picture taken and based on that form some kind of ordering. We hope that the application will seek to balance privacy and a detailed mapping as best as possible. It should be obvious for the user to see if location is currently being tracked and to turn it off.

5. Actor - Budgeter

As a single traveler, one is solely responsible for how they spend my money. A goal should be: when traveling, try to maximize experiences, but limit how much money is spent. But sometimes, it is difficult to keep track of expenses, since there are many expenses to keep track of. There are multiple forms of payment that we use (different currencies, card, online payments) as well as many different activities and expenses (transportation, food, traveling, souvenirs). Trying to keep track of receipts, bank statements, and payments is a frustrating element that takes away from the experience of traveling

A. Concern 1: Budgeting

What we would like to see from an app is an easy way to log and organize expenses during a trip, so that we can forget about keeping track of payments, and enjoy the trip experience. We want a feature where we can categorize our payment plans, and send notifications when we are nearing my spending limit.

Trip Diary Solution: The trip diary has a budget feature where a user will be sent notifications to log their expenses and warn them where they are nearing their budget. A user will be able to label different categories that they define or from a preset list from the application, and the user will be able to assign smaller budgets to each category. For each category the user will be able to see a running list of their expenses, and a total expenses list will also be available to the user. A visual graph will also be displayed, so that the user has a visual understanding of how well they are following their financial goals. Once a budget has been set, it can be edited as well.

6. Actor - Parent with Children (Finding Attractions)

As a parent, one would need to be able to find attractions that will be fun for the whole family, and document the places they visited, so that they can remember the experiences they had together. Trying to parse through several travel guides to see what exciting activities they can do as a family is a frustrating experience, since each guide has different information.

A. Concern 1: Finding and Saving Different Attractions

One should be able to view different attractions, both excursions and locations that they have visited so that they can share this data with other family members and friends. One would want to be able to not only view these locations in a list format, but also in a visually appealing way so they can understand how much distance they have traveled to visit these locations. One would also want to be able to view all the destinations they have visited so they can pick their favorite ones and see them in an organized format. They also want to be able to see their attractions so that it is apparent they are making the most of our trip and visiting as many important locations as possible. They also want to be able to view attractions by category, whether that be food, excursions, etc. This would help them note all the different types of attractions they have visited while on their trip.

Trip Diary Solution: The trip diary has a map feature where a user can view all the attractions and locations they have visited within their trip. When a user uploads a photo or video, the diary can suggest nearby locations that the destination might be or the user can manually input the attraction. Then, the user can have that location pinned on the map so that they can view how much they have traveled on their trip and how many spots they have visited. This gives the user an idea of what they have done on their day-to-day in regards to visited attractions. These pins have tags, whether that be food, excursions, hidden gems, etc. This provides organization for all the attractions visited so that the user can view all the food spots they hit, all the tourist sites they've visited, all the hidden gems they have discovered, etc.

2. Glossary of Terms

Term	Definition
Trip Summary	Overview of a completed trip or a trip in progress. One may view the trip budget, the photos of the trip, the journal entries, the timeline maps, etc.
Journal Entries	A feature that allows users to describe their trip experience with a creative and easy to use and navigate questionnaire. This feature saves the responses for future reference.
Trip	A well-defined data model used in our application to store and analyse information about the user's trips. This feature stores data such as location, photos, journal entries, attractions visited, hotel stays, flights, etc.
Daily Summary	A feature of the application that allows the user to view a concise analysis of a specified day of the user's current trip. The daily summary breaks down the users spending, location, photos taken and more under one page. In addition, the daily summary will display other relevant information to the user such as the suggested funds left to spend during the current day on the trip. The daily summary is designed to be a convenient, easy to use and aesthetically pleasing overview of a single day.
Attractions	The places that the user visits during their trip that does not include transportation or shelter. An example of an attraction would be a restaurant or a concert. Attractions could also be referring to the predicted places that the user could visit in the future. - Note: Events fall under the category of attractions.
Calendar	A feature that allows users to view major components of a trip, such as hotel check-ins, flights and planned attractions on an easy-to-follow day-to-day calendar grid.
Trip Budget	A feature that enables the user to enter the amount of money they would like to spend on the trip in total as well as each component of the trip. The Trip Budget feature will store this data and will eventually analyse this information to build a financial plan for the user.
Toolbar	A part of the application that allows the user to quickly access a different feature of the application. This toolbar is an access bar that will lie at the

	bottom of the application on most pages.
Photos	Collection of photos taken by the user during the current trip.
Geotag	Location data associated with an image taken by a camera.
Map	View within the application which allows the user to view their activity throughout their trip
Activity	User’s location history throughout the trip duration.
Pin	Point on map where the user visited an attraction or took a photo.
Transportation	Aggregated travel data for the current trip; includes varying modes of travel such as flights, ubers, trains, etc.
Save	Saves an attraction that is appealing to the user, and places it in a list, so that the user can explore attractions later for future consideration.
Commit	Adds an attraction to an itinerary and will send push notifications to remind the user to go to these events.
Budget	Allocating money to different categories (either defined by the user, or any number of the preset categories in the application) to set a spending cap for the user.

3. System Requirements

A. Enumerated Functional Requirements

Identifier	User Story	Size
REQ - 1	As a user I will be allowed to create an account and login so I can use the application, so I can get access to my trip journal.	3
REQ - 2	As a user, I will be able to view all of the trips I created so I choose to view, edit, or create new trips accordingly.	3
REQ - 3	As a user, I will be able to click on any date of my trip from the calendar	2
REQ - 4	As a user, I will be able to view the current weather in my nearby area through the Daily Summary	3
REQ - 5	As a user, I will be able to view all my photos, journal entries, and events on a single day in the daily summary	3
REQ - 6	As a user, I will be able to view suggestions of specific attractions or events for the current or future day	3
REQ - 7	As a user, I will be able to choose to add photos, videos, journal entries, and events to the current day	4
REQ - 8	As a user, I will receive a notification if an event I have planned is happening soon	3
REQ - 9	As a user, when creating a journal entry, I will be able to add pictures, drawings, and videos to my writing	3
REQ - 10	As a user, when creating a journal entry, I will be presented with multiple prompts to write on	2
REQ - 11	As a user, I will be able to add photos to my trip so I can organize them into being part of my trip diary.	3
REQ - 12	As a user, I will be able to view the photos I took for the trip so I can get a closer look at the pictures I took.	2
REQ - 13	As a user I will be able to have my location tracked so I can view where exactly I went throughout the day during my trip.	2
REQ - 14	As a user, I will be able to sort the photos I took by the location so I can easily organize and view photos how I please.	2

REQ - 15	As a user, I will be able to view my location data on a map for the trip so I can look back at exactly where I went every day of the trip without writing it down.	5
REQ - 16	As a user, I will be able to view pins on a map where I took my photos and visited so photos can be viewed by locations on the map adding another dimension to sharing and organizing photos/ marking down attraction.	2
REQ - 17	As a user, I will be able to set pins on a map where I visited attractions so exact details of the trip can be stored.	2
REQ - 18	As a user, I will be able to sort my location data by date so looking into details of past trips is easy (can be exceptionally helpful for business expenses).	1
REQ - 19	As a user, I will be allowed to set a predetermined amount of money, so that I can establish a budget.	1
REQ - 20	As a user I will be able to input money spent on purchases, so that I can keep track of how much money I am spending relative to my budget.	4
REQ - 21	As a user, I will receive notifications to input money spent on a trip, so I can be reminded about my budget.	1
REQ - 22	As a user I will be allowed to categorize purchases, so that I can better organize my budget.	1
REQ - 23	As a user, I will be allowed to assign a budget for each category, so I can allocate funds to different activities.	3
REQ - 24	As a user, I will be able to view a running list of expenses for each category and a total expense list, so I can see what specifically I spent my money on.	2
REQ - 25	As a user I will be sent notifications when I'm nearing my spending limit, to warn me about my spending limitations.	2
REQ - 26	As a user, when I click on today's date or future dates on the main calendar, I will see top attractions in the area, so that I can pick activities to do.	1
REQ - 27	As a user I will be able to filter activities by different tags, so that I can browse results more tailored to me.	1
REQ - 28	As a user I will be allowed to choose attractions by category, so that it will be easier to find places that I am interested in.	1
REQ - 29	As a user, I will be allowed to save future activities that I am interested in, so I can decide later whether I want to commit to the activities or not.	2
REQ - 30	As a user I will be allowed to delete saved activities that I am no longer interested in, so that my saved attractions view is not cluttered.	1

REQ - 32	As a user I will be allowed to delete budgets, so that my budget view is not cluttered.	1
REQ - 33	As a user, I will be able to delete any journal entry	1

B. Enumerated Non-Functional Requirements

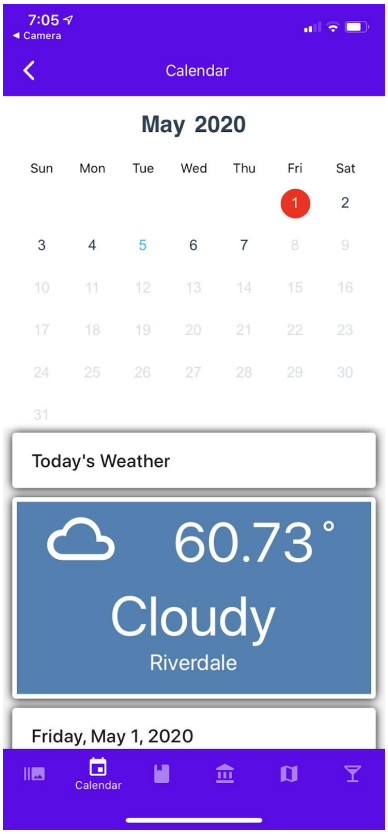
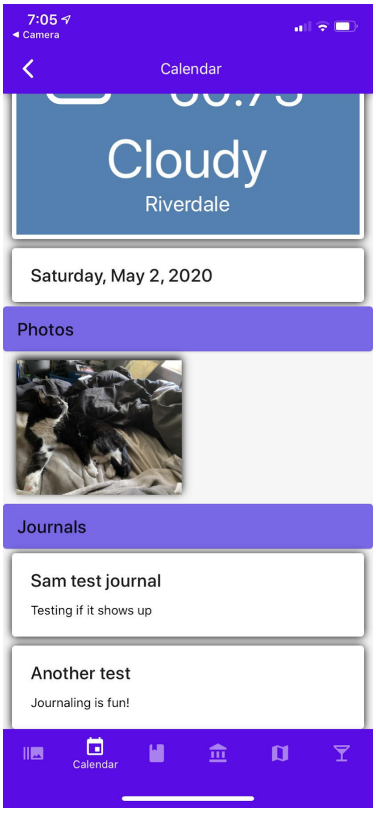
Identifier	User Story	Size
REQ - 34	As a user I will be able to access the application on my android device so that I can use the device while I'm traveling.	4
REQ - 35	As a user, I will wait no more than two seconds for the daily summary to render, no matter how many users may be active	3
REQ - 36	The mobile application will be compatible with android phones	2
REQ - 37	The application will have an 80% reliability rating (the percent chance it doesn't crash) in a month	2
REQ - 38	As a user, I will have the option to choose whether my phone is currently tracking my location so I can retain control over my privacy.	2
REQ - 39	As a user, I will be provided documentation that thoroughly describes how to use the photos, mapping, and transportation features.	1
REQ - 40	As a user, I will be assured that data being compiled by the application is accurate.	2
REQ - 41	As a user, I will be able to upload hundreds of photos without running out of memory.	3
REQ - 42	As a user, I will be assured that my password to login is encrypted and my data is secure from other users as well as secure in the cloud.	2
REQ - 43	As a user, I will be able to easily navigate through all of my photos the first time I encounter the feature.	2
REQ - 44	As a user, I will be able to easily navigate through the photos feature because of its memorableness.	2

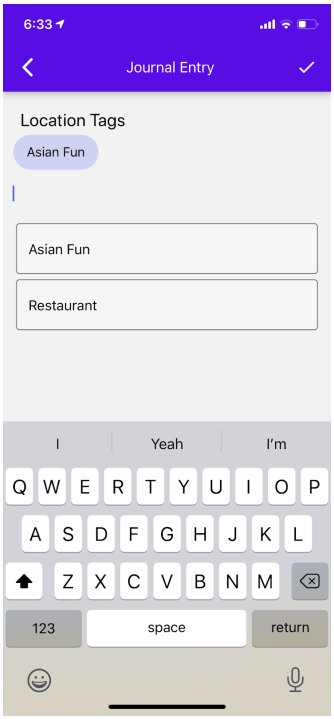
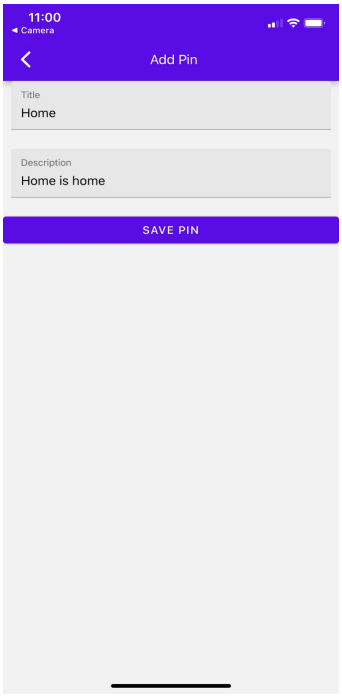
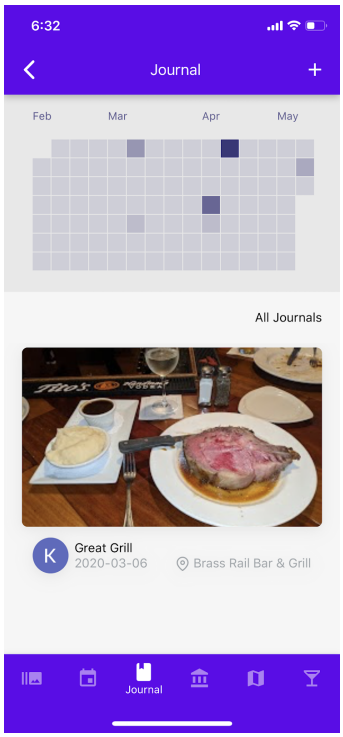
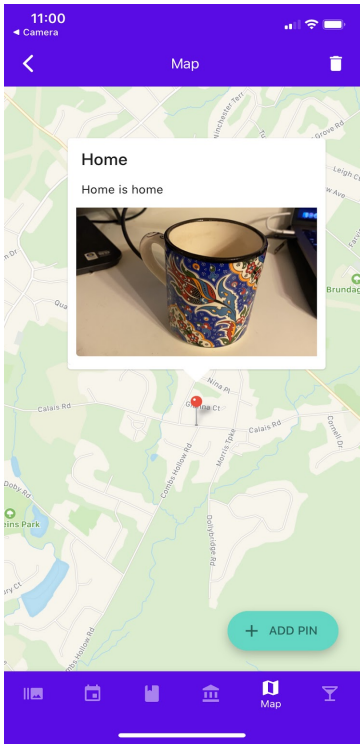
D. User Interface Requirements

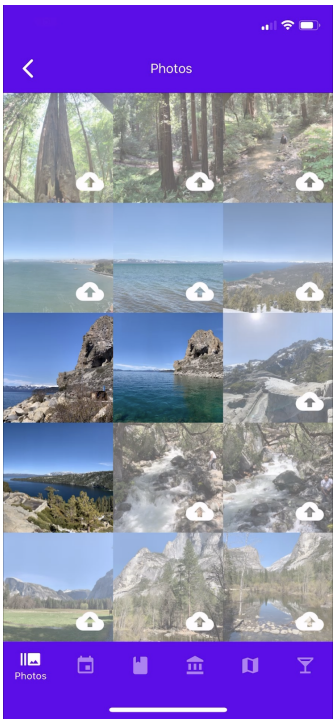
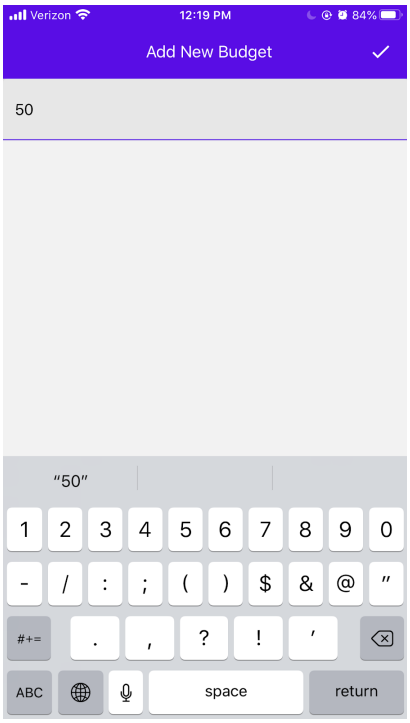
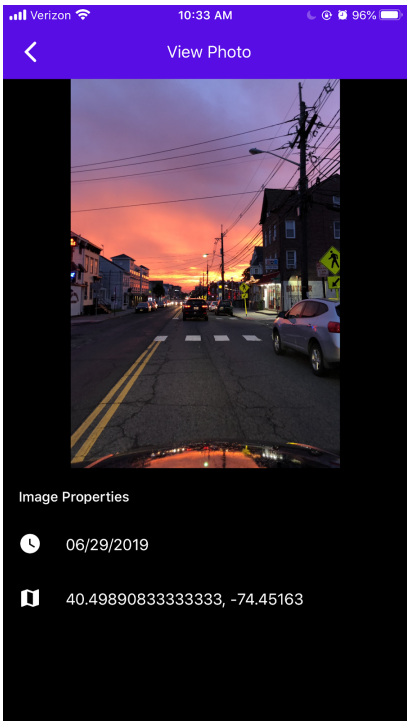
Identifier	User Story	Size
REQ - 47	As a user, I will have a log-in page to view my personal account	3
REQ - 48	As a user, I will have a page where I can select separate trips	2
REQ - 49	As a user, I will be able to view all of the dates in my trip from the calendar	2
REQ - 50	As a user, I will be able to click on a certain date from my trip, which will display a summary of entries from that day (map, photos, journal, budget)	3
REQ - 51	As a user, I will be able to click on specific entries (map, photos, journal, budget) from the daily summary page to direct me to a full page of only that entry	4
REQ - 52	As a user, I will be able to go back to key locations within the app via a toolbar located at the bottom of the app	3
REQ - 53	As a user, I will be able to view my existing journal entries from a specific date	3
REQ - 54	As a user, I will be able to view the status of the current trip I am on which includes a map with all the pins I have set and the list of pins that can be clicked on to view/add relevant photos	3
REQ - 55	As a user, I will be able to view all the pictures of a trip organized by either the location they were taken or the day of the trip they were taken	2
REQ - 56	As a user, I will be able to view all information relevant to my budget, including, my pre-set budget, how much do I have remaining, and budgets that I allocated money to.	2
REQ - 57	As a user, I will be able to view a graph, depicting all of my budget information, so that I will be able to visualize my expenses.	4
REQ - 58	As a user, I will be able to click an edit button that will allow me to edit my budget information.	1
REQ - 59	As a user I will be able to click on future dates and see top attractions in an area separated by category.	1
REQ - 60	As a user, I will be presented with a list of several thumbnails, each representing a different attraction, so that the user interface will not be cluttered.	3

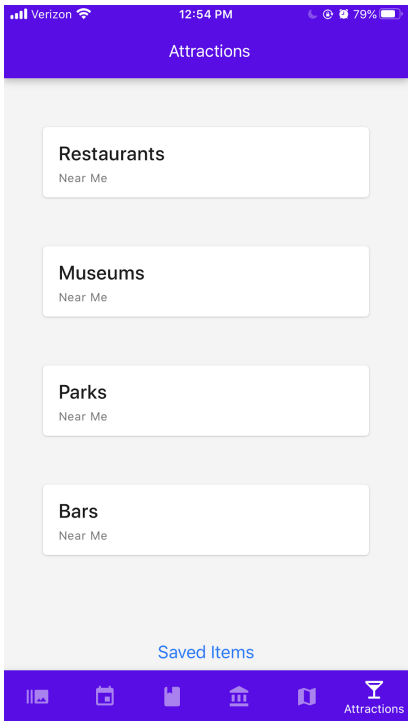
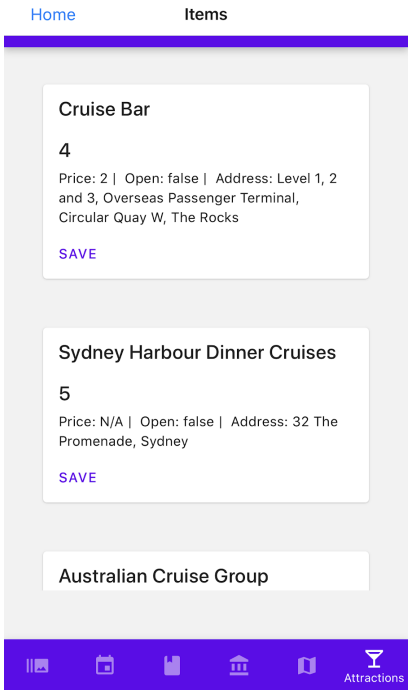
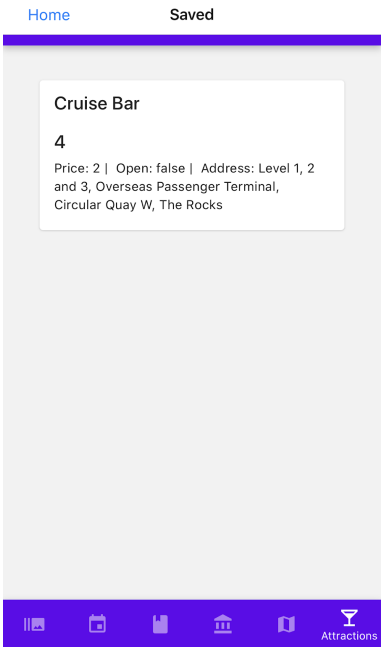
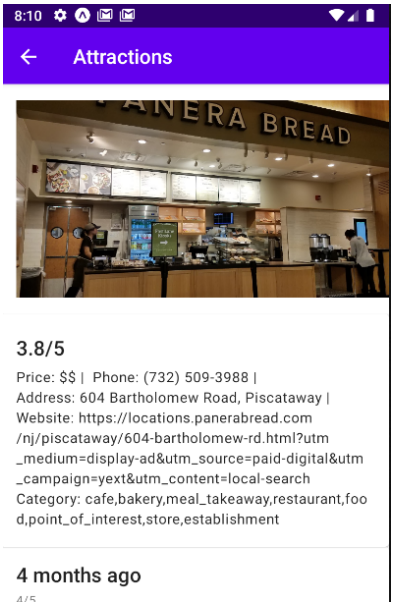
REQ - 61	As a user, I will be able to click on a button to save attractions for future consideration	2
REQ - 62	As a user, I will be able to toggle my list of expenses, so the interface will not be cluttered	1

(Cont'd) User Interface Pictures

Calendar	Daily Summary
	

Journal	Map
 6:33 Journal Entry Location Tags Asian Fun Asian Fun Restaurant	 11:00 Camera Add Pin Title Home Description Home is home SAVE PIN
 6:32 Journal Feb Mar Apr May All Journals Great Grill 2020-03-06 Brass Rail Bar & Grill Journal	 11:00 Camera Map Home Home is home ADD PIN Map

Photos	Budget
	
	

Suggestions (Pt.1)	Suggestions (Pt. 2)
 	 

4. Functional Requirement Specification

A. Stakeholders

1. Internal Stakeholders

a. **Developers**: This group of people works on developing the application in order to enhance traveling experiences
   i. Front-End
   ii. Back-End
   iii. Full-Stack

2. External Stakeholders

a. Travellers: These groups of people are interested in documenting and organizing a trip that they are currently on. Examples of these groups include (but are not limited to):

i. **Group travellers:** Having this app will allow the members of a group to coordinate their decisions about their upcoming trip in one, easy-to-use web-app. Then, during the trip, recording trip information is beneficial for the future reference of any member in a group to recall what you and your friends did in a past trip. We believe group travellers will make best use of our Attraction and Calendar features. Group travellers should take advantage of UseCase-1, UseCase-2, UseCase-6 and UseCase-7.

ii. **Institutional Travellers:** Whether on a business trip or a cool school field trip/vacation, this app is useful for the travellers themselves to view each and every trip detail so the traveller(s) can avoid confusion and miscommunication and, as a result, can remain on-schedule and on-task for the entire trip. As for the administrators/facilitators of the trip, the services that our app provides will be a great way to consolidate general records and financial records for the institution. We believe institutional travellers will make best use of our Map and Calendar features. Institutional travellers should take advantage of UseCase-1, UseCase-2, UseCase-4 and UseCase-7.

iii. **Individual Travellers:** Documenting trips can boost popularity and fan-base for popular users and celebrities. But for the more common user, this app can be used to optimize the current trip (financially, quality of trip, etc.). In any case, it is also useful for anybody travelling solo to be able to reference older trips for any number of reasons. Individual travellers will make best use of our Photo, budgeting and journaling features. Individual travellers should take advantage of UseCase-1, UseCase2, UserCase-3, UseCase5 and UseCase-8.

iv. **Couple or Family Travellers:** It is always a good feeling when memories and experiences are remembered from a family vacation. This app makes it easier to recall the experiences you had with your lover and children with our journal and photos feature. Romantic travellers will make best use of our Photo, Attraction and budget features. Romantic travellers

should take advantage of UseCase-1, UseCase-2, UseCase-3, UseCase-5 and UseCase6.

b. **Visitors:** These groups of people are especially interested in looking through their own or other's previous trips for reference and inspiration for future plans. Examples of these groups include (but are not limited to):
   i. **Fans:** Fans would be interested in planning future trips based on their favorite celebrity's past plans. For instance, say a fan's favorite artist took a vacation in Paris, stood on top of the Eiffel Tower and took a picture of themself with a unique pose. That person may be interested in going to Paris and going to the Eiffel Tower and taking a picture of themselves with the same pose as their favorite celebrity. Our app makes it easy to do everything from budget this trip to uploading their own Eiffel Tower photo to our app and social media. This stakeholder group is a future goal that does not yet make use of any current use cases, but will take advantage of future use cases.
   ii. **Family members:** Anyone that is part of a family might like to rediscover the trips that another family member took for any number of reasons. An example of this would be of a son looking to remember his lost father. Since the best memories are made on vacation, the son would find the most moving and sentimental memories while scrolling through his father's vacations in the diary and the photos feature on the father's profile. Family members and close friends will make best use of our Photo and Journaling features. Family members and close friends should take advantage of UseCase-3 and UseCase-8.

B. Actors and Goals

Actors	Type	Role	Goals
Users	Initiating, Participating	The user creates, edits, or views features of the application which initiate requests to the system. The user participates in use cases where the system sends him or her a notification.	1. To log finances of attractions, food, excursions, etc. into the trip diary 2. To determine a budget for trip and view/update this budget 3. To locate attractions, food, excursions, etc. through search or suggestions 4. To save/add or delete locations of attractions, food, excursions, etc.
System	Initiating, Participating	The system initiates notifications to be sent to the users to ask about locations or remind them about documenting their day.	1. To process requests from the client and send back responses. 2. To send notifications to the user to ask for participation or access to things like photos or locations

		The system participates in all kinds of requests sent to the service dealing with the application	
Database	Participating	The system receives requests from the service to retrieve, update, create, or destroy data.	1. Store user data 2. Analyze user data to provide interesting results such as mode, averages, ranges, etc.
Geolocation API	Participating	The system initiates a request to the geolocation API to retrieve data about the user's location	1. To determine the locations of nearby attractions and excursions so that the user can be suggested possible locations they have visited when they log in their diary 2. To determine the location the user is currently at so that the user can use that information 3. To determine the exact location of any inputted GPS coordinates or names of locations so the user can view locations on a map 4. To save locations, their GPS coordinates and the name of that location on the map, so that users can save information on destinations 5. To use metadata from photographs and videos and geotag those GPS coordinates on a map so that the user can document locations of memories and trips
Bank	Participating	The bank provides details about the user's bank account	1. To provide information on all spending so that user can select information to log into budgeting component of trip diary 2. To provide information on the total financial capacity of the user and by doing so, alerting the user if they are going to exceed the amount in their bank account

C. Use Cases

I. Casual Description

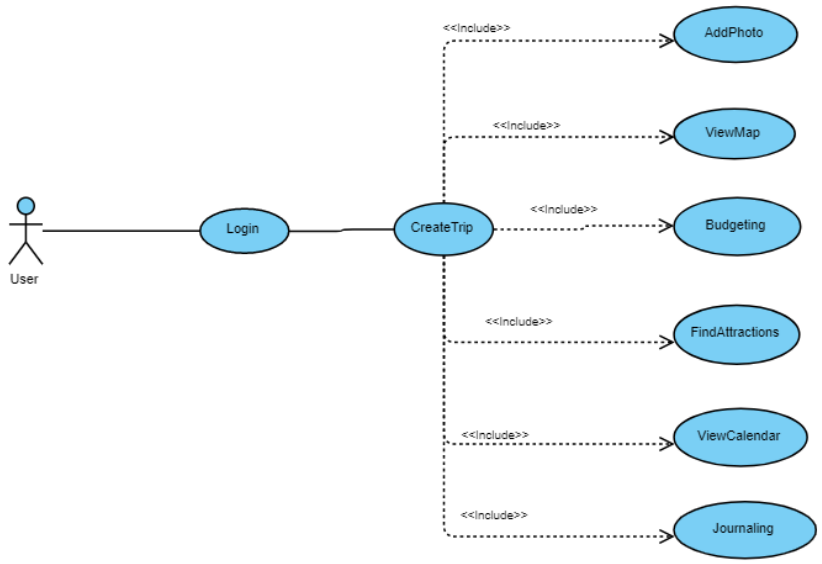
Initially we had 24 use cases to satisfy our functional requirements. Each use case had aggregated some of the functional requirements, but it was determined that each one was too specific and needed to be broadened. For example, for the mapping feature we had use cases which targeted a very specific behavior such as adding a pin to the map, tracking the user, and creating paths between pins. Now we have a single use case which encompasses all of those components of the mapping feature. We did something similar for each major feature by aggregating the basic use cases into a more broad single use case.

Furthermore, it was determined that some of our features needed to be more sophisticated. The amount of work between the system and the user was unbalanced towards the user. Therefore, in our updated use cases we have added features that increase automation such as having the system suggest attractions, create a daily summary, show budget statistics, and prompting the user with useful information from their journal. Another way we have taken the burden off of the user is by making the application more cross-functional. Each feature can utilize information from other features which can allow the system to perform sophisticated tasks such as auto-generating pins on the map based on the attractions saved rather than having the user add them all in. We have reflected these new features in our updated use cases as well.

Use Case	Casual Description
UC-1: Login	User can login to a previously made account and a new visitor can create a new account with a username and password and login
UC-2: CreateTrip	User can create new trip with starting date, ending date, and trip title and view daily
UC-3: AddPhoto	User can add a photo to a specific day/attraction for their trip and can view photos sorted in different ways (by date taken, by album, etc)
UC-4: ViewMap	User can view the attractions that were saved for each day as pins on a map. A route will be displayed between pins in the order they were added to the map. The map will be centered around the users current location which is found using the Geolocation API.
UC-5: AddBudget	User can create a budget for the trip they are going on and update their expenses during the trip to see how much money

	they have left to spend. They can also see budget statistics through graphical and other visual forms to see where and when they spent their money.
UC-6: FindAttractions	User can either search up specific attractions or can be suggested nearby attractions based on their current location, where this search can be filtered by category, price, and date and save these attractions to consider during their trip.
UC-7: ViewCalendar/DailySummary	Users will be able to set the dates of their trip and see them in a calendar as well as view their entire daily summary below their calendar within the app. If the user clicks on a certain date they will be able to view a summary of what they recorded for the trip that day.
UC-8: AddJournal	Users will be able to add, edit and view journal entries with suggested prompts. They can attach photos, drawings, or videos within entries.

II. Use Case Diagram



Key:
Straight Line = Initiates
Dotted Line = Includes

III. Traceability Matrix

Requirement #	PW	UC-1	UC-2	UC-3	UC-4	UC-5	UC-6	UC-7	UC-8
1	3	X							
2	3		X						
3	2							X	
4	3							X	
5	3							X	
6	3						X	X	
7	4							X	
8	3						X	X	
9	3								X
10	2								X
11	3			X					
12	2			X					
13	2				X				
14	2			X					
15	5				X				
16	2			X	X				
17	2				X		X		
18	1				X				
19	1					X			
20	4					X			
21	1					X			
22	1					X			
23	3					X			
24	2					X			
25	2					X			
26	1						X	X	
27	1						X		
28	1						X		
29	2						X		
30	2						X		
31	1						X		
32	1					X			
33	1								X
47	3	X							

Requirement #	PW	UC-1	UC-2	UC-3	UC-4	UC-5	UC-6	UC-7	UC-8
48	2		X						
49	2							X	
50	3							X	
51	4							X	
52	3			X	X	X	X	X	X
53	3								X
54	3			X	X				
55	2			X					
56	2					X			
57	4					X			
58	1					X			
59	1						X	X	
60	3						X		
61	2						X		
62	3						X		
63	1					X			
Max PW:		3	3	3	5	4	3	4	3
Total PW:		6	5	17	18	26	28	32	12

IV. Fully Dressed Description

UC-7: Calendar
Related Requirements: REQ-3, REQ-4, REQ-5, REQ-6, REQ-7, REQ-8, REQ-26, REQ-49, REQ-50, REQ-51, REQ-52, REQ-59 **Initiating Actor:** User **Actor's Goal:** View journal entries, weather, photos, and suggestions taken on a specific day of their trip **Participating actors:** System, Database **Preconditions:** - The user can log into their account - The user can create a trip and access the trip from the My Trips page - There is a database that can store the users information they decide to upload **Postconditions:** - The user can view the dates that they have set for the trip in the Calendar - The user can view all information about a specific day (current or previous) of their trip - The user can view their saved attractions when clicking on a future day
Flow of Events for Main Success Scenario: → 1. User clicks on the calendar tab when in a specific trip, requesting to view the calendar of the dates they have set as well as information about different days of their trip ← 2. The system processes the request and makes a database connection to retrieve the stored data ← 3. The database sends the requested information ← 4. The system generates a page displaying from the start date to the end date of their trip in the Calendar → 5. The user clicks on a previous or current day to view the Daily Summary ← 6. The system processes the request and retrieves the information from the database ← 7. The system generates a Daily Summary page underneath the Calendar **Flow of Events for Extensions(Alternate Scenarios):** **(1) Clicking to view specific aspect of day** → 1. The user clicks the "view" button on any of the aspects (photos, suggestions, journal entries, etc.) within the Daily Summary ← 2. The system directs the user to a separate page where only the specific item is shown **(2) Clicking on a future day** → 1. The user clicks on a future date of their trip in the Calendar ← 2. The system processes this request and fetches information from the database ← 3. The system generates a page of the user's saved attractions on the chosen day beneath the Calendar

UC-3: Add Photo
Related Requirements: REQ-11, REQ-12, REQ-14, REQ-16 **Initiating Actor:** User **Actor's Goal**: To add a photo from the camera roll to a specific day on a trip. **Participating Actors**: System **Preconditions**: - The System has access to user's camera roll - There exists a database in which the system can store the photo/path to photo **Postconditions:** - Successfully stored photo in storage bucket - Successfully stored photo reference in database with metadata - User can access the photo in the diary that was selected
Flow of Events for Main Success Scenario: → 1. User presses on the photos tab on the bottom of the screen ← 2. System displays all photos from a user's camera roll that are within trip dates in a grid format ← 3. System highlights which photos are already uploaded and which are local to the device → 4. User can click on an un uploaded photo to process it and upload it to trip diary ← 5. The system receives the request. It will make a database connection and add a new entry containing the path of the photo. ← 6. The system updates the album so that the user may see their new photo marked as uploaded → 7. The user can click on a photo to view it's details such as date, size, etc. **Flow of Events for Extensions (Alternate Scenarios):** ← 1. The system does not have permission to access the user's camera roll so it sends a notification requesting such access.

UC-4: View Map
Related Requirements: REQ-13, REQ-15, REQ-16, REQ-17, REQ-52, REQ-54, REQ-55 **Initiating Actor**: User **Actor's Goal:** To view a route of pins that the user has saved on the map. **Participating Actors:** System, Database, Geolocation API **Preconditions:** - Location data is safely stored within a database - The location of pins placed by user are stored within database **Postconditions:** - Path of where exactly the user went that day is drawn out with a line on the map - Pins the user placed are displayed on the map and can be clicked on for more info
Flow of Events for Main Success Scenario:

→ 1. User presses on the map tab on the bottom of the screen or selects a new day from within the map view
← 2. The system gets the user's current location from the Geolocation API and centers on those coordinates
← 3. The system queries the database for all of the pins of the locations that the user has saved and renders them onto the map
← 4. The system queries the database for all the photos taken by the user on that day
← 5. For each pin, the system searches through the photos and calculates whether any of them are within a 5 mile radius of the pin and if so, it adds it to the pin's detailed view
← 6. The system creates a route between the pins in the order of the time they were visited
→ 7. The user presses on a pin
← 8. The system displays a popup above the pin of the pin's title, description, and photos near it

Flow of Events for Extensions (Alternate Scenarios):
← 1. The system does not have permission to access the users location at all times so it sends a notification requesting such access
→ 2. The user uses a VPN to spoof their location and this results in a discontinuous map that looks glitchy when drawn out

UC-8: Journaling

Related Requirements: REQ-9, REQ-10, REQ-33, REQ-52, REQ-53

Initiating Actor: User
Actor's Goal: Add/edit a journal entry to a specific date that can accurately reflect their experience
Participating Actors: System, Database

Preconditions:
- The system has access to the database
- The system has access to the user's photo album
- The system has access to map pins

Postconditions:
- The user is presented with a writing prompt
- The user can add photos, drawings, and videos
- The journal entry will be saved to that specific day

Flow of Events for Main Success Scenario:
(1) Add Journal Entry
→ 1. The user requests to add a journal entry
← 2. The system processes the requests and generates a new page to add a journal entry
← 3. The system presents the user with a randomly generated prompt
→ 4. The user can click to do the following to their journal entry:
    a. Add photos
    b. Add videos
    c. Add drawings
→ 5. The user clicks to "save" their current journal entry

← 6. The system creates a database connection and saves the journal entry to the current day

(2) Edit Journal Entry
→ 1. The user requests to edit an existing journal entry
← 2. The system processes the request and directs the user to that journal entry's page
→ 3. The user edits the journal by either
   a. Changing the text
   b. Adding/deleting photos or videos
   c. Adding/deleting drawings
→ 4. The user clicks to "save" their journal entry
← 5. The system creates database connection and updates the information

(3) Delete Journal Entry
→ 1. The user requests to delete an existing journal entry
← 2. The system creates database connection and deletes journal
← 3. The system presents the user with the main journal page, where the entry is no longer available

Flow of Events for Extensions (Alternate Scenarios):
→ 1. The user requests to add a journal entry missing either a title, body, or both
← 2. The system notifies the user of the error, indicating they are unable to save without completed information
← 3. The system continues presenting the editing page to the user, to complete the information

UC-5: Budgeting

Related Requirements: REQ-22, REQ-25, REQ-47, REQ-48, REQ-60, REQ-23, REQ-25, REQ-61

Initiating Actor: User
Actor's Goal: To view the current budget remaining and to change or delete the budget. To add finances and to delete finances that correspond to categories to purchases. To be able to view a summary of spending by viewing statistics of the financial history.
Participating Actors: System

Preconditions:
- The user is on the "Budget" tab

Postconditions:
- The user can view the current budget they have
- The user can add a new finance or delete an old one
- The user can view budget statistics
- The user can change/make/delete a budget

Flow of Events for Main Success Scenario:
→ 1. The user clicks on "add budget"
→ 2. The user adds a new budget
← 3. The system saves this new budget in the database

← 4. The system processes the requests by generating the user's budget in the database → 5. The user views the budget → 6. The user adds new finances ← 7. The system processes this by saving these new finances to the database → 8. The user deletes finances ← 9. The system processes this by deleting these finances from the database ← 10. The system retrieves the list of finances from the user in the database → 11. The user views the list of purchases made → 12. The user clicks to view budget statistics ← 13. The system performs the mathematical equations to generate statistics ← 14. The system displays the statistics by retrieving the statistical values computed by the database
Flow of Events for Alternative Scenario: → 1. The user inputs an invalid purchase or budget (with non real positive numbers) ← 2. The system notifies the user of an error

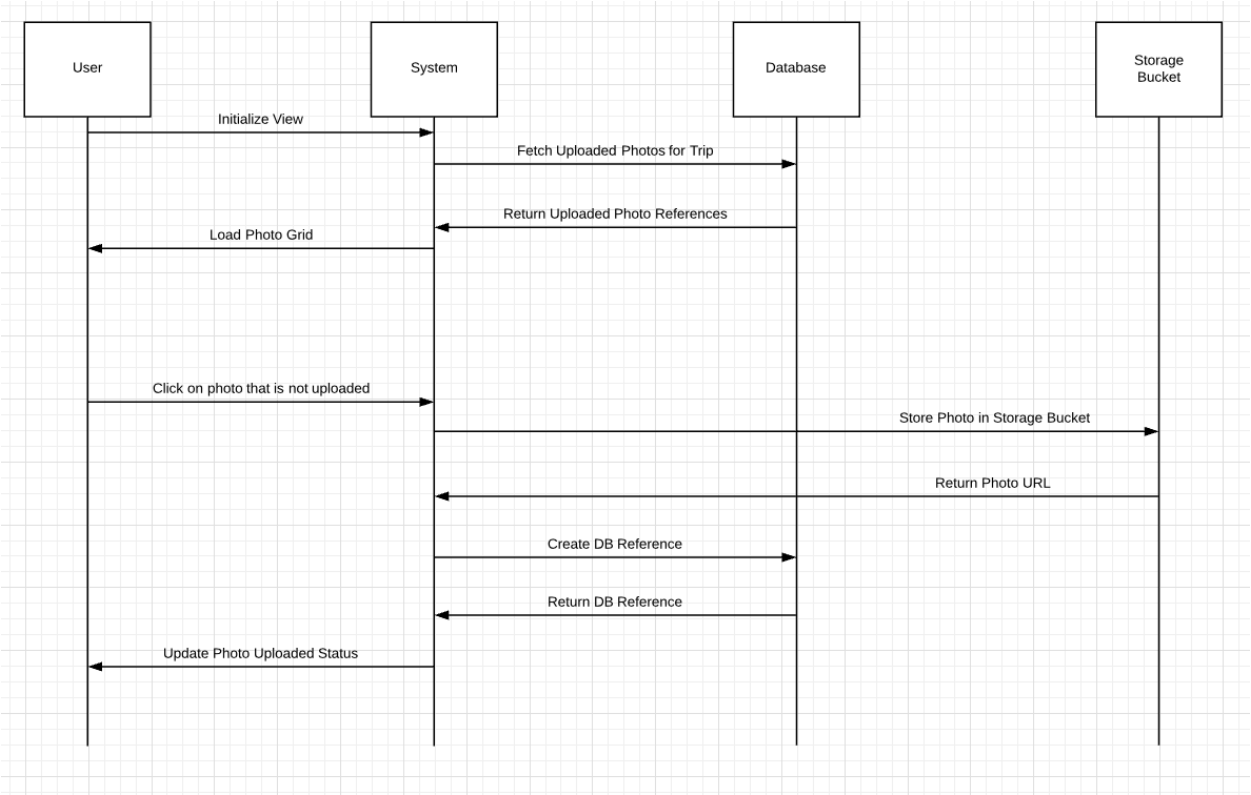
UC-6: FindAttractions
Related Requirements: REQ- 27, REQ- 28, REQ- 29, REQ- 30, REQ-59, REQ-60, REQ-61, REQ-62 **Initiating Actor**: User **Actor's Goal:** To find new attractions based around the user's location. **Participating Actors:** System, Database, Google Places API **Preconditions:** - The user has access to the database - The user has access to the Google Places API - The user has clicked on the Suggestions Tab **Postconditions:** - The user can view different categories of attractions - The user can view different attractions based on the category around the user's location. - The user can save different attractions. - The user can delete saved attractions.
Flow of Events for Main Success Scenario: → 1. The user presses on the suggestions tab ← 2. The system retrieves place information from Google Places API ← 3. The system stores places query locally into a places object. ← 4. The system retrieves saved location information from the database. → 5. The user presses on a category card and is redirected to a list of attractions based on their location. → 6. The user presses the attraction card for more information → 7. The user saves an attraction card ← 8. The system saves the attraction in the database → 9. The user is on the calendar page and clicks on a future date. ← 10. The system retrieves the saved places information from the database

← 11. The system displays the saved place information
→ 12. The chooses which place they want to add to their itinerary

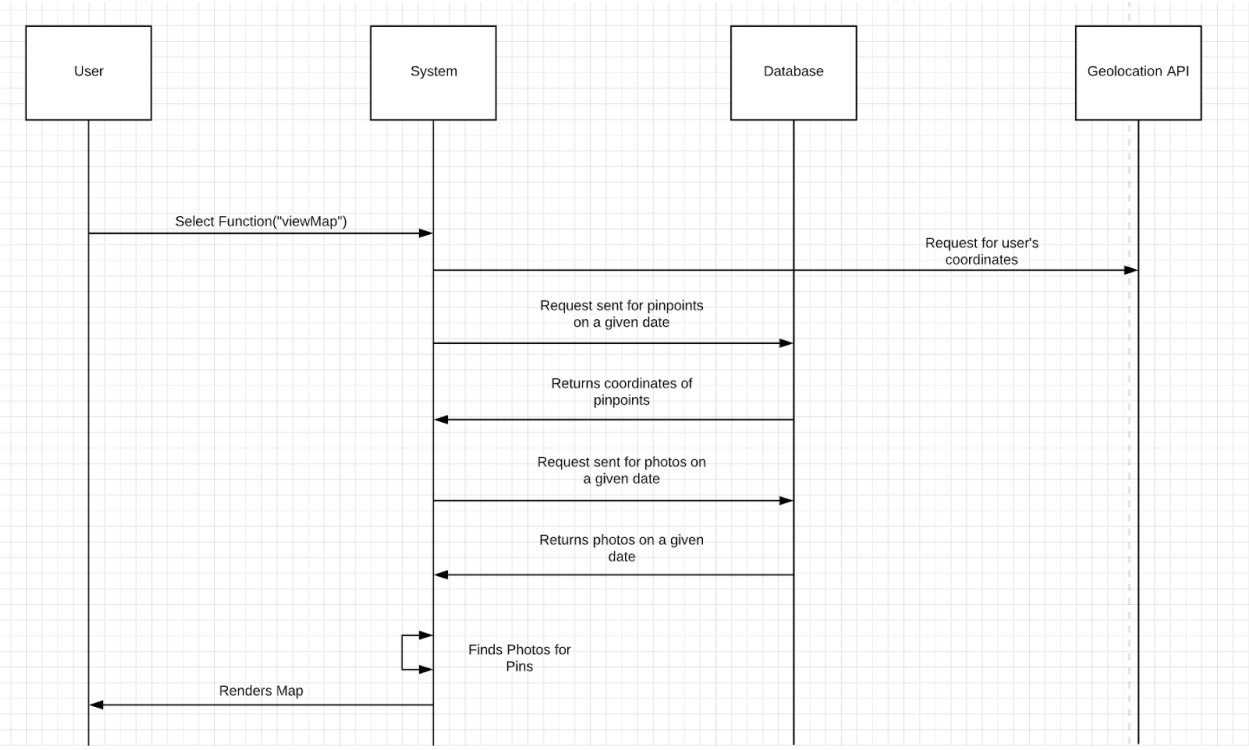
Flow of Events for Alternative Scenario:
← 1. The system fails to retrieve query information from Google Places API, sends an error message to notify users.

D. System Sequence Diagram

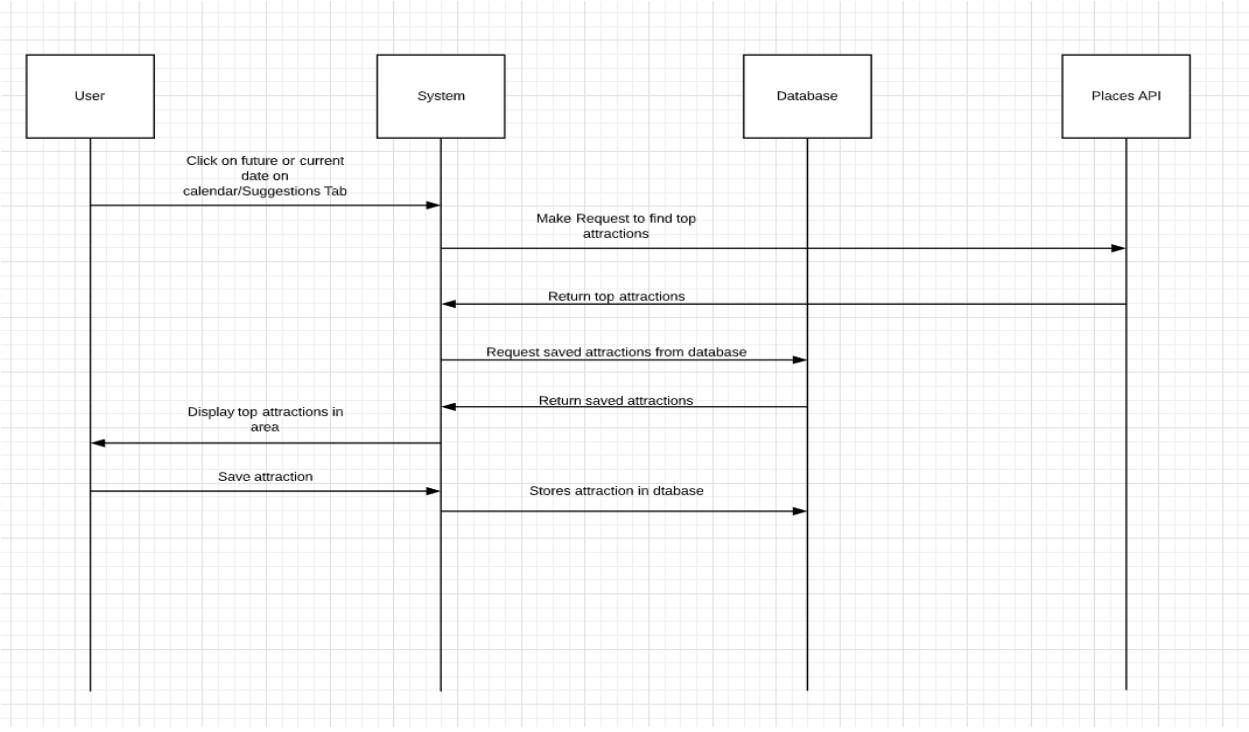
UC-5: AddPhoto



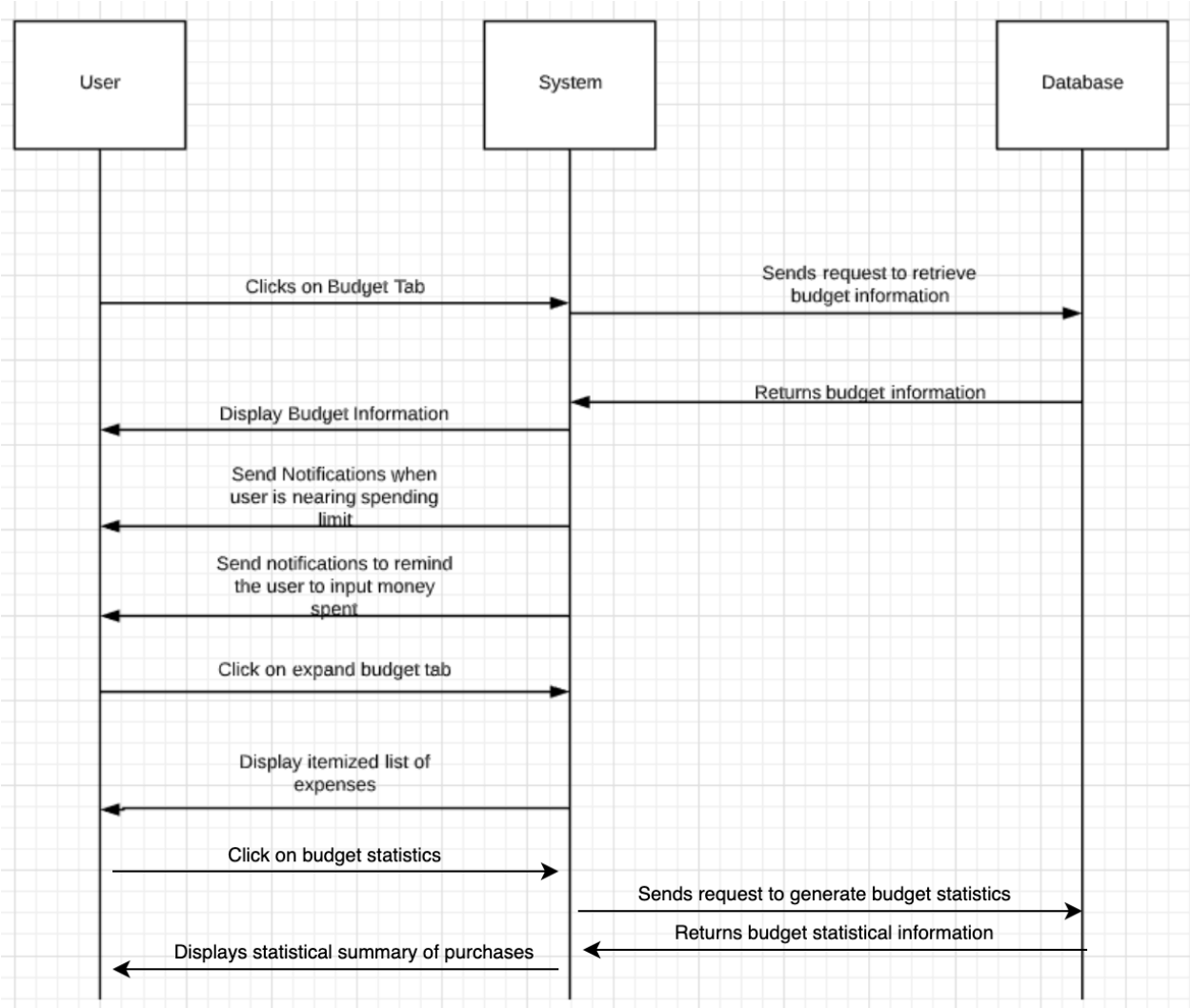
UC-6: ViewMap



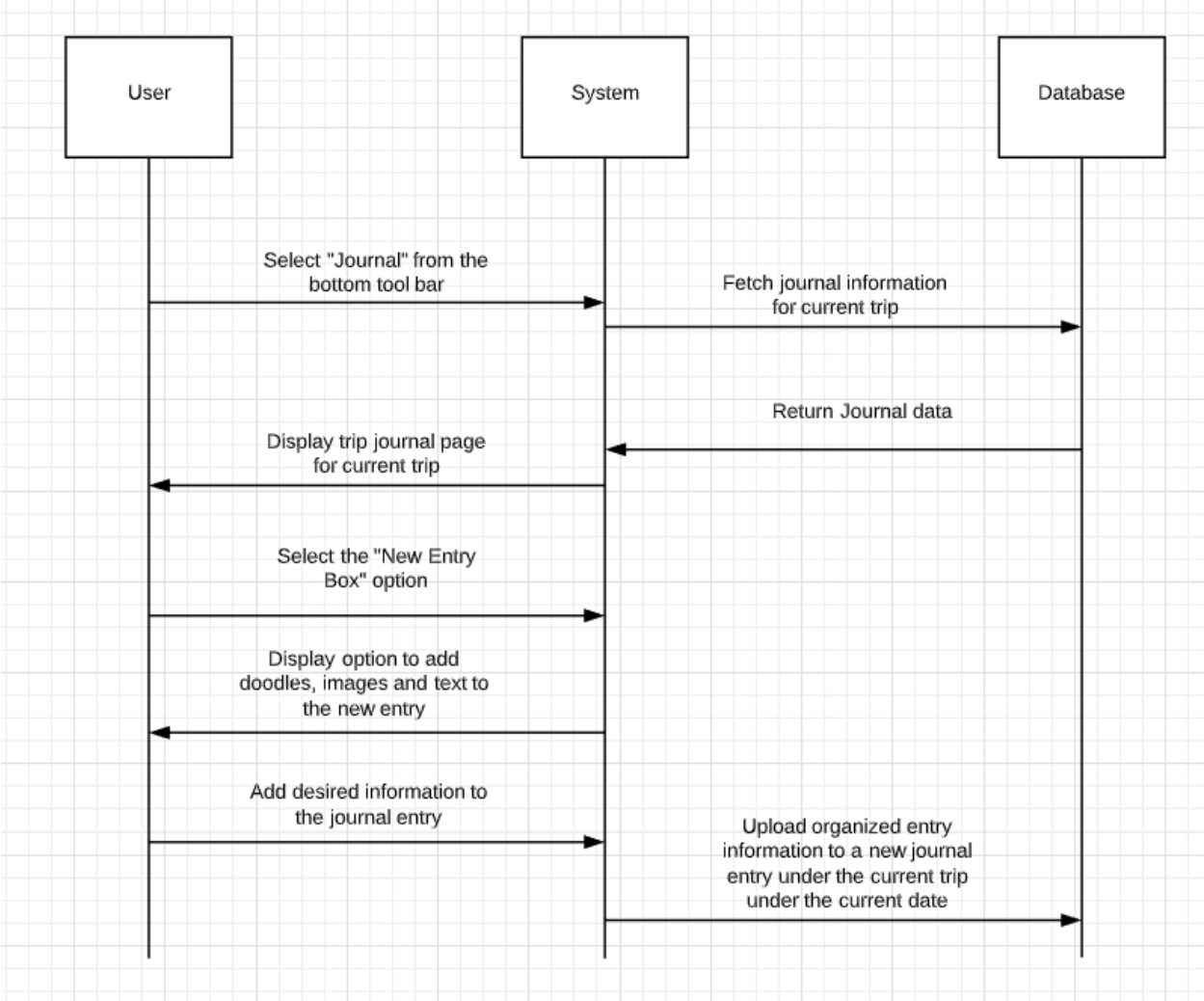
UC-6: Find Attractions



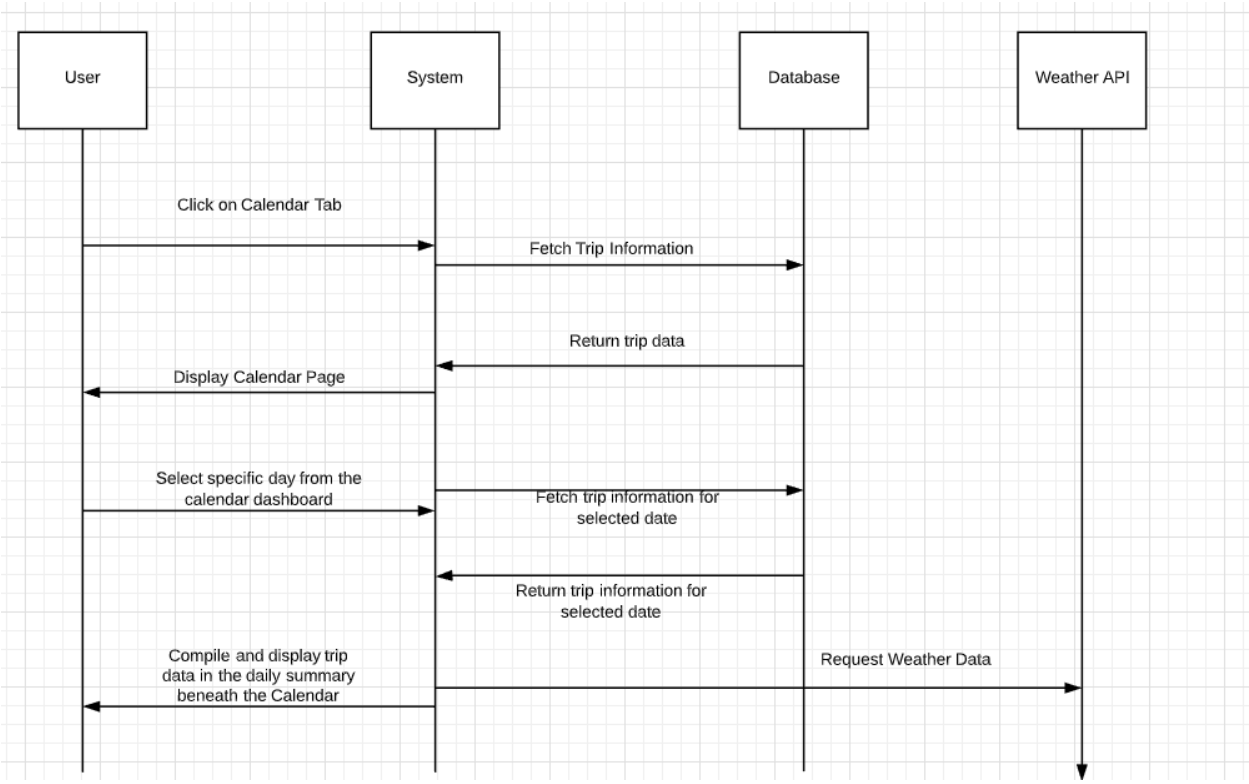
UC-5: Budget


User
System
Database
Clicks on Budget Tab
Sends request to retrieve budget information
Returns budget information
Display Budget Information
Send Notifications when user is nearing spending limit
Send notifications to remind the user to input money spent
Click on expand budget tab
Display itemized list of expenses
Click on budget statistics
Sends request to generate budget statistics
Returns budget statistical information
Displays statistical summary of purchases

UC-8: Journaling



UC-7: View Calendar



5. Effort Estimation using Use Case Points

Unadjusted Use Case Points (UUCP) quantifies our project's functional features based on the amount and complexity of our use cases. To find the UUCP, we must first define our Unadjusted Use Case Weight (UUCW) scheme; in other words we will have to decide what weight value gets assigned to each complexity and why.

Use Case Complexity	Number of Steps	Weight Value
Simple	1 to 3 Steps	5
Average	4 to 6 Steps	10
Complex	7 or more Steps	15

Unadjusted Use Case Weights (UUCW)

Use Case	Description	Category	Weight
UC-1: Login	Simple user interface. One initiating actor and one participating actor. 2 steps for the success scenario.	Simple	5
UC-2: CreateTrip	Moderate interface design. One initiating actor and two participating actors. 4 steps for the success scenario.	Average	10
UC-3: AddPhoto	Complex user interface and processing. One initiating actor and two participating actors. 7 steps for the success scenario.	Complex	15
UC-4: ViewMap	Complex user interface and processing. Three participating actors. 8 steps for the success scenario.	Complex	15
UC-5: Budgeting	Moderate interface design. Three participating actors. 14 steps for the success scenario.	Complex	15
UC-6: FindAttractions	Complex user interface and	Complex	15

	processing. Three participating actors. 12 steps for the success scenario.		
UC-7: ViewCalendar	Moderate interface design. Three participating actors. 7 steps for the success scenario.	Complex	15
UC-8: Journaling	Moderate interface design. Three participating actors. 14 steps for the success scenario.	Complex	15

UUCW = *(Total Number of Simple Use Cases x 5) +*
(Total Number of Average Use Case x 10) + (Total Number of Complex Use Cases x 15)

Then, the UUCW for our system would be (1 * 5) + (1 * 10) + (6 * 15) = 105

Unadjusted Actor Weights (UAW)

Similar to the UUCP, the Unadjusted Actor Weight (UAW) quantifies our project's functional features. However, the UAW is based on the type of our functional actors. The following table shows the standard scheme that we use to classify actors into types:

Actor Classification	Class Description	Weight Value
Simple	External system that must interact with the system using a well-defined API	1
Average	External system that must interact with the system using standard communication protocols (e.g. TCP/IP, FTP, HTTP, database)	2
Complex	Human actor using a GUI application interface	3

Actor	Description	Complexity	Weight
User	User is interacting with the system via a graphical user interface presented in a mobile application	Complex	3
Database	The database holds our data and interacts with the system through a defined API.	Average	2

System	The system holds the controller and therefore is the main “decision maker” for our system. It also reads and compiles information received from the database and will display that information through the user interface (touch-screen display)	Complex	3
GeoLocation	GeoLocation is used to fetch user locations and resolve coordinates and interact with our system through a defined API.	Simple	1
Places	Places is the system used to fetch locations around a user’s location and interact with our system through a defined API	Simple	1
MappingService	MappingService is the system used to fetch map images and render the user interactions with the map.	Average	2

Since the Unadjusted Actor weight is defined as:

> ***UAW*** = *(Total Number of Simple actors x 1) +*
> *(Total Number of Average actors x 2) + (Total Number of Complex actors x 3)*

Then the UAW for our system would be **(2 x 1) + (2 x 2) + (2 x 3) = 12**

Technical Complexity Factor (TCF)

TCF is found by assigning weights between 0 (useless) to 5 (essential) to all of the following technical factors which may come up:

Factor	Description	Weight	Perceived Complexity	Calculated Factor
T1	Distributed System, Web based system	2	4	8
T2	Response time/performance objectives should be exceptional	1	3	3
T3	End-user efficiency	1	3	3
T4	Internal processing complexity is simple	1	3	3
T5	Code reusability should not be too important	1	0	0

T6	Easy to install as an application is very important	0.5	1	0.5
T7	Easy to use as an application is very important	0.5	4	2
T8	Portability to other platforms such as android, iOS, and Web is very important	2	4	8
T9	System maintenance should be easy to have to make changes	1	3	3
T10	Concurrent/parallel processing is important to have geolocation running concurrently	1	3	3
T11	Security features are important since we hold sensitive information	1	2	2
T12	Access for third parties is irrelevant	1	0	0
T13	End user training is not needed	1	0	0
Total:				35.5

Since the Technical Complexity Factors is defined as:

***TCF** = 0.6 + 0.01 x Technical Factor Total*

Then the TCF for our system would be .6 + .355 = 0.955

Environmental Complexity Factor (ECF)

The last factor contributing to the total Use Case Points would be the Environmental Complexity Factor or ECF. The ECF is determined by taking into consideration any environmental conditions that the system will be working in. For the purposes of this system, we will assume ECF to be 1. In other words, the ECF will have no contribution to our UCP calculations.

Use Case Points Calculation

UAW = 12
UUCW = 105
TCF = 0.955
ECF = 1
UUCP = UAW + UUCW = 12 + 105 = 117

UCP = 117 * 0.955 * 1 = 111.735 ≈ **112**

Project Duration Calculation

We are given a productivity factor, PF, of 28.
Then, **Project Duration** = UCP * PF = 112 * 28 = **3136**

6. Domain Analysis

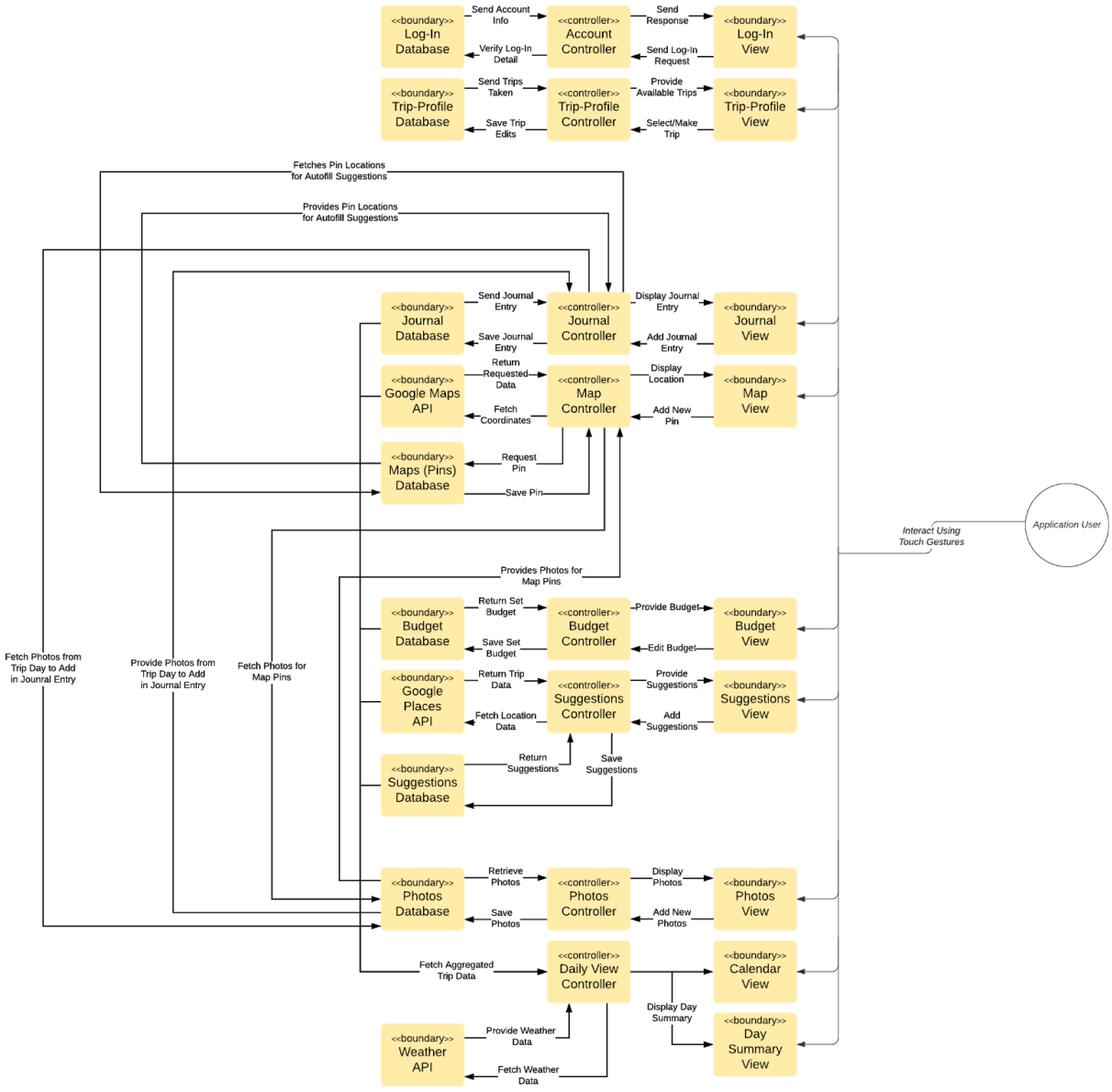
A. Domain Model

I. Concept Definitions

Responsibility	Type	Concept
R1: Store and prepare a database query that retrieves log-in and account validation records	D	Log-In Database
R2: Store and prepare a database query that retrieves created trips	D	Trip-Profile Database
R3: Store and prepare a database query that retrieves stores journal entries	D	Journal Database
R4 To supply a map interface and GPS coordinates corresponding to location	D	Google Maps API
R5: To supply locations for attractions that are within a certain distance from a location	D	Google Places API
R6: To supply the selected day's weather for the user's nearby location	D	Weather API
R7: Store and prepare a database query that retrieves the title, description, and photo associated with a location that the user saved	D	Maps (Pins) Database
R8: Store and prepare a database query that retrieves the associated object information for a particular business including but not limited to, address, price, review, etc.	D	Suggestions Database
R9: Store and prepare a database query that retrieves budgets created and items purchased	D	Budget Database
R10: Store and prepare a database query that retrieves photos	D	Photos Database
R11: Coordinates actions to create new account or log-in and delegate work to other concepts	D	Account Controller
R12: Coordinates actions to create new account or log-in	D	Trip Profile Controller
R13: Coordinates actions to add/edit new journals and retrieve existing entries	D	Journal Controller
R14: Coordinates actions to add/edit location pins and retrieve	D	Map Controller

locations visited		
R15: Coordinates actions to add/create budgets and collect spending trends	D	Budget Controller
R16: Coordinates actions to collect recommendations	D	Suggestions Controller
R17: Coordinates actions to upload new photos	D	Photos Controller
R18: Coordinates actions to provide inputted data for a specific day and calendar dates for trip	D	Daily View Controller
R19: Generated code which shows current concepts associated with log-in/making a new account, what actions can be done, and outcomes of previous actions	K	Log-In View
R20: Generated code which shows current concepts associated with trip profile(s) created, what actions can be done, and outcomes of previous actions	K	Trip-Profile View
R21: Generated code which shows current concepts associated with journal entries, what actions can be done (writing, editing, viewing entries), and outcomes of previous actions	K	Journal View
R22: Generated code which shows current concepts associated with locations visited, what actions can be done (adding pins, viewing places visited), and outcomes of previous actions	K	Map View
R23: Generated code which shows current concepts associated with the budget, what actions can be done (editing the budget, adding limitations), and outcomes of previous actions	K	Budget View
R24: Generated code which shows current concepts associated with trip suggestions, what actions can be done (viewing recommended places), and outcomes of previous actions	K	Suggestions View
R25: Generated code which shows current concepts associated with photos, what actions can be done (adding geotags, uploading/deleting photos), and outcomes of previous actions	K	Photos View
R26: Generated code which shows current concepts associated with the general trip (start and end dates), what actions can be done (selecting different dates), and outcomes of previous actions	K	Calendar View
R27: Generated code which shows current concepts associated with the specific date, what actions can be done (accessing photos, journals, attractions, etc. from the selected day), and outcomes of previous actions	K	Daily Summary View

II. Domain Diagram



Note: In actuality, there is one database (Firebase) that stores information in the system architecture. For practicality in the domain model, it was divided to provide a more detailed interaction schematic (from the subsections within the actual database). **This improved domain model** accounts for increased cross-functionality within features in the application. Additionally, it allows for a more automated experience for the user.

III. Association Definitions

Concept Pair	Association Description	Association Name
Log-In Database ↔ Account Controller	Database provides account information after the controller asks for verifications to access a user account	Provides Data
Trip Profile Database ↔ Trip Profile Controller	Database provides trips associated with the user's account to allow the user to view their separate diaries	Provides Data
Journal Database ↔ Journal Controller	Database provides journal entries previously written to allow users and allow users to save them	Provides Data
Google Maps API ↔ Map Controller	API fetches image of requested coordinates of map and sends it to map controller	Provides Data
Budget Database ↔ Budget Controller	Database fetches budget and expense information to allow users to access their previously entered data	Provides Data
Google Trips API ↔ Suggestions Controller	API fetches a list of locations based on a user's location, so that the user is presented with several locations they can travel to	Provides Data
Photos Database ↔ Photos Controller	Database fetches stored photos from the specific trip the user is looking through and sends them to photos controller	Provides Data
Photos Database ↔ Journal Controller	Database fetches stored photos from the specific trip/date to provide the user the option to add into their journal entry	Provides Data
Photos Database ↔ Maps Controller	Database fetches stored photos from specific trip/date to provide pins with photos whose location was taken within five miles of them	Provides Data
Maps (Pins) Database ↔ Maps Controller	Database fetches pins from specific trip/date that were customally added by the user with their own titles and descriptions	Provides Data
Maps (Pins) Database ↔ Journal Controller	Database fetches queried locations from	Provides Data

	the specific trip/date to provide autocomplete suggestions when the user refers to a location visited	
Suggestions Database ↔ Suggestions Controller	Database fetches saved location information that the user might be interested in visiting at a later session.	Provides Data
Weather API ↔ Daily View Controller	API provides fetched weather information based on the user's current location	Provides Data
Google Maps/Trips API, Journal/Budget/Photos/Suggestions Database ↔ Daily View Controller	Database provides all listed information that users inputted to a specific date on their trip to provide collected data	Provides Data
Account Controller ↔ Log-In View	Displays login textboxes and redirects user to homepage once successful login has been complete	Conveys Request
Trip-Profile Controller ↔ Trip-Profile View	Displays list of trips that the user has created, which can be opened once pressed	Conveys Request
Journal Controller ↔ Journal View	Displays journal entries for user's view and handles when user writes or edits an entry	Conveys Request
Map Controller ↔ Map View	Displays comprehensive image of map with the tracking information drawn onto it and pins placed onto it	Conveys Request
Budget Controller ↔ Budget View	Displays budget information that the user entered in	Conveys Request
Suggestions Controller ↔ Suggestions View	Displays list of suggestions for user to pick from to save to their trip schedule	Conveys Request
Photos Controller ↔ Photos View	Displays a list of all the photos saved by the user into the app and allows user to interact with pictures (zoom in and out)	Conveys Request
Daily View Controller ↔ Daily View	Displays aggregated information from entire trip and trip dates for the user to interact and view a specific day's summary of the inputs to their diary (photos, locations, etc.)	Conveys Request

IV. Attributes Definitions

Concepts	Attributes	Description
Log in	1. User Identification	1. Username and Password
Budget	1. Budget Creation/Edit 2. Statistics 3. Notifications	1. Can add budget, set category, set reminders, etc. 2. Average amount spent per day, and which category they spend on the most 3. Reminds user when to add expenses to budget
Suggestions	1. Suggestion Retrieval 2. Saved Attractions 3. Notifications	1. Retrieves several locations and location information around a user's destination 2. Can save attractions for future viewing 3. Can commit to an attraction, which will send notifications to remind the user when to go
Map	1. List of pins 2. Location 3. List of photos	1. Pin objects are rendered onto the view on creation of the map 2. The user's location is updated by the Geolocation API when the user taps on a new location on the map 3. Photos which are within a 5 mile radius of a pin are displayed when a pin is pressed
Calendar	1. Dates/Length of Trip 2. Current date 3. Days	1. Conveys when the trip takes place and for how long 2. Current date to be displayed in calendar 3. The calendar has access to different day objects of the trip
Daily Summary	1. Day Identifier 2. Events List	1. Identifies which day of the trip is selected 2. Events that the user has or plans to participate in. This includes journal entries, photos, events, suggestions, etc.
Journal	1. Creation Method 2. Random Prompt 3. Autocomplete feature	1. Can be writing, drawing, taking a photo, uploading a video, etc. 2. A random writing prompt will question the user to help spur creativity and ease the process of writing

V. Traceability Matrix

Use Case	PW	Domain Concepts								
		Google Maps API	Google Places API	Weather API	Login View	Trip Profile View	Map View	Budget View	Journal View	Daily View
UC-1	15				x					
UC-2	19									
UC-3	27					x				x
UC-4	25	x				x	x			x
UC-5	24					x		x		x
UC-6	24		x							
UC-7	22			x		x				x
UC-8	22					x			x	x
Max PW		25	24	22	15	121	25	25	23	121
Total PW		25	24	22	15	121	25	25	23	121

Use Case	PW	Domain Concepts									
		Login DB	Trip Prof DB	Journal DB	Budget DB	Photo DB	Map DB	Sug. DB	Sug. View	Photos View	Cal View
UC-1	15	x									
UC-2	19										
UC-3	27		x			x				x	
UC-4	25		x				x				
UC-5	24		x		x						

UC-6	24							x	x		
UC-7	22		x								x
UC-8	23		x	x							
Max PW		15	121	23	25	27	25	24	24	27	22
Total PW		15	121	23	25	27	25	24	24	27	22

Use Case	PW	Domain Concepts								
		Account Cont.	Trip Prof Cont.	Journal Conl.	Map Cont.	Budget Cont.	Sug. Cont.	Photos Cont.	Cal. Cont.	Daily View Cont.
UC-1	15	x								
UC-2	19									
UC-3	27		x					x		x
UC-4	25		x		x					x
UC-5	24		x			x				x
UC-6	24						x			
UC-7	22		x						x	x
UC-8	22		x	x						x
Max PW		15	121	22	25	24	24	27	22	121
Total PW		15	121	22	25	24	24	27	22	121

B. System Operation Contracts

1) Photos

Responsibilities: Photos need to be uploaded to a given album. The system must store photos by querying a database and also be efficient in storing photos, meaning it doesn't exceed memory. It must also notify you of failure.

Cross References: UC-3: AddPhoto

Expectations: As discussed, the system should be able to query a database in order to add a new entry for a photo. If there is a failure to access the database, then the system should provide an alert detailing the causes of failure.

Preconditions: The application must be given permission to access the photo. There should be access to a database that is either set up or can be set up.

Postconditions: Notify the user that the picture has been uploaded and refresh the page. If it failed, then there should be an alert detailing why.

2) Map

Responsibilities: The system should load the pinpoints that the user has saved during their trip onto a map. It needs to store metadata about each location such as time visited, coordinates, photos taken near it, etc. The system should load the map so that all pinpoints can be viewed.

Cross References: UC-4: ViewMap, UC-3: AddPhoto

Expectations: The system should query the database to retrieve all necessary information needed to form the map, particularly the pinpoints, including photos from the trip diary that were taken near a pinpoint. Then the system should calibrate the map so that it fits all pinpoints into a view. Also, if there are no pinpoints then the system should center the map based off of a generic coordinate.

Preconditions: Connection is or can be established with the database to retrieve pinpoint metadata. If there are no pinpoints, then there we can come up with a generic location, using an API, to center the map.

Postconditions: The system renders the map around the coordinates of the pinpoints.

3) Budget

Responsibilities: The system should save and delete finances that the user inputs into their budgeting. The system should be able to retrieve information so that budget statistics can be made.

Cross References: UC-5: AddBudget

Expectations: The information should be stored in a database so that the system can develop charts, graphs, and statistical variables to present to the user as a visual and mathematical view of their spending habits.

Preconditions: Database can store financial information that the user inputs and there are API's used to generate suggestions for finances.

Postconditions: System develops budget statistics with all the budgeting information.

4) Attractions

Responsibility: The system should be able to retrieve business place information based on a user's location. The system should also save attractions so that the user can view them later and should also allow the user to delete attractions they are no longer interested in.

Cross References: UC-6: FindAttractions

Expectations: The system is expected to use the Google Place API to query for business information, and connect to the database to retrieve saved location information. Once the system has this information, the system should format and display the information so that the user can interact with it.

Preconditions: The application can successfully connect to both the database and API to retrieve relevant information and has access to the user's current gps coordinates, so that the system can query for places that are near the user's current location.

Postconditions: The system displays businesses near the user's location and successfully handles the saving and deleting of these businesses from the user's personal saved attraction list.

5) View Calendar

Responsibility: The system needs to retrieve the user's saved photos, journal entries, events, budget from a database relating to the chosen day. It must then generate a page for the user to see all this information and provide links to add photos, journal entries, etc. It must also notify the user of any failure such as not having access to the Internet.

Cross References: UC-3: Add Photos UC-4: View Map UC-6: Find Attractions UC-5: Budgeting UC-8: Journaling

Expectations: The system requests the information from the database, filtering all the information based on what day they were added. Once the system has the information, it will generate a new page on the mobile device displaying all of the retrieved information. The system will also provide the user a link to add to the chosen day.

Preconditions: The system must be able to establish a connection to the database so that information can be transferred which includes the user retrieving previously added information as well as the user saving information to the database. The system must also have access to the user's photos/gps location if they want to save those to the specific day.

Postconditions: All of the user's saved information pertaining to the chosen day is displayed on a new page on their mobile device. Links are given in which they can add or save new information.

6) Add Journal Entry

Responsibility: The system will generate a new page on the mobile device for the user to make their journal entry. The system will then generate a random prompt as well as provide the user with links/buttons to direct them to a page where they can add photos, videos, or drawings. The system can then save the journal entry to the desired day/trip in the database.

Cross References: UC-8: AddJournal, UC-7: ViewCalendar/DailySummary, UC-3: AddPhoto, UC-4: ViewMap

Expectations: The system should be able to operate on its own without the database to generate the page, and create the prompt. However, the database will be used if the user decides to add a photo or save the journal entry.

Preconditions: The user must be able to navigate through the daily summary page or toolbar to where they can add a journal entry to a specific trip/day. There must also be a database connection so that the user can access their photos on that chosen day or save the journal entry.

Postconditions: The user creates a journal entry with photos, videos, drawings, writing and saves it to the desired day or trip. They will also be given the option to add locations to their journal based on locations in the map.

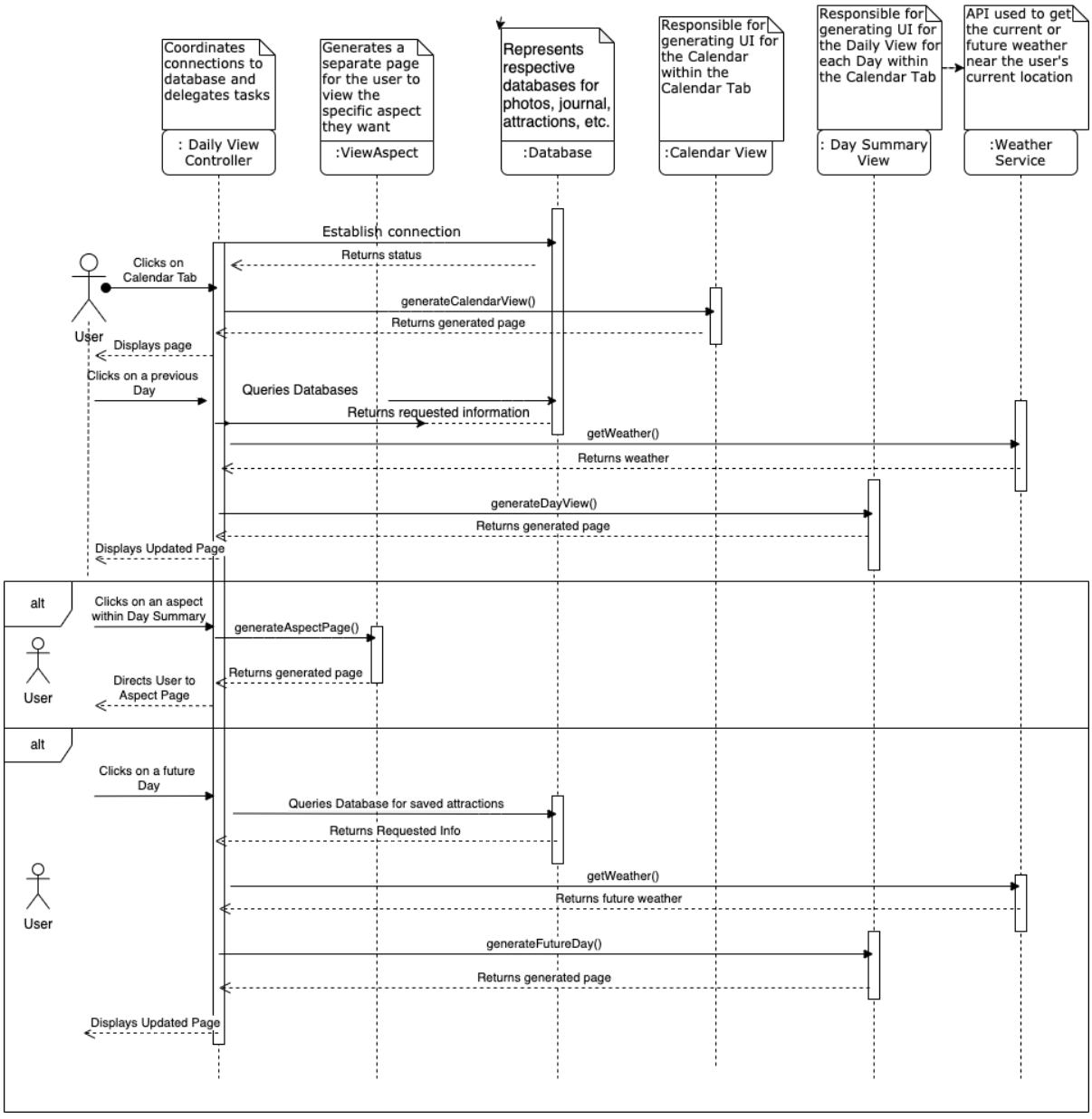
7. Interaction Diagrams

A. Sequence Diagrams / Descriptions

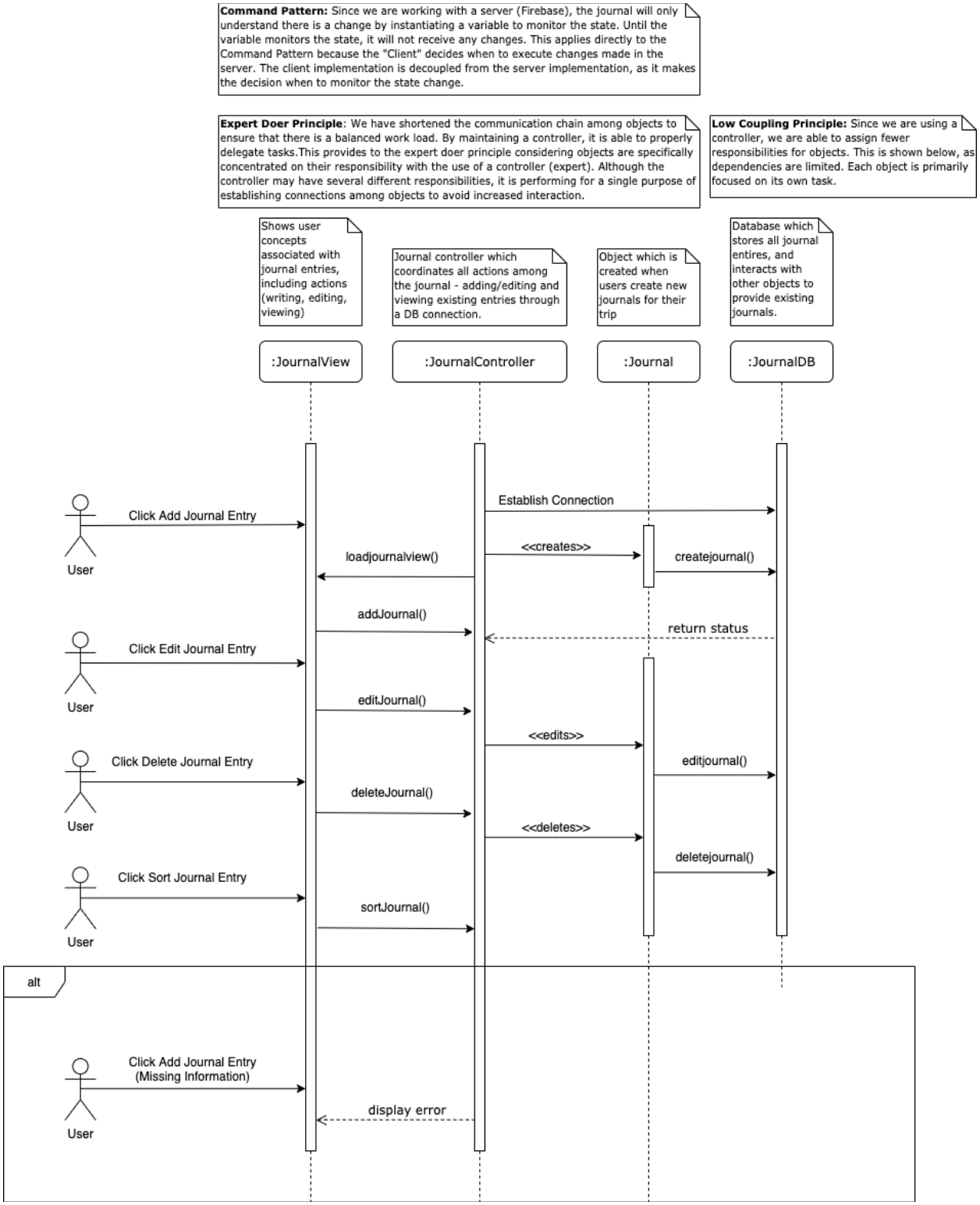
1. UC-7: ViewCalendar/DailySummary

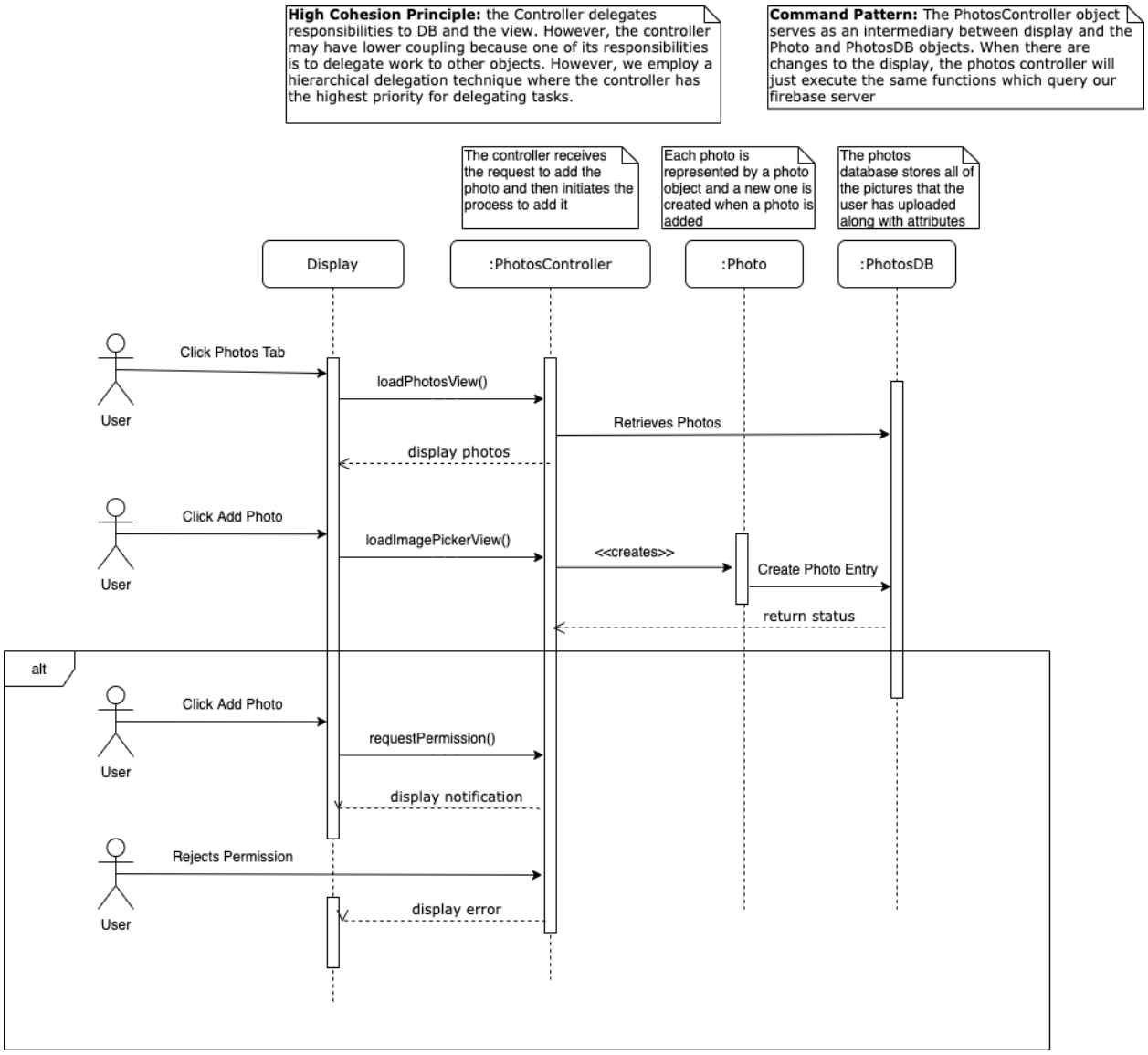
High Cohesion Principle - by utilizing a DayController to delegate tasks, we are minimizing the amount of responsibilities per object therefore increasing cohesion

Publisher-Subscriber Pattern: The Daily View Controller acts as the publisher and the respective Databases and Views are the subscribers. When an event occurs, the Daily View Controller notifies a subscriber that it has occurred, and the subscriber knows what to do and executes its function.



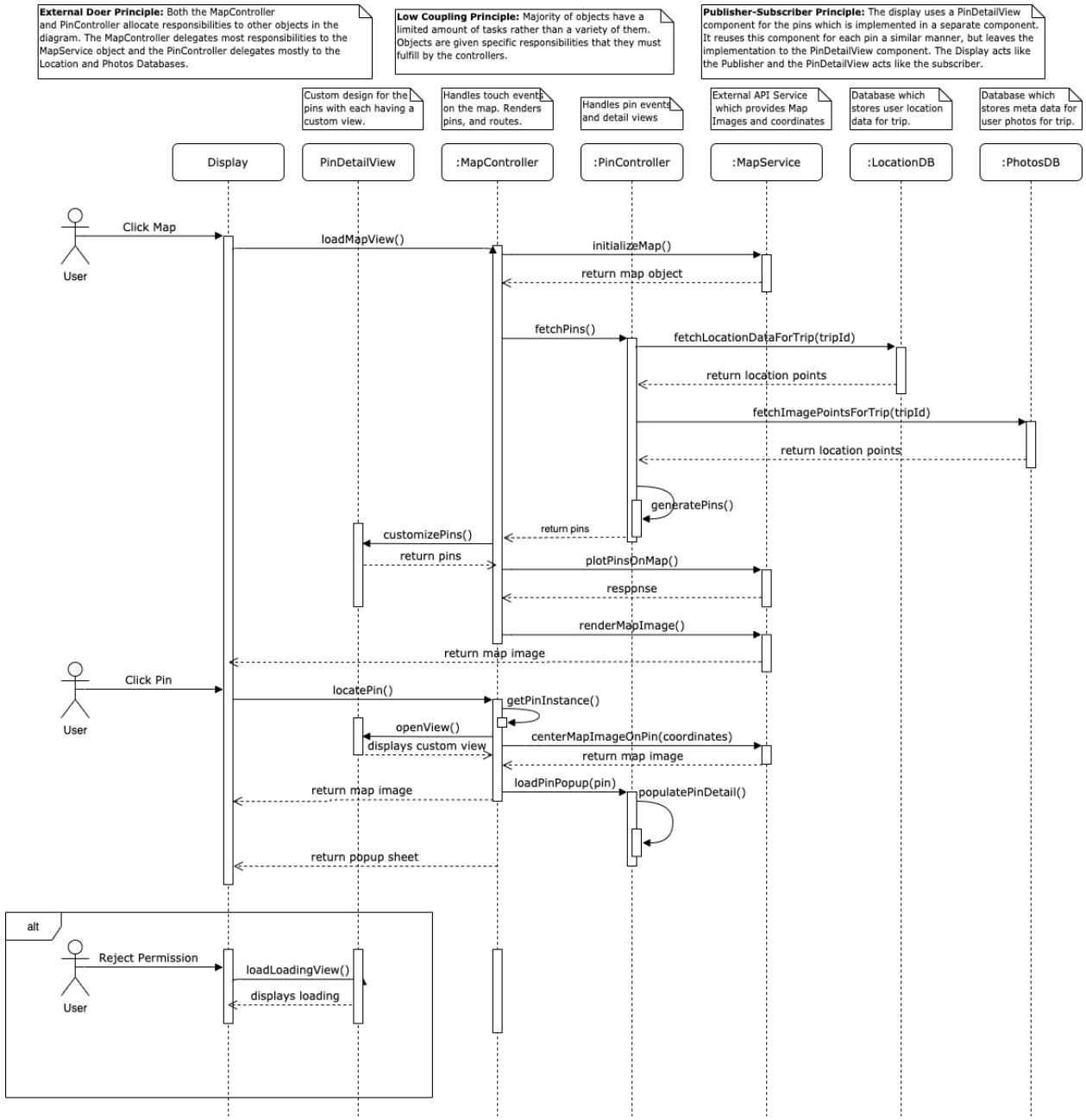
2. UC-8: AddJournal


Command Pattern: Since we are working with a server (Firebase), the journal will only understand there is a change by instantiating a variable to monitor the state. Until the variable monitors the state, it will not receive any changes. This applies directly to the Command Pattern because the "Client" decides when to execute changes made in the server. The client implementation is decoupled from the server implementation, as it makes the decision when to monitor the state change.
Expert Doer Principle: We have shortened the communication chain among objects to ensure that there is a balanced work load. By maintaining a controller, it is able to properly delegate tasks.This provides to the expert doer principle considering objects are specifically concentrated on their responsibility with the use of a controller (expert). Although the controller may have several different responsibilities, it is performing for a single purpose of establishing connections among objects to avoid increased interaction.
Low Coupling Principle: Since we are using a controller, we are able to assign fewer responsibilities for objects. This is shown below, as dependencies are limited. Each object is primarily focused on its own task.
Shows user concepts associated with journal entries, including actions (writing, editing, viewing)
Journal controller which coordinates all actions among the journal - adding/editing and viewing existing entries through a DB connection.
Object which is created when users create new journals for their trip
Database which stores all journal entires, and interacts with other objects to provide existing journals.
:JournalView
:JournalController
:Journal
:JournalDB
Establish Connection
Click Add Journal Entry
User
<<creates>>
loadjournalview()
createjournal()
addJournal()
return status
Click Edit Journal Entry
User
editJournal()
<<edits>>
Click Delete Journal Entry
editjournal()
User
deleteJournal()
<<deletes>>
deletejournal()
Click Sort Journal Entry
User
sortJournal()
alt
Click Add Journal Entry
(Missing Information)
User
display error

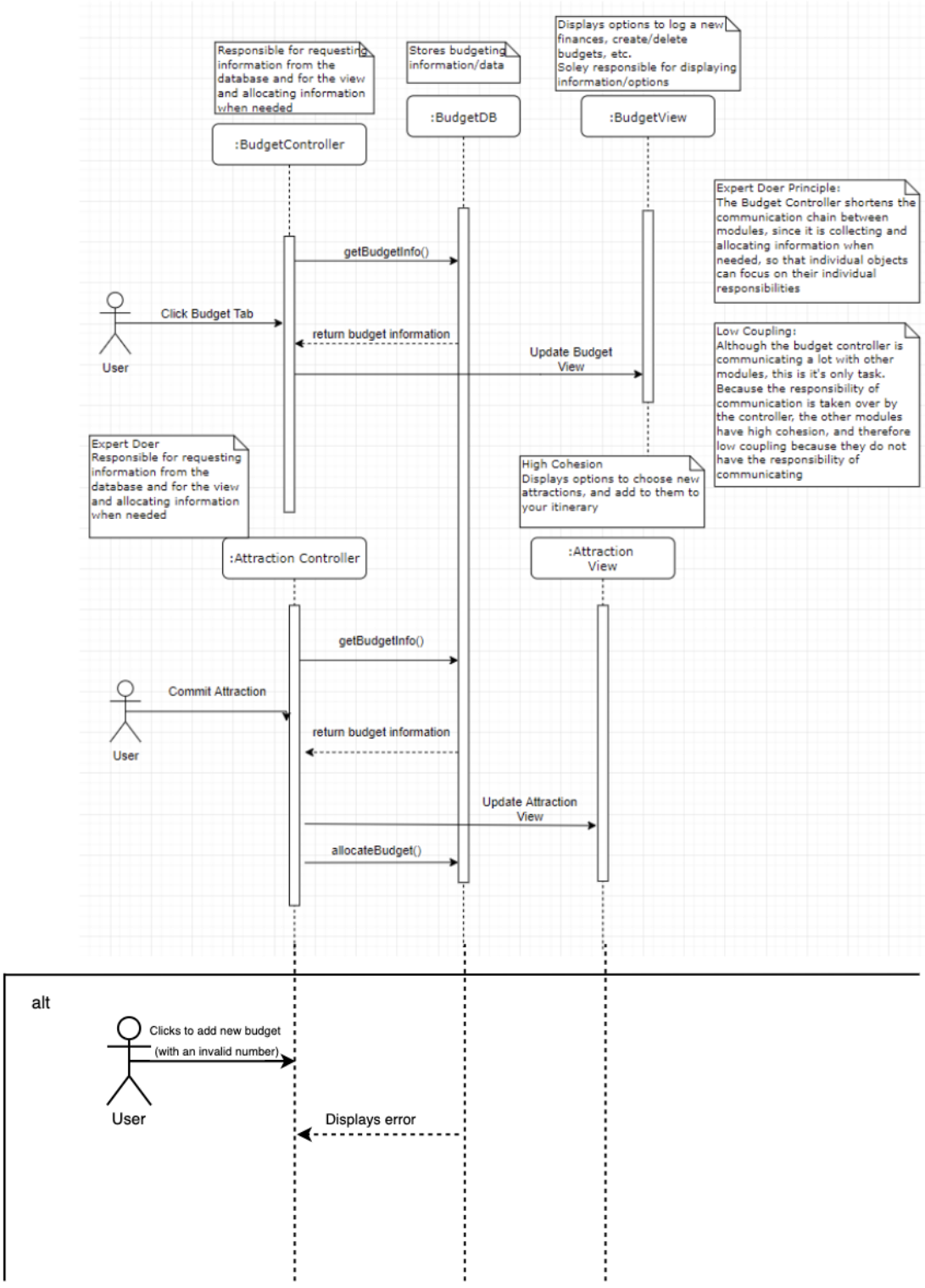


3. UC-3: Add Photo

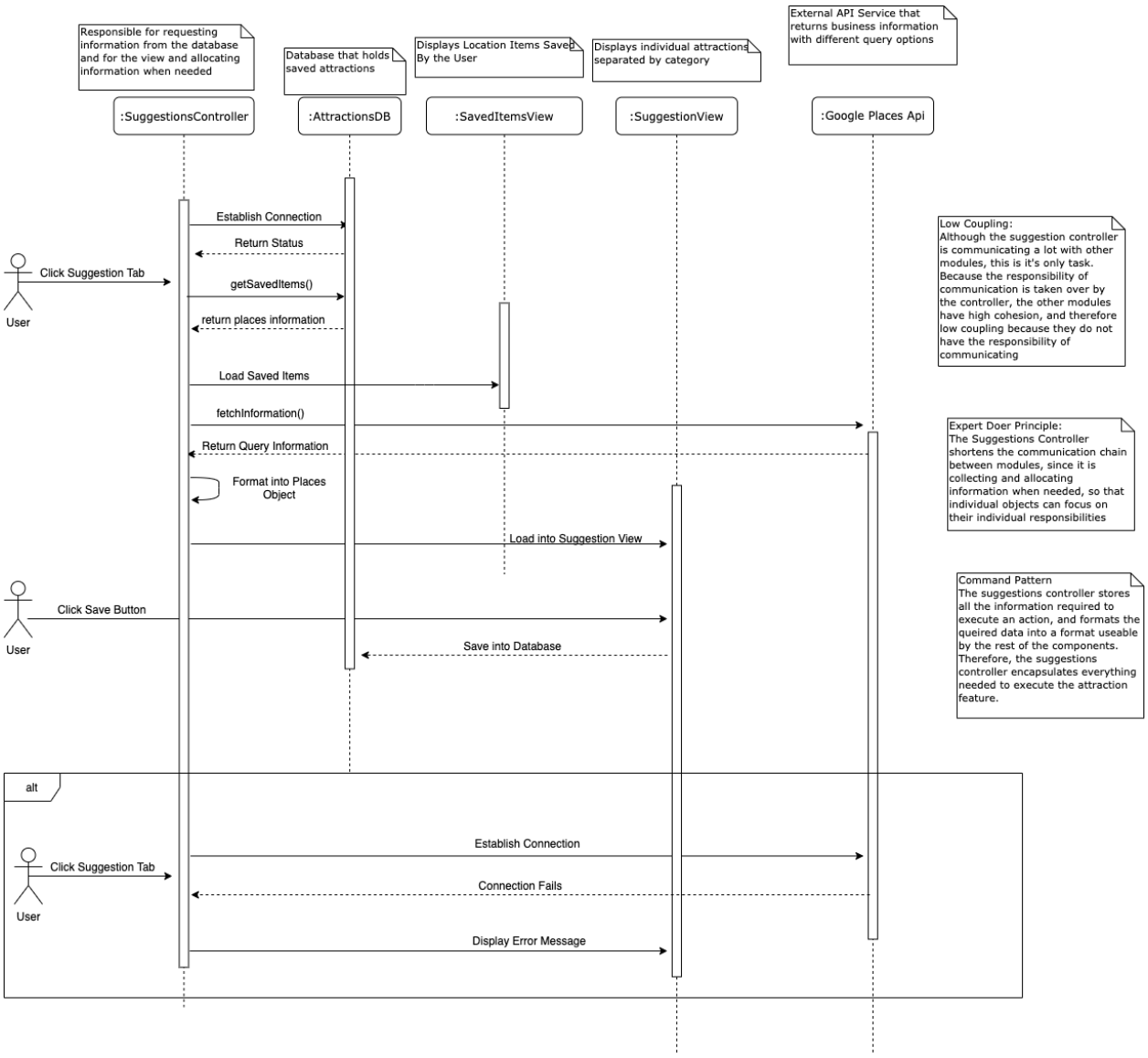
4. UC-4: View Map



5. UC-5: Add Budget



6: UC-6 Find Attraction



8. Class Diagram + Interface Specification

A. Class Diagram

CLASS DIAGRAM TOO LARGE TO BE READABLE ON THE DOC. PLEASE FOLLOW THIS LINK TO VIEW AN IMAGE.

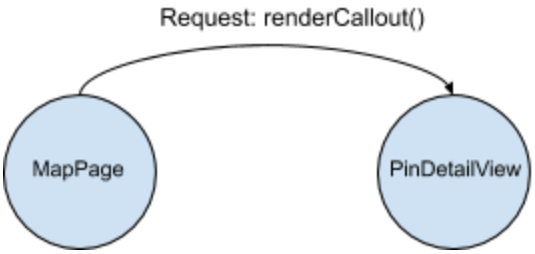
B. Design Patterns

1. Photos

High Cohesion Principle	Command Pattern
In the Photos module of this application, the implementation of the model view controller framework enforces a strict delegation of tasks across the various components that enable this feature. The PhotosPage strictly handles the rendering logic of this feature, where it creates a grid layout and the appropriate references to each photo object. The objects are then further passed to an ImageTile component which handles the individual photo rendering piece, and handles click events to pass to another component which deals with the logic for opening the photo in its full view. Finally, we have the Photo class which handles the data fetching, storing, and manipulation of photo objects in our application. As per the high cohesion principle, this implementation encourages each component to hold a well defined purpose.	In the Photos module, the PhotosPage serves as the primary controller which delegates the tasks to other components within the module. It handles the initialization of the module, calls the appropriate data collection functions, and delegates the rendering logic to make use of the incoming photo data to present the user with all of their photos on the device as well as those that are uploaded to the storage bucket.
Advantages	
1. **Delegation of Responsibility -** Applying this principle to our project enables the data fetching of this component separate from the rendering logic. It further creates more granular control over the rendering	1. **Delegation of Responsibility** - While each component is individually capable of executing their functions, the command pattern allows us to create a separate component to deal with user interaction and changes in

logic and allows us to modify the general layout of the grid, for example, without affecting the layout of the image tile. 2. **Modularity & Readability -** This principle also enforces modularity and readability as it isolates the visual logic from the data & computation logic. This allows other components to use the Photo class to fetch objects in other components without interfering with the Photos module.	data and call the appropriate components to ensure a cohesive system. 2. **Efficiency** - This pattern also ensures that each component in this module is only called upon when they are needed. As the primary delegator, the PhotosPage controller uses an event-based approach to perform tasks rather than having each component periodically check for new data.

2. Map

Publisher-Subscriber Pattern	**Expert Doer Principle**	**Low Coupling Principle**
Diagram(s)		
 **Fig: Publisher-Describer Components**		
In the map, we are using request-based communication in the form of a direct request among the MapPage and PinDetailView components. When the MapPage renders a pin onto the map, it wants the pin to have a custom callout displayed when the pin is pressed. It delegates this responsibility to the PinDetailView component. In this case, the MapPage is the publisher which provides information to the subscriber,	The expert doer principle is demonstrated in the relationship between the controller classes and the view and model classes. In the MapController, we delegate the custom callout in the MapPage to the PinDetailView; we fetch pins from the Pin class; and new pins are added to the MapPage using the AddPinPage component.	The components of the map have been set up so that they are not too dependent on one another. We have opted to use the MVC design pattern which creates a hierarchy of components, where the controller is chiefly responsible for delegating tasks. The map components consist of the MapController, MapPage, PinDetailView, Pin, and AddPinPage. The MapController delegates tasks to the PinDetailView, Pin, and AddPinPage components, and each of

PinDetailView, in order to render a custom callout.		these components do not rely on one another.
Advantages		
1. **Delegation of responsibility** - to avoid cluttering up the MapPage with front-end logic, we allow the front-end logic to be rendered in PinDetailView 2. **Reusable component** - both auto generated pins based off of attractions and user generated pins can be rendered with this component 3. **Modularity & readability** - by using a subscriber component our code becomes more modular, which is beneficial for readability. It is also more readable in the sense that it follows the "IF-THEN-ELSE" rule: if there is a pin, then render it with PinDetailView callout. When reading the MapPage, the logic is more clear.	3. **Readability** - the map code is extensive and if the code was contained in one class it would be very difficult to track what is going on. 4. **Delegation of responsibility** - each component is aware that they need to execute tasks requested by the controllers and then send back their responses	1. **Efficiency** - The lack of dependencies in the components means they don't get blocked by each other, waiting for events to finish 2. **Reliability** - If there was high coupling, then when one link breaks, the entire component may break

3. Journal

Command Pattern	Expert Doer Principle / Low Coupling Principle
Considering the use of Google's 'Firebase', the journal uses a state variable to monitor real time changes when a user will update,	The use of a controller (expert), allows for the proper delegation of tasks within the journal feature. Additionally, through the use of the

store, delete, or add a journal entry. The changes will be shown only when the app renders, which will be every time a change in state has occurred. The 'client' will decide when to execute changes made by the database server.	MVC model throughout the entire application, different realms of interaction are only required to execute their responsibility (ex. Model view will only interact with the database). The concentration of responsibility specifically aligns with the 'Low Coupling Principle' as dependencies are limited.
Advantages	
Due to the state of journals being updated only once the app renders (loads or sees changes), we are able to control when the client will execute changes found in the database. This provides for the application to avoid constantly monitoring the database, but instead only needing to execute changes once it knows the user has provided them.	Within the journal, the controller will interact with the different pages to add/edit, delete, and update the journal entries. Any database interaction will be found in the model view, which provides for the controller (expert) to simply delegate tasks.

4. Daily Summary / Calendar

High Cohesion	Publisher/Subscriber
In the calendar, CalendarPage acts as a controller, calling separate classes and functions to display the correct and updated UI page. This is seen in where the Calendar calls methods such as fetchJournals(), fetchPhotos(), and updateSelectedDay(). Each method knows exactly what they are doing, and the CalendarPage delegates the tasks to these methods accordingly, thereby increasing cohesion.	In the calendar, we are using a request based system where the calendar will request that the journals and photos be updated. This goes to the fetchJournals() and fetchPhotos() methods() which process the request. CalendarPage is the publisher who requests information when an event occurs and fetchJournals() and fetchPhotos() are the subscribers.
Advantages	
This keeps the coder more simple, neat, and easier to understand. It is also more beneficial for reusability as classes that have a specific job, such as fetching journals, can be reused for a later purpose by some other	This helps with code readability as functions are separated into different components making it easier to understand. This also increases the possibility of reusable code as

class. If everything was hard coded without functions or methods and CalendarPage was doing everything, it could not be reused.	the subscribers can be utilized by different publishers and classes.

5. Budget

Low Coupling Principle	Command Pattern
The budget controller shortens the communication chain between modules by collecting and allocating information when needed. The controller communicates with a lot of modules, but this is the only task of the controller.	The budget controller fetches information about the trip's budget and the trip's purchasing history. The budget controller also calculates the approximate budgeting that must be done per day to stay within the budget. By doing this, the budget controller includes all the tasks required for the functionality of the budget feature.
Advantages	
Because the responsibility of communication is taken over by the controller, the other modules have high cohesion. The other classes can work on their specific tasks.	The controller allows there to be a central source of the information that is displayed and changed by the user. This keeps the programming used for the changes separate, organizing the code by a central controller and by classes that are either objects or functions used by the controller.

6. Attractions

Low Coupling	Command Pattern
Low Coupling: Although the suggestion controller is communicating a lot with other modules, this is it's only task. Because the responsibility of communication is taken over by the controller, the other modules have high cohesion, and therefore low coupling because they do not have the responsibility of communicating	The suggestions controller fetches business information from the Google Places API and stores all the information, and formats the queried data into a format usable by the rest of the components. Therefore, the suggestions controller encapsulates everything needed to execute the attraction feature.

Advantages	
Having the suggestion controller allows other classes to specialize and focus on their specific responsibilities. This design method allows us more freedom to edit classes without worrying about how they will impact the relationships with other classes.	This helps organize the code better since all the methods and logic behind retrieving data are located in one class. Thus, any component that needs to access this data can just call this class, which reduces redundant work.

A. Data Types + Operation Signatures

a. Photos

Photos
Attributes
1. **id : string** - The database reference ID for this photo 2. **deviceId : string** - Hash computed from the UUID of the device the photo was taken on and the photo's creation timestamp. 3. **uri : string** - The URL location of the photo file. This can either be an HTTPS link if it's referencing an uploaded photo, or an assets:// URI if the photo exists on device. 4. **tripId : string** - The database reference ID for the associated trip object for this photo. 5. **userId** : **string** - The database reference ID for the user that owns the photo 6. **location** : **GeoPoint** - An array of numerical values representing the latitude and longitude. 7. **city** : **string** - The city found by reverse geocoding the coordinates from the photo. 8. **state : string** - The region found by reverse geocoding the coordinates from the photo 9. **creationTime : Timestamp** - The timestamp for when the photo was created. The timestamp is stored as milliseconds from epoch. 10. **tags : [string]** - Stores relevant tags for the photos.
Operations
1. **constructor(obj)** : **Photo** - Creates a photo object from the JSON response from the database. 2. **toJSON() : Object** - Creates a JSON map from the Photo object to store into the database. 3. **storePhoto() : void** - Stores the photo object reference into our database. 4. **fetchPhotos(tripId) : [Photo]** - Fetches the photo references by tripId and returns an array of Photo objects.

PhotosPage
Attributes
1. **libraryPermission : boolean** - The current system permission for loading photos from the user's device library. 2. **devicePhotos : [Photo]** - Set of photos on device that meet the trip duration 3. **uploadedPhotos : [Photo]** - Set of uploaded photos for the current trip 4. **loading : boolean** - Boolean flag used for rendering logic on if the component is loading & processing data.
Operations
1. getPhotos() : void - Fetches the current trip from the state, the photos stored on device, and photos stored in storage buckets and notifies the appropriate rendering functions to update the view. This function is called on initialization and whenever the user requests a refresh on the view. **2. uploadPhoto(photo) : void** - User initiated function to start the photo upload process. This function processes the selected photo for the additional metadata, and prepares the object to be then passed into the Photo component to upload the data to Firestore. **3. renderImageTile({ photo, index }) : React.Component** - Callback function used to render a single image tile that is passed into the List builder to separate rendering logic. **4. render() : React.Component** - Primary function to render the view. Switches between a loading spinner and a List view based on state.

ViewPhoto
Attributes
1. photo : Photo - The photo reference to be shown in full view.
Operations
1. render() : **React.Component** - Creates a full screen view containing the Photo, and relevant metadata stored as part of Image properties section.

b. Map

MapPage
Attributes
1. **trip : Trip** - trip object contains the tripId and dates, which the MapPage needs to fetch pins from a trip 2. **mapRegion : Object** - contains latitude, longitude, latitude delata, longitude delta which provides a location and a map sizing for the map to be rendered 3. **initialRegion : Object** - same contents as mapRegion but initialRegion is what the map is centered on originally 4. **hasLocationPermissions : boolean** - true if the user grants permission to access his/her location, false if rejects permission **5. deleteMode : boolean** - true if the user presses on the delete button in the header, which allows one to delete pins 6. **numAutoPins : int** - the number of pins which correspond to attractions the user has saved 7. **pins : [Pin]** - consists of first pins generated from attractions and then pins the user created 8. **photos : [Photo]** - consists of photos user has uploaded for this trip so that we can pair them with pins based on proximity
Operations
1. **getLocationAsync() : void** - asks for permission for the user's location and then sets initialRegion with the user's initial coordinates if granted 2. **doDelete(pin: Pin) : void** - removes the pin from pins and updates the UI to display the map without the map 3. **render() : React.Component** - renders the Map with the pins displayed and with a route between the pins that correspond to attractions

Pin
Attributes
1. **id : int** - unique identifier for the pin for database purposes 2. **tripId : int** - unique identifier for the selected trip 3. **userId : int** - unique identifier for the user 4. **coords : Object** - contains the latitude and longitude of the pin's location 5. **title : string** - title for the pin 6. **description : string** - description for the pin 7. **photoUrl : string** - url of the photo associated with the pin

Operations
1. **constructor(obj) : Pin** - Creates a pin object from the JSON response from the database 2. **toJSON() : Object** - Creates a JSON map from the Pin object to store into the database 3. **storePin() : void** - Saves a pin to the database 4. **fetchPins(tripId : int) : void** - Returns all of the pins for a given trip from the database

AddPinPage
Attributes
1. **userId : int** - unique identifier for the user 2. **coords : Object** - contains the latitude and longitude of the pin's location 3. **title : string** - title for the pin being added 4. **description : string** - description for the pin being added 5. **photoUrl : string** - url of the photo associated with the pin with respect to proximity 6. **titleError : boolean** - true if there is an error with the user's entry for the title of the pin being added 7. **descriptionError : boolean** - true if there is an error with the user's entry for the description of the pin being added 8. **pins : [Pin]** - consists of first pins generated from attractions and then pins the user created 9. **photos : [Photo]** - consists of photos user has uploaded for this trip so that we can pair them with pins based on proximity
Operations
1. **getImage() : string** - checks all photos to see if there is one within 5 miles of the pin being added and if there is then the url of that photo is returned 2. **async submitPin() : void** - submits a pin if there are no errors in the title and description and if there is no other pin within a 5 mile radius of the photo 3. **render() : React.Component** - displays the UI of the AddPinPage with text inputs for the title and description and a button to submit the pin

c. Journal

Journal
Attributes
1. **id : int** - unique identifier for the pin for database purposes 2. **tripId : int** - unique identifier for the selected trip 3. **userId : int** - unique identifier for the user 4. **title: string** - title of journal entry 5. **note: string** - note of journal entry 6. **locations: [string]** - location tags inputted in journal entry 7. **url: string -** journal card cover photo 8. **date: string -** date journal entry was written
Operations
1. **toJSON(): object** - creates a JSON map from the Journal object to store into the database 2. **storeJournal(): void** - saves journal to database 3. **fetchJournals(tripId : int): [Journal]** - returns all of the journals from a given trip from the database 4. **deleteJournal(docId: int): void** - deletes specified journal entry from database 5. **updateJournal(docId: int, newTitle: string, newNote: string, newLocations: [string], newUrl: string): void** - updates specified journal entry in the database

JournalPage
Attributes
1. **journals: [Journal]** - journal entries fetched from the database
Operations
1. **arrayToObjects([string]): [Object]** - creates objects out of array to display existing location tags when user edits journal 2. **getChartData(): [Object]** - provides data to display on the journal commit data 3. **getCount(date: string): int** - gets count of journals written on a specified date 4. **render() : React.Component** - displays the UI of the main JournalPage with cards for existing journal entries

AddJournalPage
Attributes
2. **editJournal: boolean** - identifies if user is attempting edit an existing journal entry 3. **tagsSelected: [string]** - autofill options for location tags 4. **title: string** - already existing title if user is trying to edit journal entry 5. **note: string** - already existing note if user is trying to edit journal entry
Operations
1. **createNote(userId: int, tripId: int, title: string, note: string, locations: [Object], url: string): void** - creates a journal to save to the database based on the user input day 2. **editNote(dbId: int, title: string, note: string, locations: [Object], url: string): void** - 3. **backToJournal (): void** - provide inputs to change journal based on user's new inputs 4. **extractTitle(): [string]** - gets title from all map pins and saved attractions to use as location tags 5. **extractUrl(): [string]** - gets urls from all map pins and saved attractions to use as journal cover photos 6. **onCustomTagCreated(userInput: string): void** - allows for the addition of a non-autofilled location tag 7. **journalDate(): string** - provides date journal was written 8. **handleAddition(locations: [string]): void** - handles addition of a location tag based on user input 9. **handleDelete(index: int): void** - handles deletion of a location tag if a user requests to delete a tag 10. **render() : React.Component** - displays the UI of the AddJournalPage, which presents the user with options to add location tags, a title, and note to their journal entry

d. Calendar / Daily Summary

CalendarPage
Attributes
1. **tripID : string** - representing current trip identification 2. **startDay : Javascript Date Object** - the date the trip starts 3. **endDay : Javascript Date Object** - the date the trip ends 4. **selectedDay : Javascript Date Object** - the currently selected day in the calendar 5. **options : [string]** - conditions to display currently selected date 6. **error : object -** catches error when fetching photos 7. **weatherCondition : string** - current weather condition retrieved from weather API 8. **temperature : int** - current temperature retrieved from weather API 9. **city : string** - city nearest to user where there is weather data available 10. **latitude : float** - user's current latitude location 11. **longitude : float** l- user's current longitude location 12. **locationResult: string** - notifies if permission has been granted for location 13. **photos : [images]** - photos of the current day selected 14. **loading : boolean** - if the photos are loading to be displayed 15. **journals : [journals]** - journals of the current selected day
Operations
1. **getLocationAsync() : void** - gets the user's location and updates the longitude and latitude 2. **fetchWeather() : void** - retrieves information from weather API and sets the temperature, weatherCondition, and city attributes 3. **updateSelectedDay() : void** - updates the selected day of the calendar 4. **fetchJournals() : void** - fetches the journals of the selected day and updates the journals array field 5. **fetchPhotos() : void** - fetches the photos of the selected day and updates the photos array field 6. **toTimeStamp(year: int,month: int,day: int,hour: int,minute: int,second: int) : integer** - returns the timestamp of the given inputs in milliseconds 7. **render():** CalendarPage - returns the CalendarPage, a React.Component instance

e. Budget

Budget
Attributes
1. **amount: integer** - represents the budget that the user has dedicated to their trip 2. **id: string** - stores the id for the budget 3. **userId: string** - stores the id of the user logged in 4. **tripId: string** - stores the id of the trip that the user has made a budget for
Operations
5. **toJSON(): Budget** - a Budget object is made with the attributes above 6. **storeBudget(): void** - stores a reference to the budget in Firebase 7. **fetchBudget(tripId: string): integer** - retrieves the budget for the trip

BudgetItem
Attributes
1. **amount: integer** - represents the amount of the purchase made 2. **type: string** - represents the type of purchase made 3. **id: string** - stores the id for the purchase 4. **userId: string** - stores the id of the user logged in 5. **tripId: string** - stores the id of the trip that the user has made a purchase for
Operations
1. **toJSON(): Budget** - a BudgetItem object is made with the attributes above 2. **storeBudget(): void** - stores a reference to the purchase in Firebase 3. **fetchBudget(tripId: string): [BudgetItem]** - retrieves the array of purchases for this trip

AddBudget
Attributes
1. **budgetval: integer** - represents the budget for the user's trip
Operations
1. **createBudget(): void** - creates a Budget object for the trip 2. **viewBudget(): void** - allows the user to return to the BudgetPage to view the budget 3. **render(): AddBudget** - returns AddBudget, React.Component instance

AddBudgetItem
Attributes
1. **budgetval: integer** - represents the purchase value for the user's purchase 2. **typebudget: string** - represents the type of purchase for the user's purchase
Operations
1. **createBudgetItem(): void** - creates a BudgetItem object for the budget and for the trip 2. **viewBudgetItem(): void** - allows the user to return to the BudgetPage to view the budget and the purchase history 3. **render(): AddBudgetItem** - returns AddBudget,Item React.Component instance

BudgetPage
Attributes
1. **budget: integer** - represents the budget that the user has dedicated to their trip 2. **budgetitems: array** - contains the list of purchases that have been input by the user 3. **perDay: integer -** calculated by the system to give an estimate of how much the user should spend per day to remain within the budget 4. **showBudgetButton: boolean** - this will show the add new budget button if there is no budget 5. **showItemButton: boolean** - this will show the add new purchase if there is a budget already
Operations
6. **render(): BudgetPage** - returns the BudgetPage, React.Component instance

7. **checkBudgetPerDay(): void** - sets the approximate spending per day
8. **checkBudget(): void** - if there is an overall budget, the budget button is replaced with a purchase button, or else the budget button remains
9. **checkBudgetItem(): void** - if there is a budget, then the budget items are set in the state

f. Attractions

SuggestionsController
Attributes
1. **Screens:array** - array of screen objects that contain different components to display 2. **Homestack:object** - Stack navigator object for navigation
Operations
1. **createStackNavigtor():** Creates a new instance of stack navigator 2. **createAppContainer: void** - Creates an app container to display stack navigator

Category
Attributes
1. **Categories:array** - Array of names of different categories 2. **Attractions:object** - Array of attraction objects with different information
Operations
1. **getLocation() async:** Retrieves the current location of the user 2. **pressHandler(): void** - Navigates to attractions page when category clicked 3. **toSavedAttractions(): void -** Navigates to the saved attractions page

SavedAttractions
Attributes
4. **savedAttractions:array** - Array of saved attractions
Operations

1. **alertHandler(): void** - Warns user of deletion of object
2. **deleteHandler(item): void** - Deletes object from saved attractions

Retrieve
Attributes
1. **name:String** - Name of business 2. **price:String** - Price of attractions 3. **rating:String** - Rating of business 4. **address:String** - Address of business 5. **opening_hours:Array**- Opening hours of business 6. **saved:Boolean**- Boolean to determine if item was saved or not 7. **id:String** - Place id of business from API 8. **buttonText:String** - Displays ‘save’ or ‘saved’ on attraction item 9. **photoRef:String** - String the references photos of saved attractions 10. **coords:String** - Coordinates of business 11. **tripID:String** - TripID retrieved from Google Firebase 12. **review:array**- Array of review objects that contain review information on businesses 13. **Types: Array**- Categories to describe business 14. **Weekday:Array**- Displays opening hours through week 15. **number:String** - Phone Number of business 16. **Photos:Array**- Array of photo references associated with business
Operations
1. **storeAttaction(): void** - Stores attraction object to Google Firebase 2. **getInformation(loc, tripID): promise** - Calls Places API with current location coordinates and the tripID, returns array of attraction objects with associated information 3. **fetchAttaction(): void -** Returns attractions from Google Firebase 4. **getDetails(): void -** Calls places api for more information on specific business 5. **addSavedItems(): void** - Add saved items to array stored locally on application 6. **setSavedItems(): void** - Set saved items to array stored locally on application 7. **getSavedItems(): array** - Return saved items to array stored locally on application 8. **setSavedState(): void** - Save state of all attraction items to array stored locally on application 9. **getSavedState():array**- Return state of all attraction items to array stored locally on application

Attractions
Attributes
1. **sortP:Boolean**- Boolean determine if price is sorted in ascending or descending order 2. **sortR:Boolean**- Boolean determine if Rating is sorted in ascending or descending order 3. **Attractions:Array** - Array of attractions objects
Operations
1. + saveItem(item): void - Saves select item locally and to firebase **2. + sortPrice(): void -** Sorts price both ascending and descending **3. +sortRating(): void -** Sorts rating both ascending and descending **4. +priceCompare(a,b): void -** Comparator used for sorting price **5. +revPriceCompare(a,b): void -** Comparator used for sorting price in reverse **6. +ratingCompare(a,b): void -** Comparator used for sorting rating **7. +revRatingCompare(a,b): void -** Comparator used for sorting rating in reverse **8. +convertToDollars(price): String -** Converts price number to dollar signs **9. +toInformationWindow() : void -** Navigates to information window associated with attraction

Information Window
Attributes
1. -place_id:String - Id passed from attraction **2. -name: String -** name passed from attraction **3. -rating: String -** rating passed from attraction **4. -price: String -** price passed from attraction **5. -address: String -** address passed from attraction **6. -Loaded: Boolean -** Boolean to determine if component is loaded or not **7. -details: array -** Array of details object with additional information on selected business
Operations
1. + _renderItem: void - Renders each photo in a horizontal scrollview **2. + content(): view -** Renders component view after associated information is loaded into details array **3. + loading(): void -** Renders a loading wheel when waiting for information to be loaded

B. Object Constraint Language (OCL)

a. Photos

PhotosPage Contract
Class: PhotosPage **Relevant Operations**: getPhotos() : void, render() : void, uploadPhoto(photo) : void **Relevant Attributes:** trip: Trip, libraryPermission: boolean, loading: false, photos: [Photo]
Invariants
1. There must always be a reference to the trip object: **context** MapPage **inv:** self.trip **not** null 2. The user must be logged in: **context** firebase.auth(), **inv:** firebase.auth().currentUser **not** NULL
Preconditions
1. You can only load photos on device if the system has been granted library permissions. **context**: MediaLibrary, **pre:** MediaLibrary.requestPermissionsAsync() 2. You can only view a photo if it has been processed and uploaded. **context:** PhotosPage, **pre**: photo.uploaded is TRUE
Postconditions
1. The photo's tab shows a grid of uploaded photos and device photos.

b. Map

MapPage Contract
Class: MapPage **Relevant Operations**: doDelete() : void, render() : void, addPin() : void **Relevant Attributes:** trip : Trip, pins : array, selectedMarker : Pin, hasLocationPermissions : boolean, mapRegion : Obj
Invariants
1. There must always be a reference to the trip object: **context** MapPage **inv:** self.trip **not** null

Preconditions
1. You can only delete a pin if there is a pin and a selected pin: **context** MapPage::doDelete() : void **pre:** self.pins->size() > 0 **and** self.selectedMarker **not** null 2. You can only render the map if permission for location is granted: **context** MapPage::render() : void **pre:** self.hasLocationPermissions = true 3. You can only add a pin when the map has rendered: **context** MapPage::addPin() : void **pre:** self.mapRegion **not** null
Postconditions
1. The size of the pins array is one smaller after deletion: **context** MapPage::doDelete() : void **post**: self.pins→size() = self.pins→size()@pre - 1

AddPinPage Contract
Class: AddPinPage **Relevant Operations:** async submitPin() : void, getImage() : void, getDistance(coords1 : Obj, coords2 : Obj) : int, render() : void **Relevant Attributes:** userId : int, tripId : int, pins : array, photos : array, descriptionError : boolean, titleError : boolean
Invariants
1. There must be a reference to the tripId: **context** AddPinPage **inv:** self.tripId **not** null 2. There must be a reference to the userId: **context** AddPinPage **inv:** self.userId **not** null
Preconditions
1. Cannot add pins in the same place: **context** AddPinPage::submitPin() : void **pre:** self.pins->forAll(pin | AddPinPage::getDistance(pin.coords, pinToAdd.coords) > 5) 2. Photos array cannot be null when getImage() is called: **context** AddPinPage::getImage() : void **pre:** self.photos **not** null 3. There must be no errors in the title and description when the submit button is pressed: **context** AddPinPage::submitPin() : void **pre:** self.descriptionError = false **and** self.titleError = false
Postconditions
1. The size of the pins array is one larger after pin is submitted: **context** AddPinPage::submitPin() : void **post:** self.pins→size() = self.pins→size()@pre + 1

c. Journal

AddJournalPage Contract
Class: AddJournalPage **Relevant Operations**: createNote(): void, editNote(): void, handleAddition(): void, handleDelete(): void **Relevant Attributes:** trip: Trip, tagsSelected: [string], title: string, note: string, editJournal: boolean
Invariants
1. There must be a reference to the tripId: **context** AddJournalPage **inv:** self.tripId **not** null 2. There must be a reference to the userId: **context** AddJournalPage **inv:** self.userId **not** null
Preconditions
1. A journal must already exist in order for one to be edited: **context:** AddJournalPage::editNote() **inv:** editJournal = true 2. A location tag must exist in order for it to be deleted: **context:** AddJournalPage::handleDelete() **inv:** tagsSelected.length > 0 3. There must be no errors in the title or note for a journal to be created: **context:** AddJournalPage **inv:** self.title and self.note not **null**
Postconditions
1. The size of the journals array is one larger after journal is submitted: context AddJournalPage::createNote() : void post: self.journals→size() = self.journals→ size()@pre + 1

JournalPage Contract
Class: JournalPage **Relevant Operations**: getChartData(): [Object], getCount(): [Object] **Relevant Attributes:** trip: Trip, journals: Journal
Invariants
1. There must be a reference to the tripId: **context** JournalPage **inv:** self.tripId **not** null

Preconditions
1. Journals must exist for user to get count of journals on specific dates **context** JournalPage::getCount() **inv:** journals not **null** 2. Journals must exist for user for entry data to be provided to the commit chart **context** JournalPage::getChartData() **inv:** journals not **null**
Postconditions
1. The JournalPage shows a list of journal cards when all journals have been fetched from the database

d. Calendar/ Daily Summary

CalendarPage Contract
Class: CalendarPage **Relevant Operations:** async getLocationAsync() : void, fetchWeather() : void, fetchJournals(j) : void, fetchPhotos() : void, toTimeStamp(year : int, month : int, day : int, hour: int, minute : int, second : int) : int **Relevant Attributes:** tripId : int, photos : array, journals : array, startDay : Javascript Date Object, endDay : Javascript Date Object, selectedDay: Javascript Date Object, latitude : int, longitude : int, temperature : int, weatherCondition : string, city : string
Invariants
1. There must be a reference to the tripId: **context** CalendarPage **inv:** self.tripId **not** null
Preconditions
1. There must be a reference to startDay: **context** CalendarPage **inv:** self.startDay **not** null 2. There must be a reference to endDay: **context** CalendarPage **inv:** self.endDay **not** null 3. The selected day is the startDay: **context** CalendarPage **inv:** self.startDay == self.selectedDay 4. There must be a reference to the latitude and longitude to fetch the weather: **context** CalendarPage::fetchWeather : void **pre:** self.latitude **not** null and self.longitude **not** null
Postconditions
1. There are references to weatherCondition, temperature, and city after weather is fetched: **context** CalendarPage::fetchWeather() : void **post**: self.weatherCondition **not** null **and** self.temperature **not** null **and** self.city **not** null

2. If journals and photos have been added to the selected day, the journals array and photos array are larger than when they started: **context** CalendarPage::updateSelectedDay() : void **post:** self.journals → size() > self.journals→ size()@pre **and** self.photos → size() > self.photos→ size()@pre

e. Budget

AddBudget Contract
Class: AddBudget **Relevant Operations**: createBudget(): void **Relevant Attributes:** budgetval: integer
Invariants
1. There must be a reference to the tripId: **context** AddBudget **inv:** self.tripId **not** null 2. There must be a reference to the userId: **context** AddBudget **inv:** self.userId **not** null
Preconditions
1. The value for the budget must be a valid positive numeric value: **context** AddBudget **inv:** self.budget **not** negative
Postconditions
1. There is now a value for the budget for the user's trip: **context:** AddBudget::createBudget(): void **bold:** self.budget **not** null

AddBudgetItem Contract
Class: AddBudgetItem **Relevant Operations**: createBudgetItem(): void **Relevant Attributes:** budgetval: integer, typebudget: string
Invariants
1. There must be a reference to the tripId: **context** AddBudgetItem **inv:** self.tripId **not** null 2. There must be a reference to the userId: **context** AddBudgetItem **inv:** self.userId **not** null

Preconditions
1. There must be a budget that already exists: **context** BudgetPage **inv:** self.budget **not** null
Postconditions
1. The size of the purchases array is one larger after a new purchase is created: **context** AddBudgetItem::createBudgetItem() : void **post:** self.budgetitems->size() = self.budgetitems >size()@pre + 1

BudgetPage Contract
Class: BudgetPage **Relevant Operations**: checkBudgetPerDay(): void **Relevant Attributes:** budget: integer, perDay: integer, showBudgetButton: boolean, showItemButton: boolean
Invariants
1. There must be a reference to the tripId: **context** BudgetPage **inv:** self.tripId **not** null 2. There must be a reference to the userId: **context** BudgetPage **inv:** self.userId **not** null
Preconditions
1. There must be an overall budget for there to be a purchase logged: **context** BudgetPage **inv:** self.budget **not** null 2. There must be an overall budget for there to be a budget goal per day: **context** BudgetPage **inv:** self.budget **not** null 3. The add purchase button must show for there to be a budget goal per day or for a purchase to be logged: **context** BudgetPage **inv:** self.showBudgetButton: false, self.showItemButton: true
Postconditions
1. The BudgetPage shows all the purchases (if any) and the budget (if there is one) for the user's trip and based on this information, the budget per day is displayed: **context** BudgetPage::checkBudgetPerDay(): void **post:** self.perDay **not** null

f. Attractions

Category Contract
Class: Category **Relevant Operations**: getLocation() : void **Relevant Attributes:** attractions: [object]
Invariants
1. There must be a reference to the tripId: **context** CategoryPage **inv:** self.tripId **not** null
Preconditions
1. There must be a location returned in order to retrieve information: **context** Category::getLocation() **inv:** location **not** null
Postconditions
1. The attractions array is populated with the 20 most popular destinations in the user's current area **context** Category::setSavedState():void **post**: self.attractions → size() = self.attractions → size@pre + 20

Attractions Contract
Class: Category **Relevant Operations**: saveItem(item):void, sortPrice():void, sortRating():void, convertToDollars(price):String **Relevant Attributes:** attractions:[object]
Invariants
1. There must be a reference to the tripId: **context** CategoryPage **inv:** self.tripId **not** null
Preconditions
1. There must be a price information in the attractions array to sort: **context** Attraction::sortPrice() **inv:** self.attractions **not** null 2. There must be a rating information in the attractions array to sort: **context** Attraction::sortRating() **inv:** self.attractions **not** null 3. There must be an attraction information in order to save an item: **context** Attraction::saveItem(item) **inv** self.attractions **not** null

4. To save an item, an item can not already be saved **context** Attraction:saveItem(item) **inv** self.attractions.saved = false
Postconditions
1. The attractions array is sorted by price 2. The attractions array is sorted by rating 3. The selected attraction is saved: **context** Attraction::saveItem(item) **post: context** Retrieve.savedItemsArray → size() = **context** Retrieve.savedItemsArray → size()@pre + 1 4. The saved attractions array is unchanged: **context** Attraction::saveItem(item) **post: context** Retrieve.savedItemsArray → size() = **context** Retrieve.savedItemsArray → size()@pre

InformationWindow Contract
Class: Category **Relevant Operations**: getDetails(item):promise, _renderItem(): view, content(): view **Relevant Attributes:** Details:[object], loaded:boolean
Invariants
1. There must be a reference to the tripId: **context** InfomrationWindowPage **inv:** self.tripId **not** null
Preconditions
1. In order to display information getDetails() must not return null: **context** Retrieve::getDetails(item) **inv:** self.Details **not** null 2. In order to display photos the Details array must be populated with a photo reference array: **context** InformationWindow::_renderItem() **inv** self.Details.photoReference **not** null 3. In order to render content to the screen the loaded attribute must be set to true after confirmation that getDetails() has finished retrieving information: **context** Retrieve::content() **inv** self.loaded = true
Postconditions
1. The information is displayed to the screen 2. The photos are rendered in a horizontal scrollview 3. The content is rendered to the page

SavedAttractions Contract
Class: Category **Relevant Operations**: deleteHandler(item): void **Relevant Attributes:** attractions: [object], savedItems: array
Invariants
1. There must be a reference to the tripId: **context** SavedAttractions **inv:** self.tripId **not** null
Preconditions
1. The item must exist for it to be deleted: **Context** SavedAttractions **inv**: self.attractions.id not **null** 2. The item must have an item id for it to be properly displayed in the flatlst: **context** SavedAttrations: **inv:** self.attractions.id **not** null
Postconditions
1. The item is deleted from the savedItems array: **context** SavedAttactions::deleteHandler() **post: context** Retrieve.savedItems → size = **context** Retrieve.savedItems → size@pre - 1

C. Traceability Matrix

1. System Sequence and Interaction Diagrams Relations

This traces the current Interaction Diagrams from the System Sequence Diagrams from Report 1.

Interaction Diagrams

a. **View Calendar/Daily Summary:** Our group decided to add this interaction diagram because we thought that viewing the calendar and daily summary was an important event in the succession of our application and needed to be mapped out in an interaction diagram
b. **View Daily Summary:** evolved from system-sequence diagram *ViewDailySummary*
c. **Add Journal Entry:** evolved from system-sequence diagram *AddJournalEntry*
d. **Add Photo:** evolved from system-sequence diagram *AddPhoto*
e. **View Map:** evolved from system-sequence diagram *ViewMap*

f. **View Budget Statistics:** evolved from system-sequence diagram *ViewBudget*. This specifically looks at how a user can view their budget statistics.
g. **View Budget Overview:** evolved from *ViewBudget* and *FindAttractions*. This details how the budget is viewed once an attraction is committed to the user's profile.

2. Classes from Domain Diagram Concepts

This lists the Domain Concepts from Report 1 and relates them to the classes and objects that are in our Class Diagram.

Domain Diagram Concepts

a. **Log-In Database:** became a part of the *Account* class. This change is because we realized having an account controller class will handle the login responsibilities defined in our domain concept.
b. **Trip Profile Database:** became a part of the *TripViewController* class. This controller coordinates with the Trip Profile Database to save information relating to the current trip.
c. **Journal Database:** became a part of the *JournalController* class.The JournalController can create, edit, or delete journal entries which are saved in the journal database.
d. **Google Maps API:** became a part of our our *MapsController* which uses Google Maps API to display a map of the user's nearby location
e. **Google Trips API:** The *SuggestionsController* uses the Google Trips API to access nearby attractions and events and display them to the user
f. **Budget Database:** The *BudgetController* class communicates with the budget database to create, log, save, and delete information related to a user's budget.
g. **Photos Database:** evolved into the *PhotoController* class which coordinates with the photo database to add and delete photos
h. **Account Controller:** became the *Account* class, handling all of the user's profile information
i. **Trip Profile Controller:** evolved into *TripController*
j. **Journal Controller:** evolved into *JournalController*
k. **Map Controller:** evolved into *MapController*
l. **Budget Controller:** evolved into *BudgetController*
m. **Suggestions Controller:** evolved into *SuggestionsController*
n. **Photos Controller:** evolved into *PhotosController*
o. **Calendar Controller:** evolved into *CalendarViewController*. The slight name change came from our group deciding that the controller is mainly focused on displaying information
p. **Daily View Controller:** evolved into *DailyViewController*
q. **Log-In View:** evolved in *LoginEntity* which can create or delete a user, or retrieve and update login information from the user. All this information will be displayed through the user interface to the user.

r. **Trip-Profile View:** evolved into *TripViewEntity*. Our group decided to include the trip-profile view in this object because the object has specific fields which will be displayed to the user
s. **Journal View:** evolves into *JournalEntity* which is an object that is created by the user. The journal object can then be displayed.
t. **Map View:** evolved into *PinEntry* which marks certain locations on the map. Our group changed the name to make the object more specific to pins, giving more clarity on its function in our program.
u. **Budget View:** evolved into *BudgetEntity*. Our group thought that the budget should be its own object to which the user can request to edit, add, delete, or view.
v. **Suggestions View:** evolved into *AttractionsEntity*. We altered the name to make it clear that it is possible attractions that we are suggesting to the user.
w. **Photos View:** evolved into *PhotosEntity* which can retrieve a photo to view, or add and delete a photo
x. **Calendar View:** evolved into *Calendar* which is mainly responsible for displaying the daily summary depending on which day is clicked
y. **Daily View:** evolved into *DailySummary* which displays all of the current information including budget, suggestions, journal entries, and photos on a specific day

When looking back at this section of report 2 it becomes evident that because we hadn't started building out our app yet our vision of how it would be structured was very different from how it ended up becoming. We ended up using all all of this basic architecture to setup the views and controllers for the app but we also built out several other classes to organize how the data of the app was stored such as Pin which stored fields relevant to the location of the pin, Budget which stores details about budget, Retrieve which stores information about selected attractions, Journal which stores information relevant to journal entries, and many more custom classes for the actual storage of information. Also there were several views renamed, each of them being named a page, some pages didn't need controllers as the implementation was done in the view itself, and other minor changes were made too.

9. System Architecture + System Design

D. Architectural Style

It is imperative to realize that we can utilize different architectural styles to optimize different areas on our system.

1. Layered Architecture

The form of layered architecture we will be using is Model-View Controller (MVC), which is further explained below. As a general overview, we will encompass the following layers:

1. **Presentation (UI) Layer**
   Visible to the user, and present information concerning the application. Additionally, encryption and decryption are performed in this layer.
2. **Application (Service) Layer**
   Used to synchronize communication within the application, primarily to translate between the UI and domain layer.
3. **Business Logic (Domain) Layer**
   Represents business model, which includes object-oriented design (set of classes).
4. **Data Access (Persistence) Layer**
   Isolates upper layers from the database. Benefits include easier migration to other storage engines, easier modification of the database layer if needed in the future, etc.
5. **Database**
   Stores all information concerning data for the application. This is expanded upon in ‘Client-Server Access’.

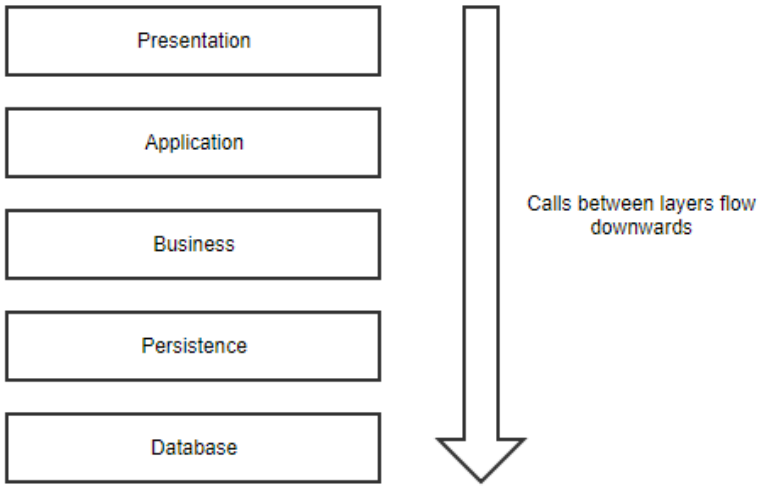


Figure 1

2. Model-View Controller (MVC)

This is a form of 'Layered Architecture'. This architectural style provides for the interaction between the (1) model, (2) view, (3) controller. A description of each component, and how it pertains to the Trip Diary system, is provided below:

6. **Model**
   Contains data, state, and application logic. The model will provide content to the view so the user is able to interact with it through the display. Or, the model can directly interface with the controller to change the view. This lays right above the database.
7. **View**
   The view component provides a direct output, and interfaces with the display so the UI is available to the user. In the case of the Trip Diary, the 'view' component would display the following for each division of our application:
   a. **Calendar** → Provide a calendar view of the dates of the user's trip. Additionally, the user will be able to see their daily summary
   b. **Daily Summary** → Provide a view of a summary of uploads from a specific date on the trip
   c. **Journal** → Provide a view of journal entries within that day, which includes previous entries and the option to create a new one
   d. **Map** → Provide a view of the map, which includes locations the user has visited within that day. The user additionally has the option to add locations to the map.
   e. **Photos** → Provide a view of photos the user has uploaded for that specific date, with the option to upload additional pictures.
   f. **Budget** → Provide a view of the budget, which has a user-friendly interface to display charts and statistics concerning spending trends for a specific date. Additionally, the user has the option to input additional budgets.
   g. **Suggestions** → Provide a view of suggestions for places to visit pertaining to the user's trip
8. **Controller**
   Used to coordinate with the model to provide change based on user input. The controller will instantiate change within the model, and allow for the view to change display. The controller will be controlled by an input gesture on the app, considering it is a smartphone application.

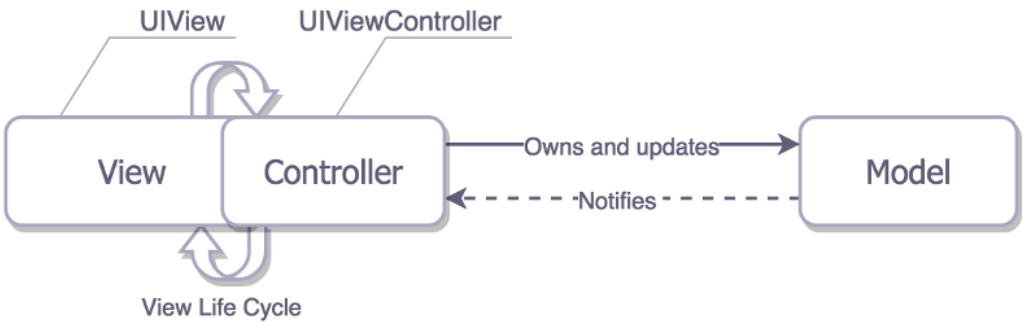


Figure 2

3. Client-Server Access

We are utilizing a database, which is abstracted through ‘Firebase’, which is a part of the Google Cloud Platform. This database (server) will be able to fulfill requests that are sent by the ‘client’, or the user for the system. Database requests will be used for a multiple for reasons within the ‘Trip Diary’ application. Below are select examples:

1. **Log-In** → User will request to access their account, and the database must verify the credentials.
2. **Journal** → User will request old journal entries, and the database must retrieve previous journal entries for the user to view/edit. Additionally, the user may want to store a journal entry, and the database must fulfil that request.
3. **Budget** → User will request to see their spending habits and inputted budgets, and the database must provide the needed financial data. Additionally, the user may want to submit a budget or expenses, and the database must store the needed information.
4. **Photos** → User will request to upload and access photos, and the database must store/retrieve them.
5. **Map** → User will request to see locations they have visited or may ask to store visited locations. The database must fulfil this by either sending previous locations or by storing new locations.

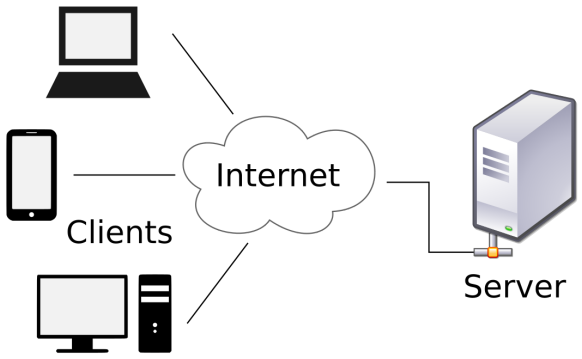
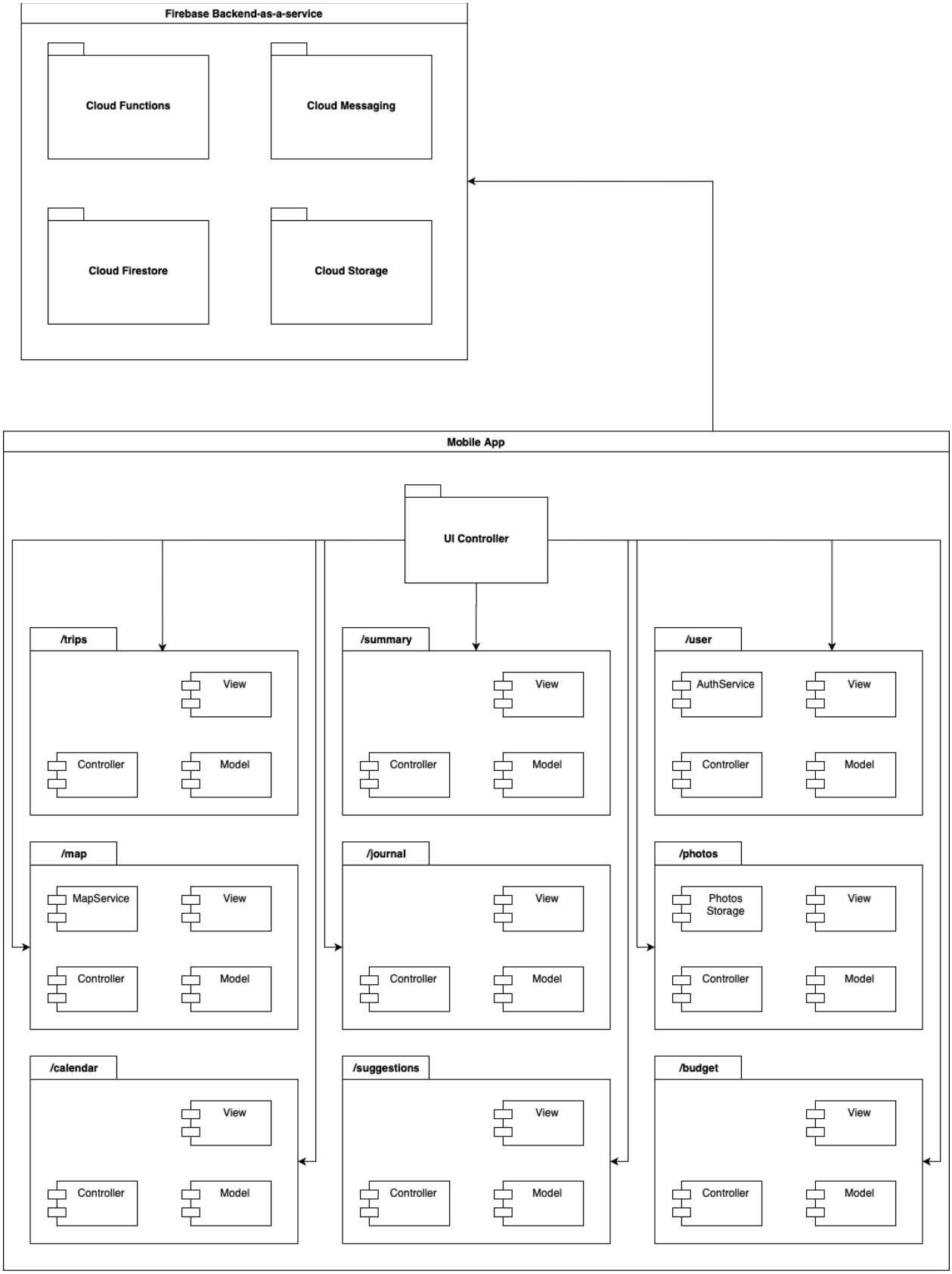


Figure 3

E. Identifying Subsystems

1. Firebase Backend-as-a-service: backend service which includes all of the core functionality, allowing us to focus on development of the platform itself from it's available software development kits.
    - **a.** Cloud Firestore: Firebase provides database service which offers features such as access control layers, caching, and snapshot connections to the database for real time changes.
    - **b.** Cloud Storage: Firebase provided storage buckets used for storing user files and data, such as photos.
    - **c.** Cloud Messaging: Firebase provides notification service which allows us to send push notifications to users on a time basis, or based on changes to our data.
    - **d.** Cloud Functions: Backend functions delegated into microservices which can be scheduled to run on demand.
2. Mobile App: This piece is the user-facing piece of our applications. All of the user interactions are facilitated through the main UI Controller. The UI controller maintains access control, then passes off the functionality to each of the sub-components.


Firebase Backend-as-a-service
Cloud Functions
Cloud Messaging
Cloud Firestore
Cloud Storage
Mobile App
UI Controller
/trips
View
Controller
Model
/summary
View
Controller
Model
/user
AuthService
View
Controller
Model
/map
MapService
View
Controller
Model
/journal
View
Controller
Model
/photos
Photos Storage
View
Controller
Model
/calendar
View
Controller
Model
/suggestions
View
Controller
Model
/budget
View
Controller
Model

F. Mapping Subsystems to Hardware

1. Firebase services help us avoid the hassle of managing the physical hardware running the services, but we've selected their central US datacenter location to host our backend services specified above. The resources are allocated on demand and scale according to the traffic our mobile app faces.
2. Our client application runs on both android and ios mobile operating systems and provides the user with the interface to carry out the functions of our product.

G. Persistent Data Storage

1. Our application persists user data using the Firebase Cloud Firestore service to save the state of the application, and provide a storage location. Cloud firestore employs a nosql database and provides us with a javascript library which handles authentication, caching, and provides additional features such as snapshot listeners for real-time updates to the user data.
2. We store the following collections in our database:
    a. **/users** - documents containing user data
    b. **/trips** - documents containing properties of a trip, which can be owned by a user
    c. **/trips/{id}/locations** - documents containing location points for a user for a given trip
    d. **/trips/{id}/journals** - documents containing journal entries created by the user for a trip
    e. **/trips/{id}/photos** - documents containing references to photos taken by the user for a trip
    f. **/trips/{id}/calendar** - documents containing event data stored in the user's trip calendar
    g. **/trips/{id}/expenses** - documents containing expense data for a trip

H. Network Protocol

1. HTTP

Considering Trip Diary development will be placed on a mobile application, it is common to use HTTP (Hypertext Transfer Protocol). With this protocol (which is used commonly by Android applications), we are able to support TLS (Transport Layer Security) and connection pooling. This benefit allows the application to use a common database for multiple users, and allows for the efficient utilization of resources (i.e. storage). As provided below, the client would be the Trip Diary Application, and the server could potentially be the database server.

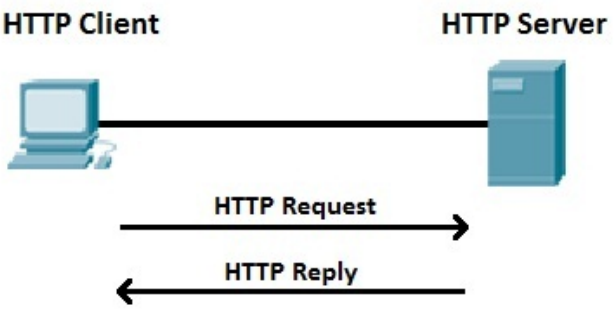


Figure 3

F. Global Control Flow

1. Execution Orderliness

Our system is an *event-driven* system, meaning that it will wait in a loop for an event to occur so that it can respond accordingly. In this case, an event is initiated by the user through the user interface of the mobile application. For example, the system will wait for the user to enter a correct username and password, and click the login button before taking the user to their account page. Users can trigger different actions depending on how they interact with the system, so there is no linear path of events. For example, if the user wants to add an entry to a specific day of their trip, they will click on an item they want to add, yet they do not need to go through this series of steps if they want to add a photo, or add their budget. In this way, the user can choose how they want to interact with the application, which will provide them with a better user experience.

2. Time Dependency

Since our system is an event-response type, so there is little concern for real time. Therefore, there will not be any timers in our system. Our system will wait in a loop for an event to occur which is triggered by the user.

3. Concurrency

Ideally, our system will utilize multiple threads to execute commands simultaneously. Multithreading is the ability of the central processing unit to generate multiple threads of execution which can be executed at the same time.

In our particular mobile application, the following objects will have their own threads to execute concurrently, GPSClass and the main user interface thread. The main user interface thread will handle all other executions of our program. The GPSClass is always tracking the location of the user and requires its own thread of execution. The attractions/photos features represent the intersection between the GPSCLass thread and the main user interface thread, therefore, the threads are synchronized because they must share resources. The attractions thread uses the GPSLocation to find and suggest new attractions based on the user location. The photo

feature stores the GPS location of different photos taken. Otherwise, when clicking to view a day, the main thread of execution will have to sequentially go through the users photos, suggestions, journal entries, budget, and map to display them. (This may take too much time, so using multiple threads to display this information is a possibility). When adding any trip or entry, everything will be handled through the main thread.

G. Hardware Requirements

Our React Native application requires a screen display, a communication network and storage. Because we are using a firebase as a database, this offloads the amount of required local hard disk storage.

Minimum Resolution: 720x1280p
Minimum Communication Network: 70 kb/s
Minimum Hard Disk Space: 1gb

10. Algorithms & Data Structures

A. Algorithms

Simple statistical algorithms will be used to develop Budget Statistics, a way for users to view how they are budgeting. Specifically, we will be using mathematical equations to develop the mean and mode, which are measures of central tendency. We will also be using range, a measure of variability, so that the user knows the general range of their spending habits. These algorithms are simple math equations that will be used to calculate these variables.

B. Data Structures

The main deciding factor in the data structures we used was flexibility. Most of the information we store in data structures must be accessed and displayed in their entirety on a page (all the photos from an array of photos, all the pins on a map, etc). This means that having O(1) search and access is not important, so an arraylist will suffice for most purposes as it has O(1) insert which is the only efficient operation we need and is fairly flexible.

1. **Calendar**
   The Calendar will have two variables which it uses to generate the page. It will take the start date and end date from the trip object to display all of the days of the trip. When a date is selected or clicked, the calendar page will query each of the databases (photos, journal, etc.) to retrieve all the information pertaining to that day to display in the Daily Summary and store the information received in an array.
2. **Journal**
   Each journal will be an object which will hold different factors of a journal entry. No additional data structures are needed for the journal object as it is stored in the database (bypasses need for data structure).
3. **Map**
   Each map will have 2 variables, one for the x coordinate and the other for the y coordinate. This will dictate the middle of the city where the trip takes place. Each map object will also have an **arraylist** of pin objects which the user has placed to pinpoint locations of interest they have visited and marked down on their trip. An arraylist allows for any additional pins to be added without issue.
4. **Photo**
   Each photo has three variables which store the size of the photo, the type of the photo, and the datetime the photo was taken. No additional data structures are

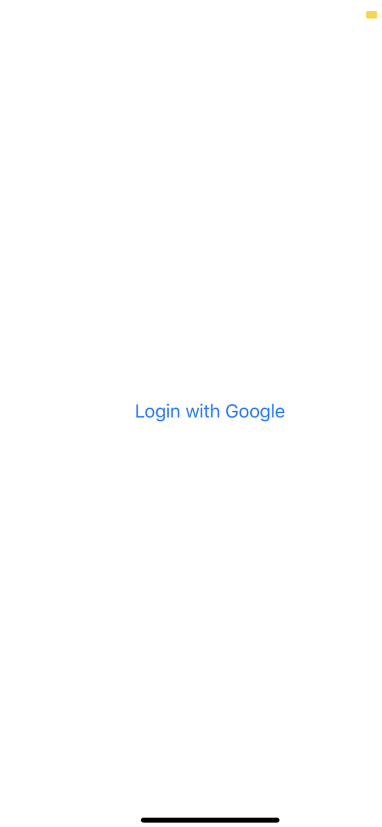
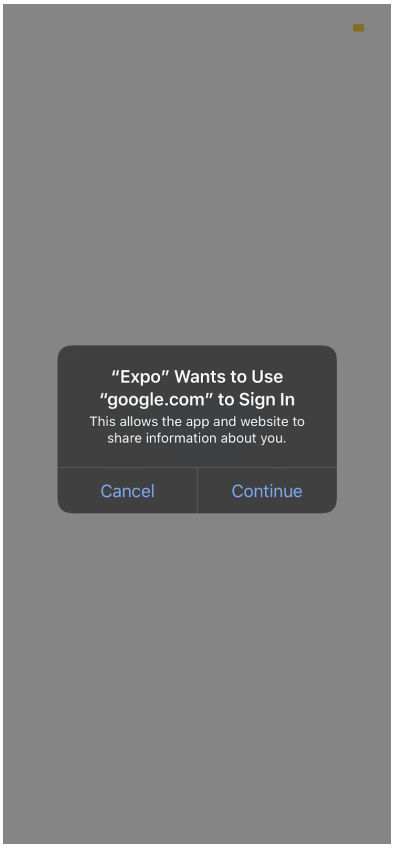
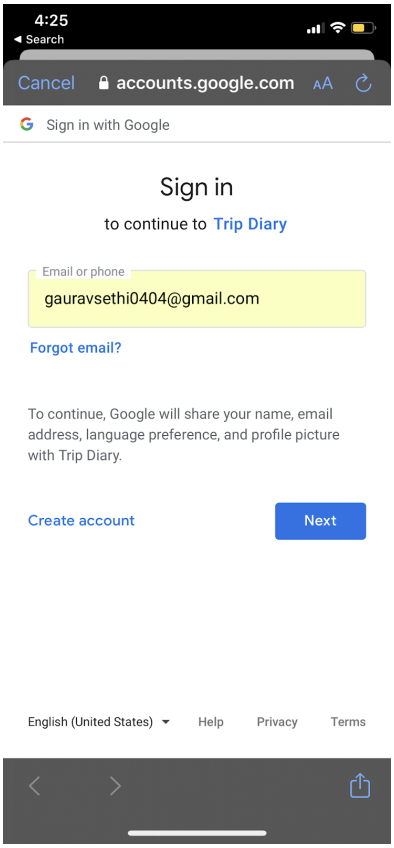
needed for a photo object as the physical photo is stored in the database (bypasses need for data structure).

5. **Pin**
   Each pin will have a longitude and latitude variable which will store the exact location of where the pin was placed. Pin objects will have no other data structures.
6. **Trip**
   Each trip will be an object which will hold different factors of the trip. No additional data structures are needed for the trip object as it is stored in the database (bypasses need for data structure).
7. **Budget**
   The budget statistics will use two **arraylists** to store tips and tricks or encouragement to continue saving, that are presented at random for the user. If the user is low on their budget, it will use the list that represents tips and tricks and if the user has not used much of their budget, then the user will be told to keep on saving. Another **array** will be used to keep track of the categories that the user spends most on. This list will be updated as the user continues to spend more in their trip. The list will include the amount spent and the corresponding category and will be organized by most spent to least spent. This will give the user an idea of what area of purchases they spend most on.
8. **Attraction**
   Each attraction that is retrieved by the Google Places API will be stored in an attraction object. Each of these attractions will be added to an **arraylist** of attractions, depending on category, since we do not know how many attractions the user will be requesting.

11. User Interface Design + Implementation

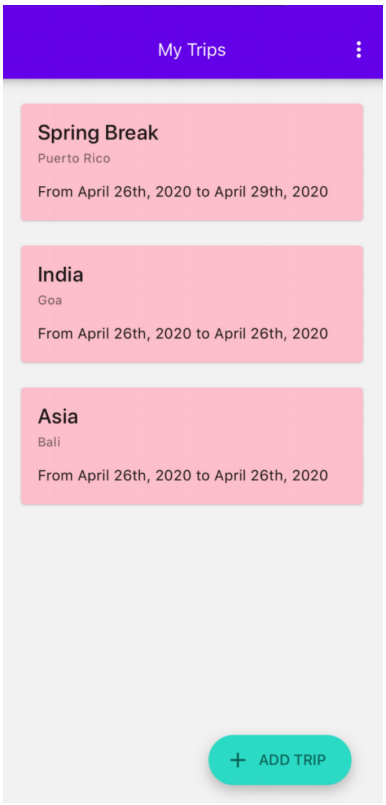
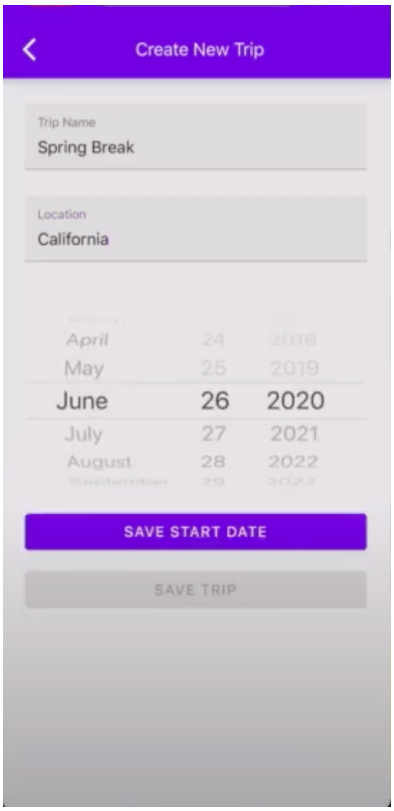
A. Final Design + User Effort Estimation

UC-1: Create Account/Login

<table>
<tr><th colspan="3">Create Account/Login</th></tr>
<tr>
<td> 
</td>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>The app is loaded with this screen to allow the user to login to the application through a google account. All facilitated by the google login API and firebase, the user will be able to login to his google account and enter the application.</td>
<td>The user can authorize the app to use google to login to the app by clicking continue or decline access by clicking cancel.</td>
<td>The user is brought to a google login page where he can enter his email and password and once authorized will be taken to the application.</td>
</tr>
<tr><td colspan="3">Feature Notes:</td></tr>
</table>

1. Provided through the Google platform, TripDiary is able to be accessible (on Android and iOS) through a user's already existing Google account. 2. Individual user's data is protected and only accessible with authorized google login
User Click Rate: 3 clicks 1. User clicks on app 2. User authorizes google login 3. User enters in google information and clicks next to complete login

UC-2: Create Trip

Create Trip	
	
Once the user successfully logs in they are brought to their my trips page. Through separate blocks, the user is able to distinguish between different trips.	If a user wishes to start a new 'Trip Diary', they must interact with the 'Create Trip' page by clicking the 'Add Trip' button. Through this, the user is able to create a 'trip name' and add a 'location' as well as a starting date.

	Through use of a back button, they are able to return to the previous page.

Feature Notes:

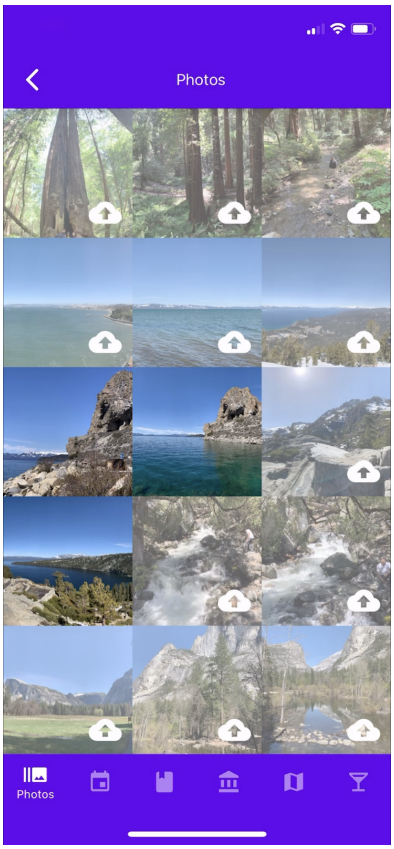
1. By clicking on a trip the user will be led to a calendar/daily summary page where they can view a basic overview of their trip.

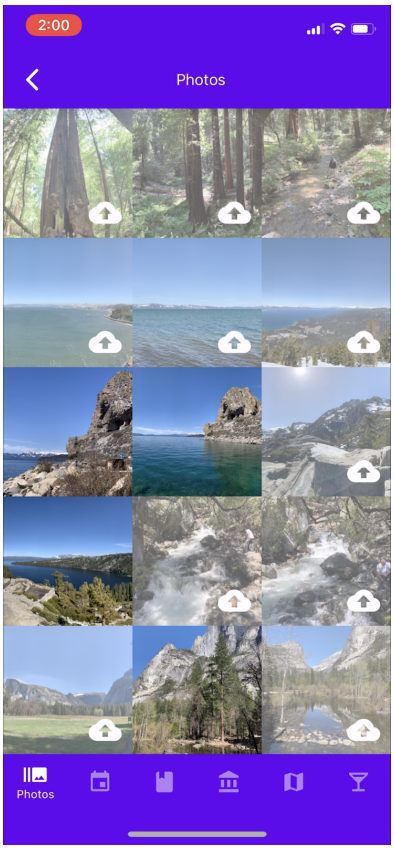
User Click Rate: 4 clicks

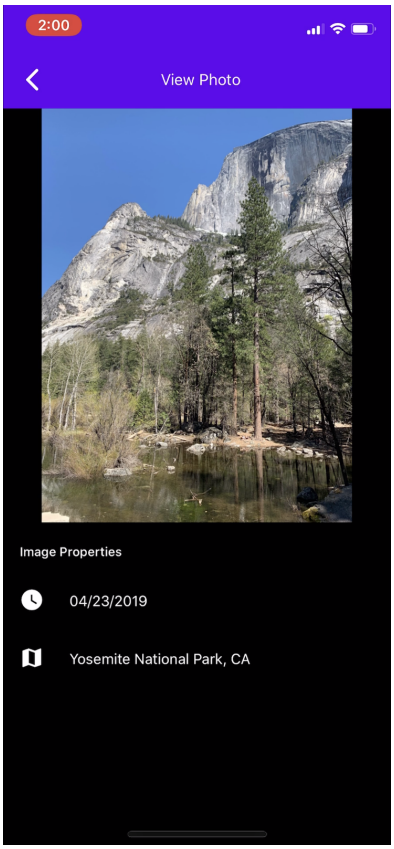
1. Click on add trip
2. Enter trip title and location
3. Select start date and click save start date
4. Click save trip

UC-3: Photos

Upload Photo

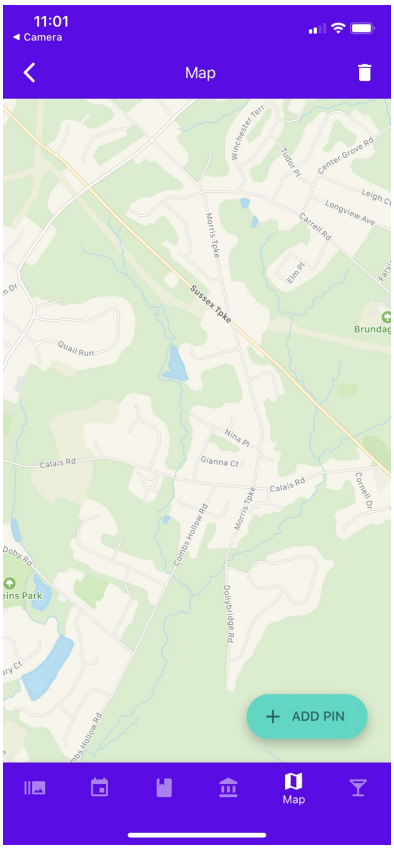
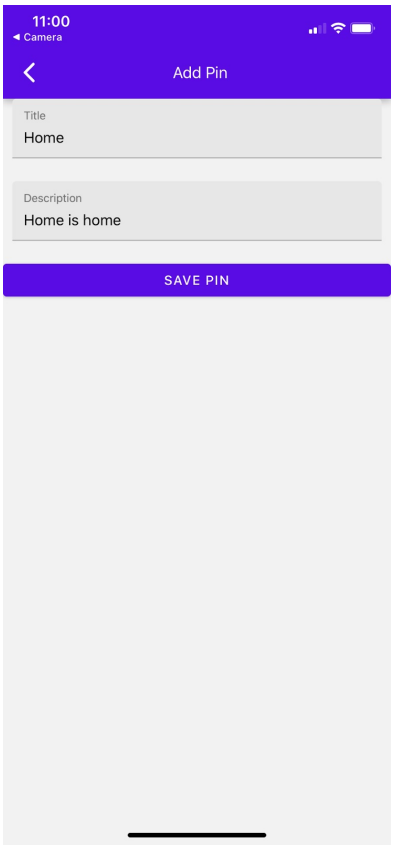
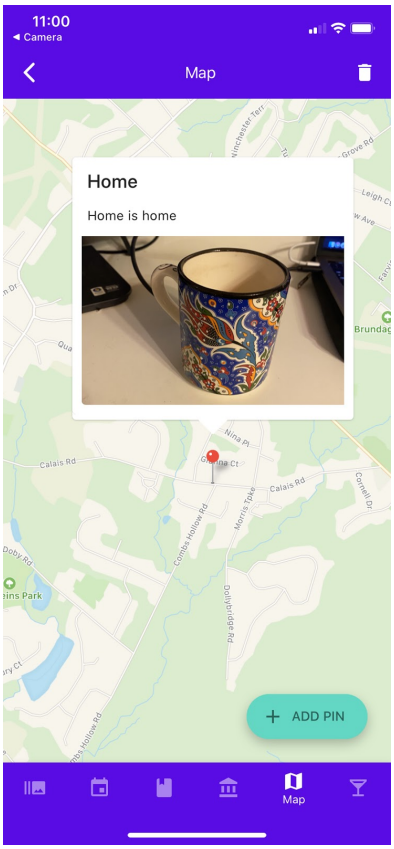




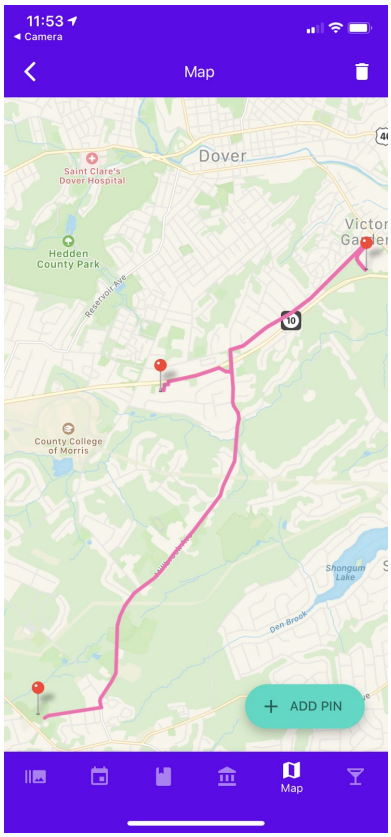
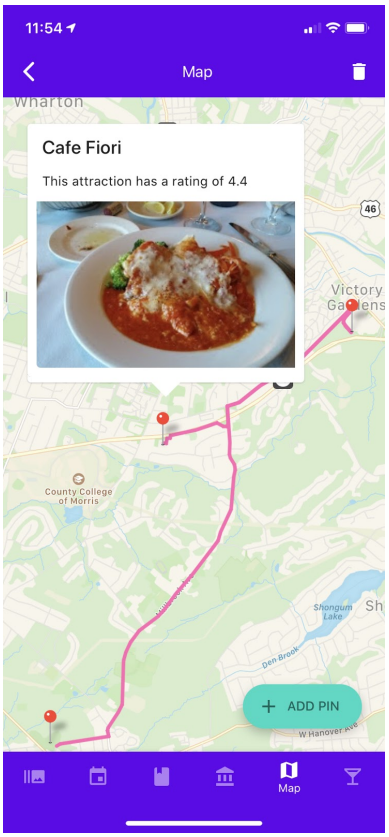


On first load of the photos tab for the trip, the application fetches all photos on the device that are taken within the duration of the trip specified. It then fetches the photos uploaded to the storage bucket and cross checks to see if they are also on the device. Then, it sorts the photos by date taken, and displays them in grid format.	In the previous screenshot, you'll notice that the center picture in the last row was marked as "not uploaded". After tapping on it, the user can initiate an explicit action to process and upload the picture. Once it has completed, the opacity changes for the photo tile to represent a successful upload.	Once we tap on the same photo again, after it has been uploaded, it allows the user to open a full screen view of the photo. This view displays the photo, the time it was taken, and the relative location.
Feature Notes: 3. When a user opens this tab, it requires the user to grant the system with access to the photo library permission. Without this permission, the system will only load previously uploaded photos. 4. The system relies on the ubiquitous EXIF format to identify which photos are within the trip.		
User Click Rate: 2 clicks 4. User scrolls to find the picture they want to upload. 5. User clicks on photo to initiate processing and uploading the file.		

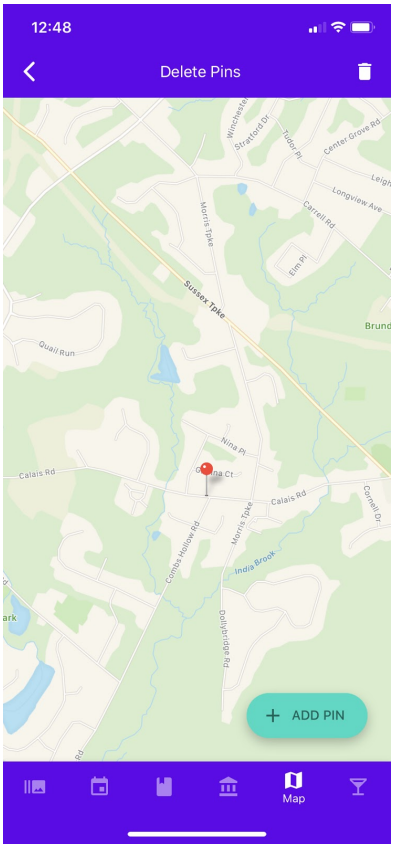
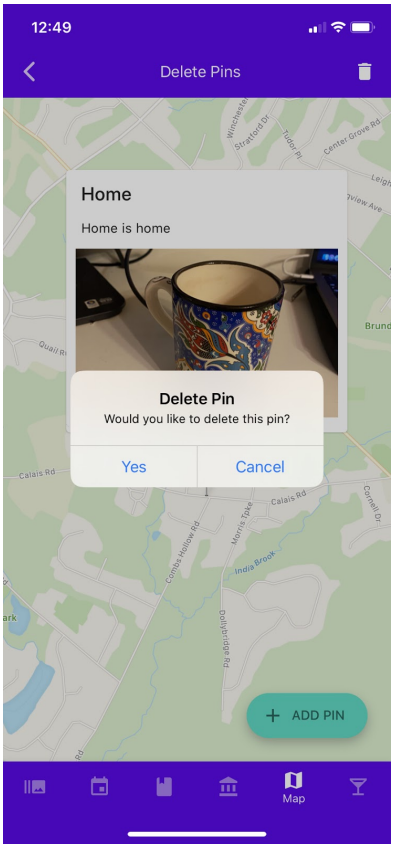
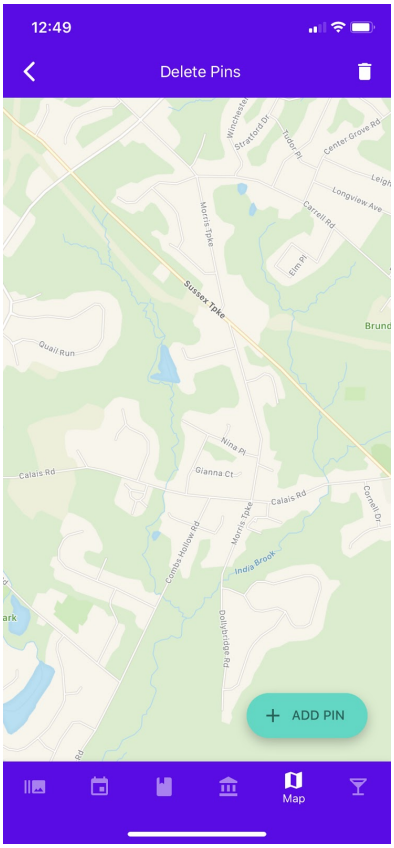
UC-4: ViewMap

<table>
<tr><th colspan="3">Add User Pin</th></tr>
<tr>
<td> 
</td>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>On the main map view the user can do the following actions to add a custom user-generated pin:
1. Press on a desired location on the map
2. The system will then update the coordinates to where the press was made
3. User Presses on the Add Pin button</td>
<td>In the add pin page view, the user is required to do the following to successfully submit the pin:
1. Enter a title
2. Enter a description
3. Press save pin</td>
<td>Once the save pin button is pressed, the user is navigated back to the location he/she left off of. Here, the user may do the following:
1. Select the pin, which will display a callout generated by the system
2. To hide the callout, the user may press anywhere on the map</td>
</tr>
<tr>
<td colspan="3">Feature Notes:
1. When the user navigates to the add pin page, the system will fetch all of the user’s photos on that trip and will automatically select the first photo it finds which is within a 5 mile radius of the location for the pin being added</td>
</tr>
</table>

2. When the ‘save pin’ button is pressed a query will be made to the database, storing the pin in a document format
User Click Rate: 7 clicks 1. Click on the map 2. Click on the add pin button 3. Click on ‘title’ to type in a title for the pin 4. Click on ‘description’ to type in a description for the pin 5. Click ‘save pin’ 6. Click on pin to view custom callout 7. Click off of pin to hide the callout

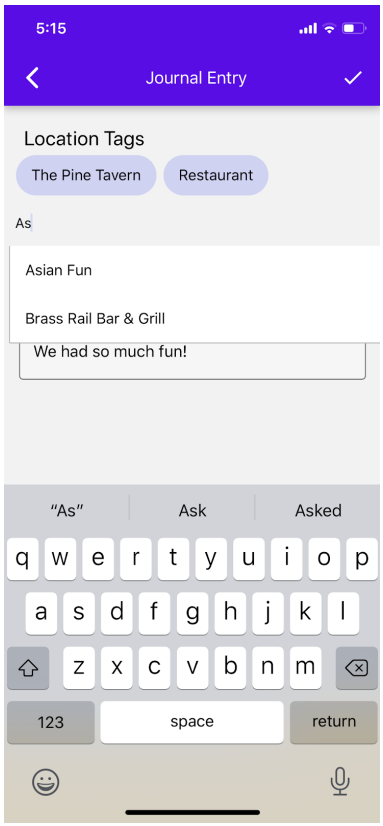
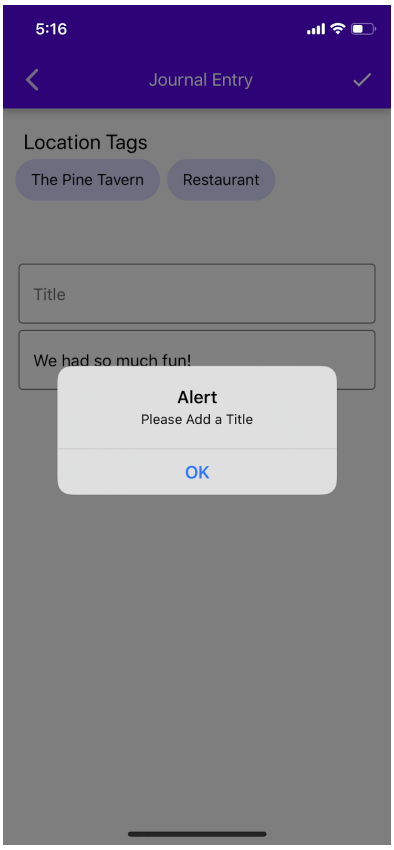
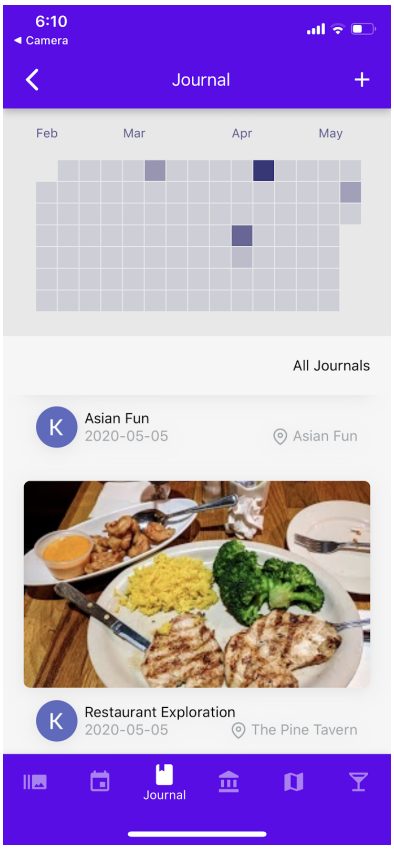
View Routes	
	
The user is displayed a route between attractions rendered as pins on the map and can do the following: 1. Scroll between routes to see which attractions can be visited at a time 2. Press on pins to view details of the	If the user presses on the pin, then he/she can do the following: 1. Press anywhere on the map to hide the callout

attraction	
Feature Notes: The callout for pins that correspond to attractions are different from pins made by the user in the following ways: 1. The description tells you the rating of the attraction 2. The photo is fetched from the Google Photos API, which returns a photo corresponding to the name of the attraction	
User Click Rate: 3 clicks 1. User can scroll through the map between routes 2. User can press on a pin to view callout 3. User can press on the map to hide the callout	

Delete Pin		
		
If the user wants to delete a pin from the map, he/she can do the following:	Once the user has pressed on the pin, an alert comes up asking the user to confirm the	If the user deleted the pin, it is removed from the view. To return from the ‘delete pins’

1. Press on the delete icon in the header of the view 2. Press on the pin which you desire to delete	deletion or cancel it: 1. The user may click 'yes' in order to delete the pin 2. Alternate scenario: The user may click 'cancel' to cancel the deletion	mode, the user can do the following: 1. Press on the delete icon in the header of the view
Feature Notes: There is a difference in deleting user pins and auto-generated pins corresponding to attractions: 1. Deleting a user pin: The pin will be removed from the map permanently 2. Deleting an auto-generated pin: The pin will be removed from the map for the duration of one's usage of the map and a new route will appear, but when the user reloads the map, the attraction will appear again since it is still a saved attraction		
User Click Rate: 4 clicks worst case 1. Click delete icon 2. Click on the pin to delete 3. Click 'Yes' to delete 4. Alternate scenario: click 'Cancel' to cancel deletion 5. Click delete icon		

UC-8: AddJournal

<table>
<tr><th colspan="3">Add Journal Entry</th></tr>
<tr>
<td> 
</td>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>User wishes to add a journal by clicking the '+' button on the main page. They will be able to access the following features:
1. Autofilled location tags based on map pins added in the map feature or attractions saved in the attractions feature
2. Journal title
3. Journal entry</td>
<td>Alternate Scenario: The user has tried to submit without needed information. An error will presented for the following situations:
1. User is missing a title
2. User is missing a note
3. User is missing both a title and note</td>
<td>User views their journal entry on the main page. They will be presented with the following:
1. Journal card with their Google avatar, journal title, the date journal entry was written, and location from the location tags added
2. Updated journal commit chart, which will make squares darker vs. lighter depending on the count of journals on a</td>
</tr>
</table>

		specific day as compared with other dates

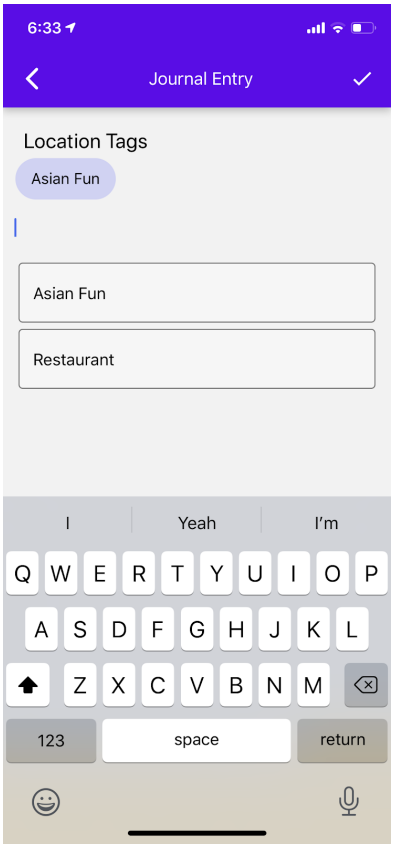
Feature Notes:

1. The journal card will automatically be given a picture depending on if the location tag is a:
    a. Saved attraction: the Google Trips image will be used as the card image
    b. Map pin: if there is a picture in the user's photos that is within 5 miles of the map pin location, then it will be added as the card image
    c. Neither: a default 'Pink Golden Gate Bridge' image will be shown
2. You are able to add *as many location tags as you would like*, but the first tag will be the one present on the journal card in the main view

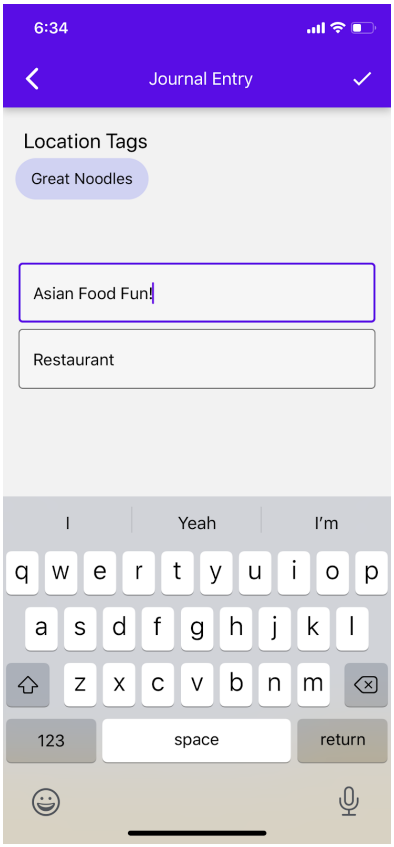
User Click Rate: (1) 6 clicks worst case (2) Varying keystrokes

1. Click '+' button on main page to be navigated to add journal entry
2. Click 'Location Tags' and type location tags in.
    a. User effort is reduced due to the autofill opportunity for locations within the system (map pins, saved attractions)
3. Click 'Journal Title' and type in desired title
4. Click 'Journal Note' and type in desired note
5. Alternate Scenario: Click out of alert to fill in missing information
6. Click 'check mark' and save journal entry

Edit Journal Entry

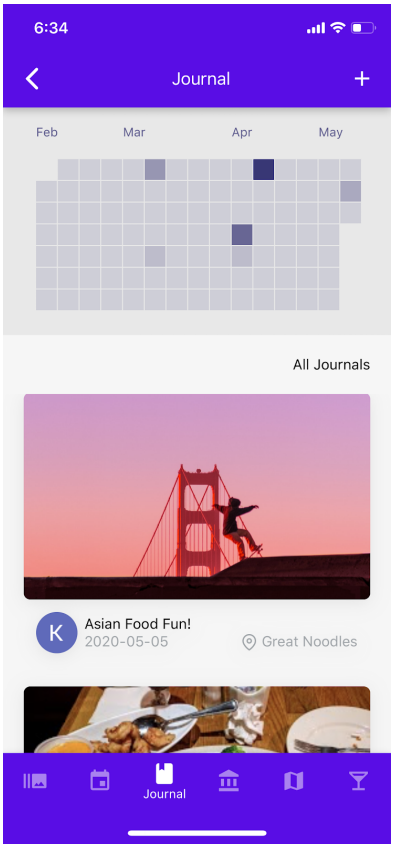


The user has chosen the 'Asian Fun' journal to be edited



The user has the option to change the following:

1. Location Tag
2. Title
3. Note

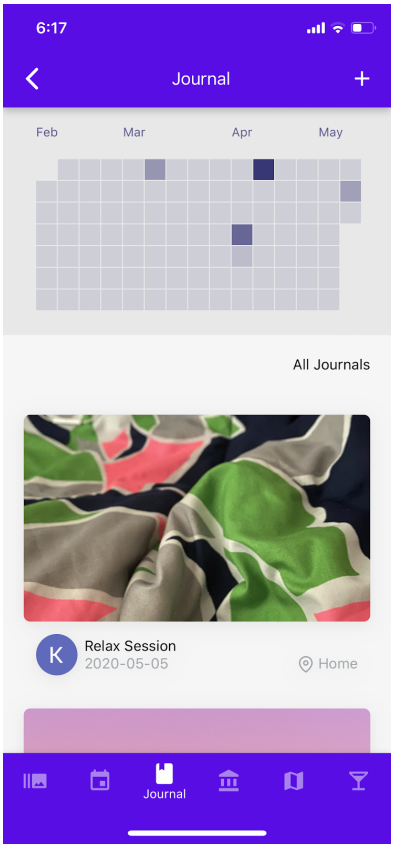


The main journal view presents a changed location, title, and image (due to the changed location tag)

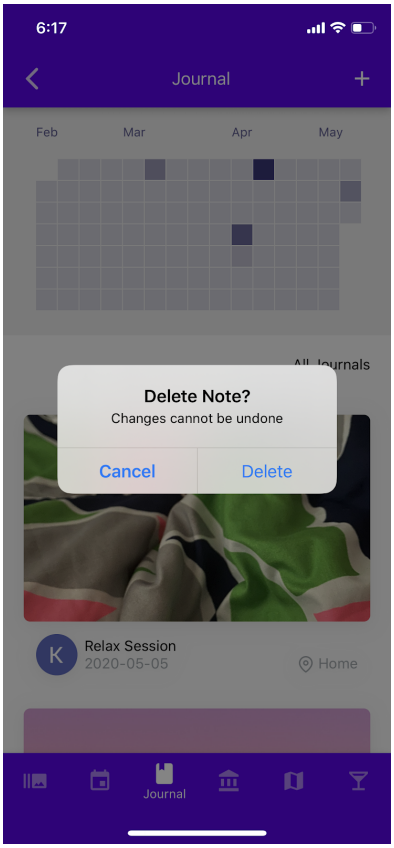
User Click Rate: (1) 6 clicks worst case (2) Varying keystrokes

1. Click journal card that needs to be edited
2. Click to change desired options: 3 clicks worst case
    a. Location tags: click an existing location tag to remove it from the list
    b. Journal title: click the title and type to edit
    c. Journal note: click the note and type to edit
3. Alternate Scenario: Click out of alert to fill in missing information
4. Click 'check mark' and save journal entry

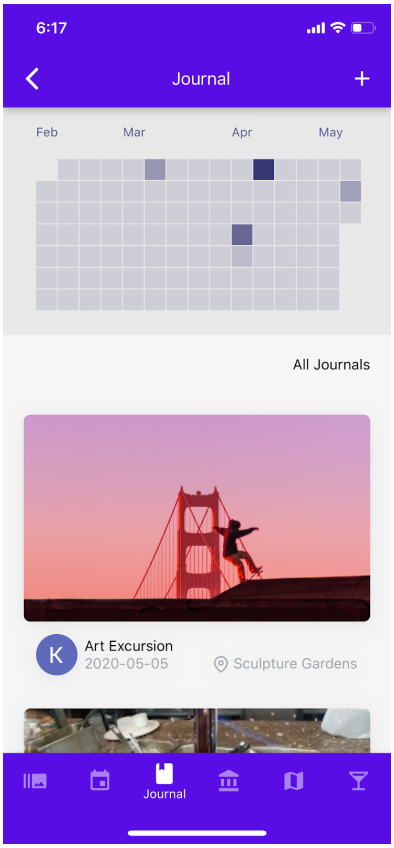
Delete Journal Entry



User is on the main page, and wishes to delete their ‘Home’ journal entry



1. User has long-pressed the journal card which they would like to delete
2. This prompts the ‘Delete Note’ alert



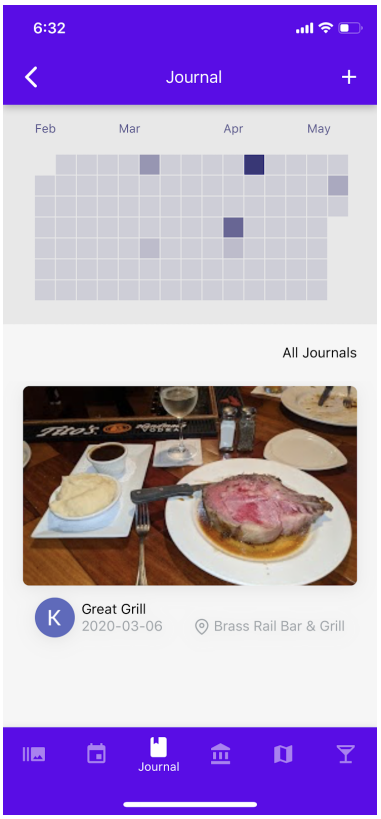
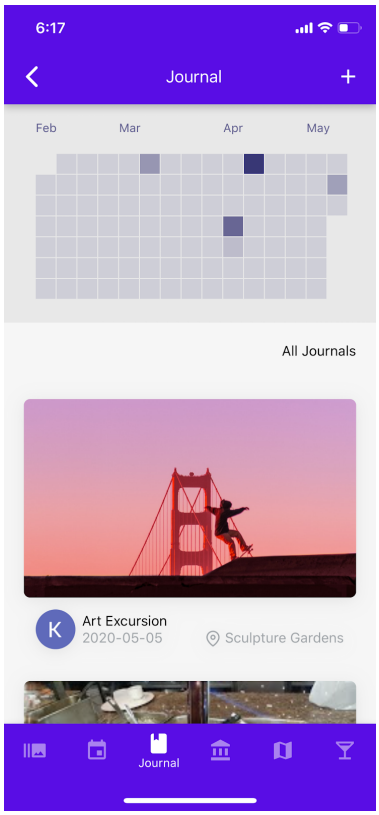
The journal card is no longer available on the main journal view

Feature Notes:

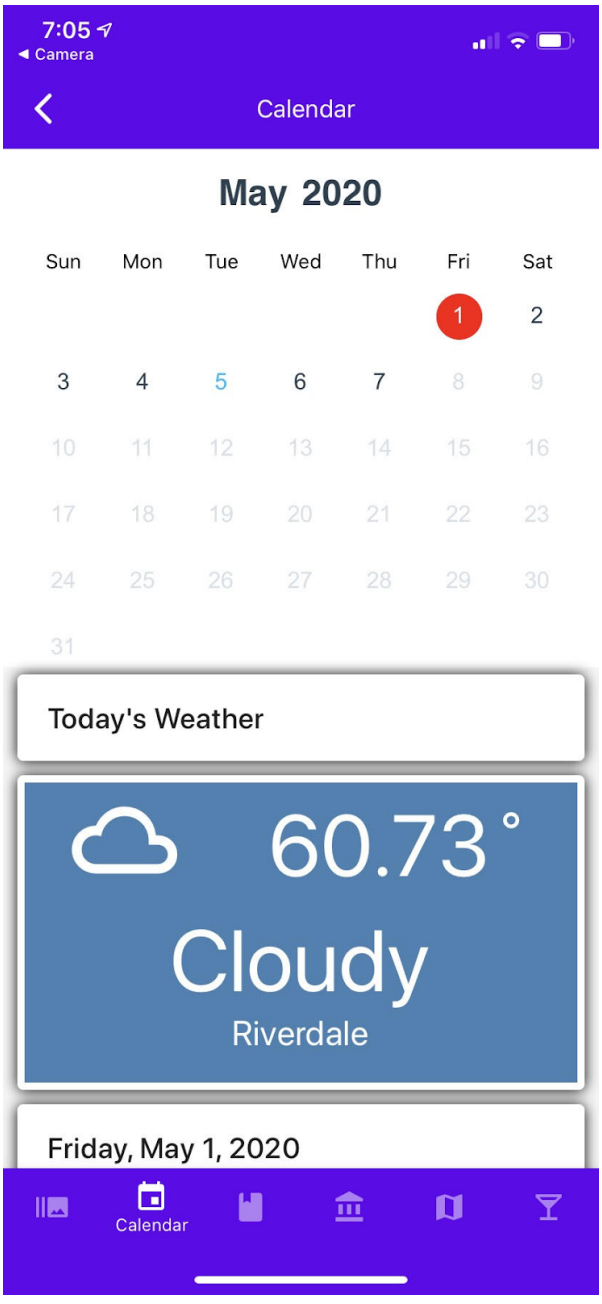
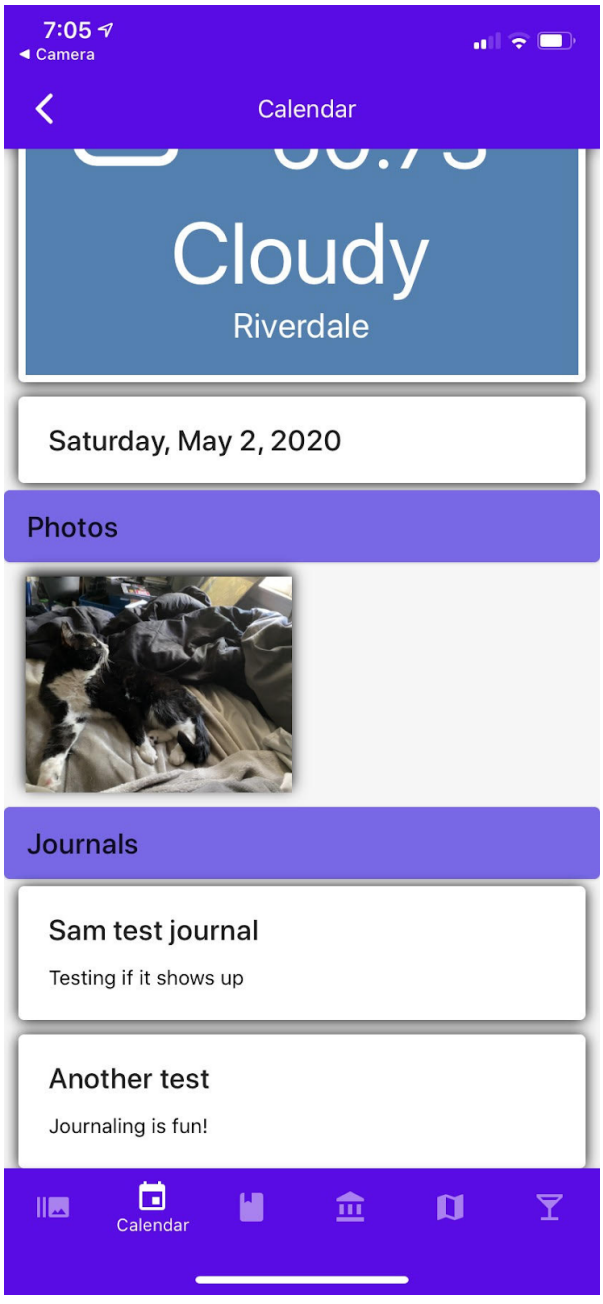
1. The picture on the ‘Relax Session’ card is an example of a user-added photo associated location tag. The image was selected because it was taken within a 5-mile radius of the map pin to which location tag refers to.
2. The picture on the ‘Art Excursion’ card is an example of a default image. A default image is used when the user adds a location tag not associated with a pin or an attraction.

User Click Rate: 2 clicks

1. Click for 3+ seconds journal card that needs to be deleted
2. Click ‘Delete’

Sort Journal Entries	
	
User has clicked 03-06-2020 square to sort and view all journals from that date only	User has clicked the ‘Reset’ button to be able to view all journal entries
User Click Rate: 2 clicks 1. Click on a journal commit chart square 2. Click ‘All Journals’ to reset	

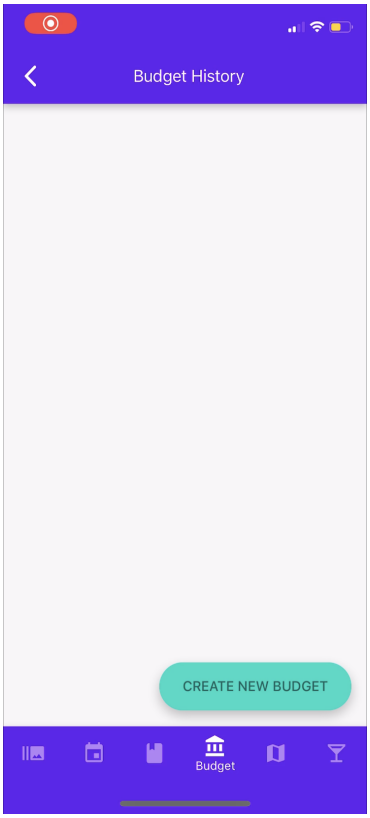
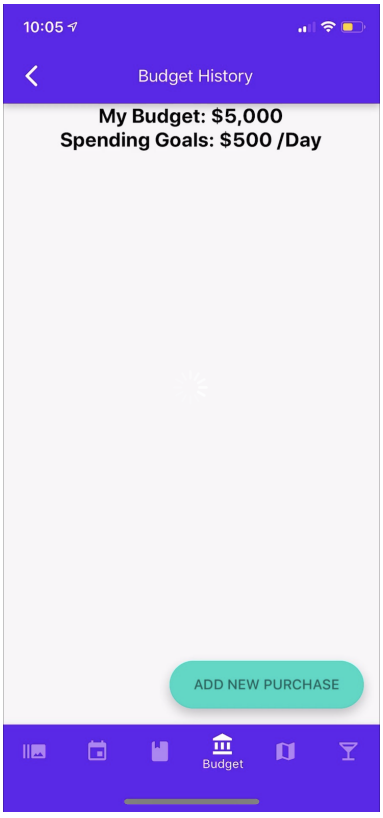
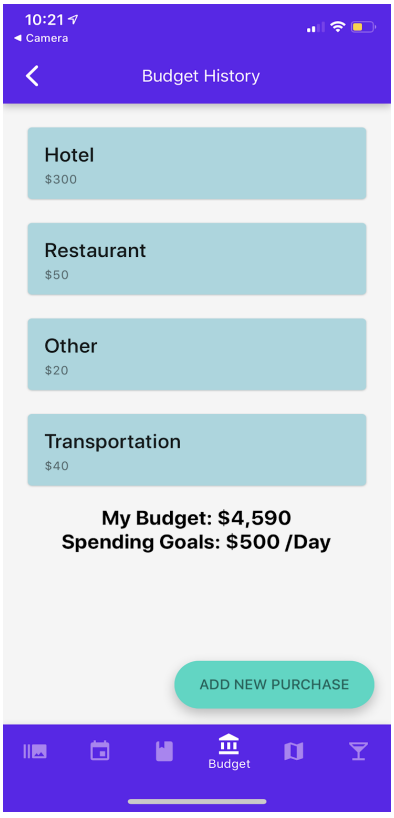
UC-7: ViewCalendar/DailySummary

Selecting a Day	
	
The user has selected a trip from the MyTrips page and has navigated to the Calendar tab through the toolbar at the bottom of the page	1. User selects a day of their trip by clicking on it within the calendar 2. The user can view all of the photos and journal entries created on that day

User Click Rate: 1 click (worst case)

1. User clicks on a day in the calendar

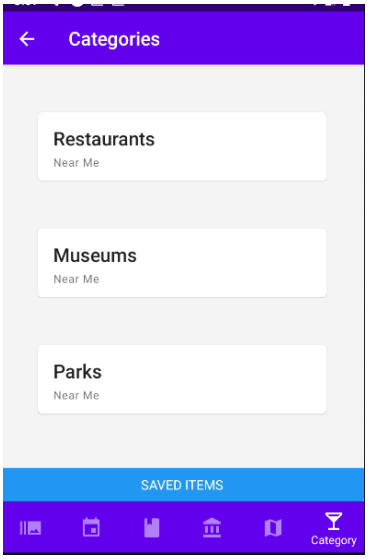
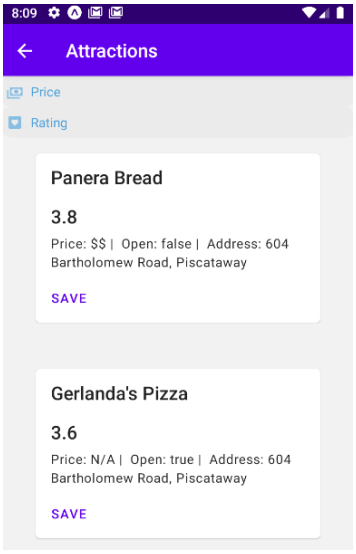
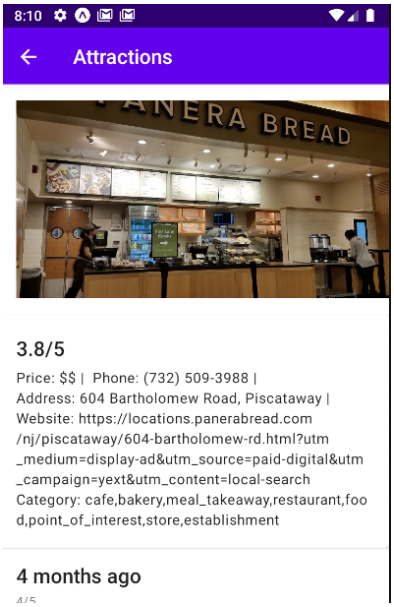
UC-5: Budget

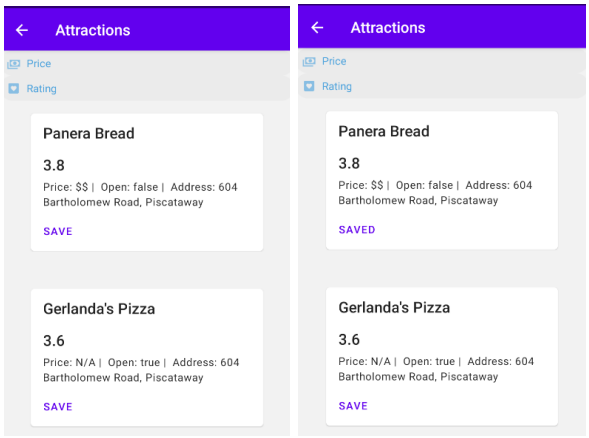
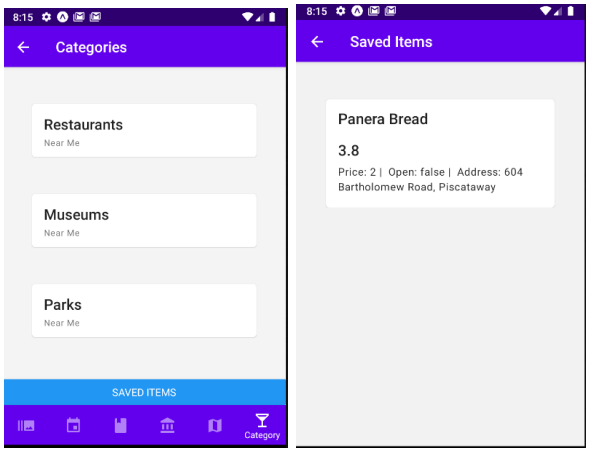
Viewing Budget Information		
		
1. User selected the budget tab 2. User presses to create new budget	1. User views new budget created and suggested spending 2. User presses to add new purchase	1. User views all the purchasing history as well as the subtractions in budget

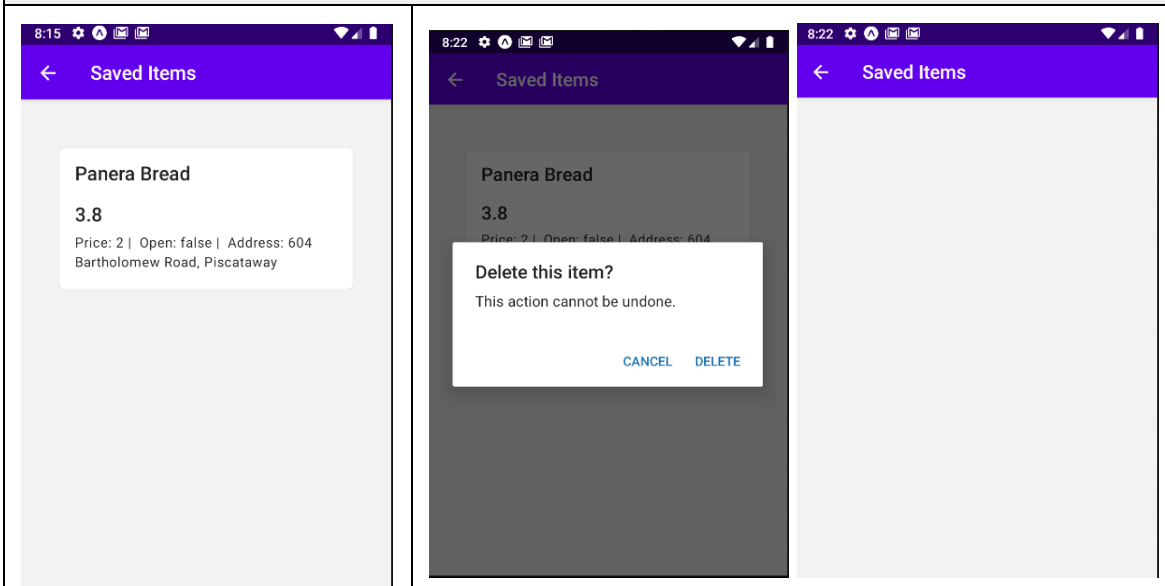
User Click Rate: 4 clicks - worst case

1. User clicks on create new budget
2. User clicks to create new budget
3. User clicks on add new purchase
4. User clicks to create new purchase

UC-6: Find Attractions

<table>
<tr><th colspan="3">Viewing Attraction Information</th></tr>
<tr>
<td> 
</td>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>1. The user has selected the attractions page using the attractions tab on the main navigation bar.
2. User selects category of attraction</td>
<td>3. Users select businesses that they are interested in, for example we can select panera bread.</td>
<td>4. Users are presented with additional business information such as multiple photos, user reviews, price, phone number, website, etc.</td>
</tr>
<tr>
<td colspan="3">User Click Rate: 3 clicks
5. User clicks on tab
6. User clicks on category
7. User clicks on business
8. Can view information</td>
</tr>
</table>

View Saved Attraction Information	
	
1. User clicks on the save button for Panera Bread. The text changes from 'save' → 'saved'	2. User clicks the back button in the upper navigation bar 3. User clicks on the saved items tab and can then view the attraction he or she just saved.
User Click Rate: 3 clicks 1. Save button 2. User clicks on back arrow 3. User clicks on save item tab	

<table>
<tr><th colspan="2">Delete Saved Attraction Information</th></tr>
<tr><td colspan="2"> 
</td></tr>
<tr><td>1. Long presses an attraction in the saved items page</td><td>2. Presented with a window with two options to cancel or delete. User presses delete button
3. User brought back to saved items page with attraction deleted
2. User presses cancel button (Alternative Scenario)
3. User brought back to saved item page with attraction not deleted (Alternative Scenario)</td></tr>
<tr><td colspan="2">User Click Rate: 2 clicks
1. User long presses an attraction
2. presses the delete button.
2. User presses cancel button</td></tr>
</table>

12. Design of Tests

A. Test Cases - Unit Testing

TC-1: Login

Test-case Identifier:	TC-1
Use Case Tested:	UC-1: Login
Pass/fail Criteria:	The test passes if the system is able to successfully retrieve the user from the database
Input Data:	Text

Test Procedure	Expected Result
Step 1: Type in a username and password which exist in the database	System renders the trip list view
Step 2: Type in a username and password which does not exist in the database	System sends an alert saying that the account does not exist and the user must create an account

TC-2: Create a Trip

Test-case Identifier:	TC-2
Use Case Tested:	UC-2: CreateTrip
Pass/fail Criteria:	The test passes if the user enters a valid title name
Input Data:	Text

Test Procedure	Expected Result
Step 1: Enter a blank title name and submit	Systems sends an alert saying title cannot be empty
Step 2: Type in a valid title and submit	System adds trip to list of user's trip and displays trip in view

TC-3: Add Pin of Current Location

Test-case Identifier:	TC-3
Use Case Tested:	UC-4: ViewMap
Pass/fail Criteria:	The test passes if the user adds a pin to his or her current location, with the pin not having any other pins in a 5 mile radius
Input Data:	User presses the ‘submit pin’ button

Test Procedure	Expected Result
Step 1: User presses add pin in a location where there already is a pin	System sends an alert saying there is already a pin here
Step 2: User adds a pin in a location where there are no other pins in a 25 foot radius	System adds the pin to the map and re-renders the map

TC-4: Submit Pin

Test-case Identifier:	TC-4
Use Case Tested:	UC-4: ViewMap
Pass/fail Criteria:	The test passes if the user submits a pin with a valid title and description
Input Data:	Text input

Test Procedure	Expected Result
Step 1: User submits a pin with an invalid title	System provides an alert that the title is invalid
Step 2: User submits a pin with an invalid description	System provides an alert that the description is invalid
Step 3: User submits a pin with a valid title and description	The pin is successfully rendered onto the map, displaying its title and description

TC-5: View Map

Test-case Identifier:	TC-5
Use Case Tested:	UC-4: ViewMap
Pass/fail Criteria:	The test passes if the a MapView component is rendered onto the

	page after permission is granted for location
Input Data:	User presses to grant/reject location permission

Test Procedure	Expected Result
Step 1: User rejects permission for the system to get location	System provides a view with a spinning indicator and asks user to accept permission
Step 2: User grants permission but the Map View is not successfully fetched	System provides a view with a spinning indicator
Step 3: User grants permission and the Map View is successfully fetched	The Map View is displayed on the screen

TC-6: Upload Photo

Test-case Identifier:	TC-6
Use Case Tested:	UC-3: Photos
Pass/fail Criteria:	The test passes if the system successfully uploads the photo
Input Data:	File

Test Procedure	Expected Result
Step 1: User clicks on a photo in the grid of photos which has not been uploaded yet. **Step 2:** System processes the photo by fetching the creation timestamp, the location coordinates, and reverse geocoding the coordinates. **Step 3:** System uploads the photo to the Firebase Storage bucket. **Step 4:** System creates a database object for the photo.	System uploads the photo and marks the photo as uploaded by changing the opacity of the photo in the grid.

TC-7: Sort by Rating

Test-case Identifier:	TC-7
Use Case Tested:	UC-6: FindAttractions
Pass/fail Criteria:	The test passes if the user is displayed attractions in order by ranking.
Input Data:	User presses the rating tag

Test Procedure	Expected Result
Step 1: User presses the rating tag	The system displays the attractions in order by ratings in order from highest to lowest
Step 2: User presses the rating tag again	The system displays the ratings in order from lowest to highest

TC-8: Sort by Price

Test-case Identifier:	TC-8
Use Case Tested:	UC-6: FindAttractions
Pass/fail Criteria:	The test passes if the user is displayed attractions in order by qualitative price rankings.
Input Data:	User presses the price tag

Test Procedure	Expected Result
Step 1: User presses the rating tag	The system displays the attractions in order by price from highest to lowest
Step 2: User presses the rating tag again	The system displays the attractions in order by price from lowest to highest

TC-9: Save Attraction

Test-case Identifier:	TC-9
Use Case Tested:	UC-6: FindAttractions
Pass/fail Criteria:	The test passes if the user saves an attraction successfully

Input Data:	User presses the save button
Test Procedure	**Expected Result**
Step 1: User saves an attraction that has been already saved	System displays a warning saying the attraction has already been saved
Step 2: User saves an attraction that has not been interacted with	System successfully saves attraction
Step 2: User tries to save the attraction that they just saved.	System prevents the user from saving redundant entries.

TC-10: Delete Attraction

Test-case Identifier:	TC-10
Use Case Tested:	UC-6: FindAttractions
Pass/fail Criteria:	The test passes if the user saves an attraction successfully
Input Data:	User presses the save button
Test Procedure	**Expected Result**
Step 1: User long presses an attraction	System displays a warning window asking the user if they want to delete the item.
Step 2: User presses the delete button	System removes the attraction item from the saved items view

TC-11: Add Budget

Test-case Identifier:	TC-11
Use Case Tested:	UC-5
Pass/fail Criteria:	Budget correctly appears in the budget page after added
Input Data:	User inputs a number
Test Procedure	**Expected Result**
1. User clicks to add a new budget 2. User inputs a number and presses to add the new budget 3. User views the budget in the budget	1. System takes the user to the add budget page 2. System saves a reference for the new budget item and returns to the budget

page 4. User views the average spending per day	page 3. System retrieves the amount for the budget object and displays it on the budget page 4. System retrieves the number of days of the trip and calculates the average spending per day

TC-12: Add Purchase

Test-case Identifier:	TC-11
Use Case Tested:	UC-5
Pass/fail Criteria:	Purchase correctly appears in the budget page after added
Input Data:	User inputs purchase amount and type
Test Procedure	**Expected Result**
1. User clicks to add a new purchase 2. User inputs a purchase amount and purchase types and presses to add the new purchase 3. User views the new purchase in the budgeting history on the budget page 4. User views a decrease in the budget due to the purchase	1. System takes the user to the add new budget item page 2. System creates a reference for the new purchase item and saves this before returning to the budget page 3. System retrieves the purchase amount and type and displays it in the history 4. System retrieves the budget and subtracts the purchase amount from the budget and displays this on the screen

TC-13: Calendar View

Test-case Identifier:	TC-13
Use Case Tested:	UC-7: ViewCalendar
Pass/fail Criteria:	Test passes if the Calendar Page generates and displays correctly
Input Data:	User taps on the calendar icon
Test Procedure	**Expected Result**
Step 1: User clicks on “calendar”	System generates a calendar view of days that begin at the beginning of the month and extend to

Step 2: System generates a calendar page with correct date information **Step 3:** System will display the calendar information along with the trip information within the calendar	the last day of the month (based on the date range)

TC-14: Daily Summary

Test-case Identifier:	TC-14
Use Case Tested:	UC-7: ViewCalendar
Pass/fail Criteria:	Test passes if the Daily/Trip Summary generates below the calendar
Input Data:	User taps on the calendar icon and Date Range or Date Selection

Test Procedure	Expected Result
Step 1: User clicks on “calendar” **Step 2:** System generates a calendar page with correct date information **Step 3:** User specifies the date of the trip they would like to focus on by tapping on the date in the calendar view **Step 2:** System compiles the data from th single date selection and displays a summary of the data called the “Daily Summary” correctly under the calendar display	System displays the correct daily information that corresponds to any day of a trip. This includes the compilation of photos and journal entries only from the specified date.

TC-15: Journal Entries

Test-case Identifier:	TC-15
Use Case Tested:	UC-8: Journalling
Pass/fail Criteria:	Test passes if all journal entries are correctly displayed
Input Data:	User taps on the journal icon on the bottom toolbar

Test Procedure	Expected Result
Step 1: User navigates to the journal page by	System generates a journal entry page that

tapping on the journal icon on the bottom toolbar **Step 2:** System compiles all journal entries for the current trip and displays them correctly and with the corresponding data	contains every journal entry from the current trip with the correct information. Here the user can navigate to view any of their journal entries.

B. Test Coverage

TC-1 - Logging Into the System

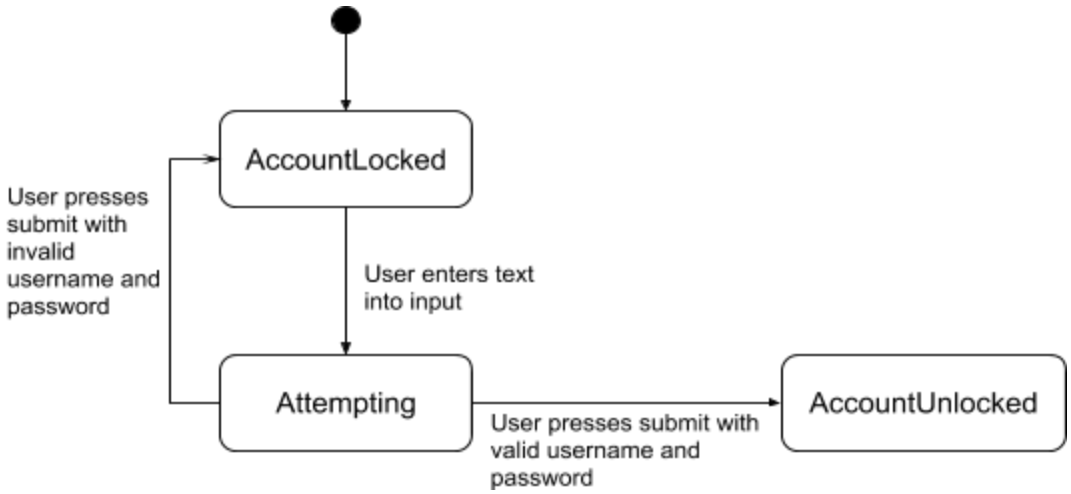
State-based Testing for Logging In:

Define the following states as combinations of attribute values:

1. “AccountLocked” = defined as:
   (loggedIn == false) && (username == “”) && (password == “”)
2. “Attempting” = defined as:
   (loggedIn == false) && ((username != “”) || (password != “”))
3. “AccountUnlocked” = defined as:
   (loggedIn == true) && (username != “”) && (password != “”)

Define the relevant events:

1. User submits username and password to be authenticated by the system
2. User enters text into username or password input



With this form of testing, we want to cover all possible states - {AccountLocked, Attempting, AccountUnlocked} - at least once. We can do so by triggering the events - {submit, text input} - which cause state transitions and may illuminate bugs in our logging in procedure.

Consider the following valid and invalid coverages for each transition:

(1) **Valid**: AccountLocked → Attempting: User enters text
Expected Result: Login view re-renders to display input from user

(2) **Valid**: Attempting → AccountUnlocked: User presses submit with valid input
Expected Result: Username and password are not empty and the system verifies that the user exists in the database - renders the home page view

Invalid: Attempting → AccountLocked: User presses submit with invalid input
Expected Result: Username or password is empty or the system does not recognize the user in the database - re-renders the login view with empty username and password inputs

TC-2 - Information for Creating a Trip

Consider the following equivalence classes for entering the title of the trip:

Class A	Class B	Class C
Consists of valid characters; Length is greater than 0	Consists of invalid characters	Has length of zero

Class A covers the valid inputs and class B and class C cover the invalid inputs. If the input is valid then it must consist of characters recognized by the system and have a non-zero length. If this is the case, then the trip is added to the user's list of trips and the trip view is re-rendered. Otherwise, there are two cases where the input can be invalid. Class B represents the first case, where there exists some characters in the input which are not supported by the system. An input in class B would alert the user that the characters used are not supported. The other case is represented by class C, when the user submits a trip with an empty title. This is invalid so the system sends an alert saying the title cannot be empty.

We may choose just a single candidate from each of class A, class B, and class C because they are equivalence classes. As a result, we may achieve high coverage by just using three inputs. This test case would require us to write three unit tests. One which checks that the expected result of a valid input from class A is a success. The other two would check that the expected result from inputs from Class B and Class C, respectively, are failures.

TC-3 - Tests location between pins when adding a pin

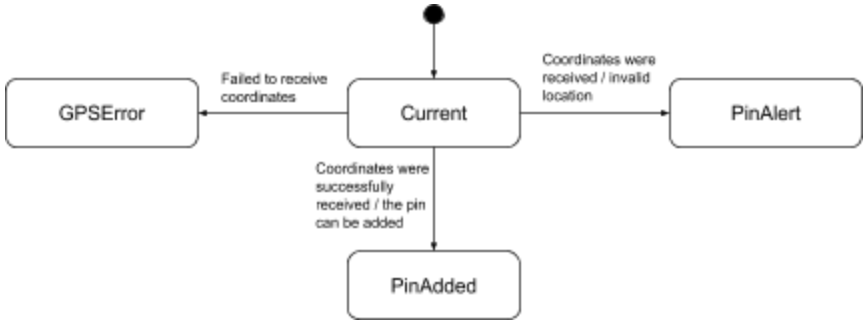
State-based Testing for location when adding a pin:

Define the following states as combinations of attribute values:

1. “Current” = defined as:
   (coords == null) && (canAdd == false)
2. “PinAlert” = defined as:
   (coords != null) && (canAdd == false)
3. “PinAdded” = defined as:
   (coords != null) && (canAdd == true)
4. “GPSError” = defined as:
   (coords == -1)

Define the relevant events:

1. User presses the add pin button under the map, which retrieves the user’s gps coordinates and checks if a pin can be added in the user’s current location



The Current state represents the state of the application before the user presses the button. The PinAlert state shows an alert saying that the pin was not added to the map. The GPSError state shows an alert saying that the system failed to retrieve the gps coordinates. The PinAdded state shows the map with a new pin in the user’s current location.

This state diagram shows both the valid and invalid cases. The valid case is when the state transitions from Current to PinAdded and the map is re-rendered to display the new pin. There are two invalid cases - when the system fails to retrieve the coordinates and when there is another pin within a 25 foot radius of the user’s current location.

All of these cases can be tested by using transitions between states.

TC-4 - Tests title and description validation when adding a pin

Define the following equivalence classes:

Class A	Class B	Class C
Title is empty; Description is either empty or non-empty	Description is empty; Title is non-empty	Title and description are non-empty

We have three disjoint classes consisting of all possible inputs to the title and description for a user-generated pin. We have defined an invalid submission to have an empty title and/or an empty description and a valid submission has a non-empty title and a non-empty description. For our tests to run successfully, we would expect that alerts would be displayed by the system when the user attempts to make an invalid submission, and we would expect that the system adds the pin to the map when the user makes a valid submission.

In our unit tests, to check that the test runs as expected, we will utilize the submitPin() function within the AddPinPage class. This function returns some useful information such as the number of errors and the type of errors that occurred. If we find that number of errors is 0, then we know that it was a valid submission. If the number of errors is one or two and the error type has to do with the title, then we know the input was a member of class A, so we will expect the alert for title errors to come up. Otherwise, the input will be a member of class B, so we will expect the alert for description errors to come up.

TC-5 - Tests Map renders when location is granted

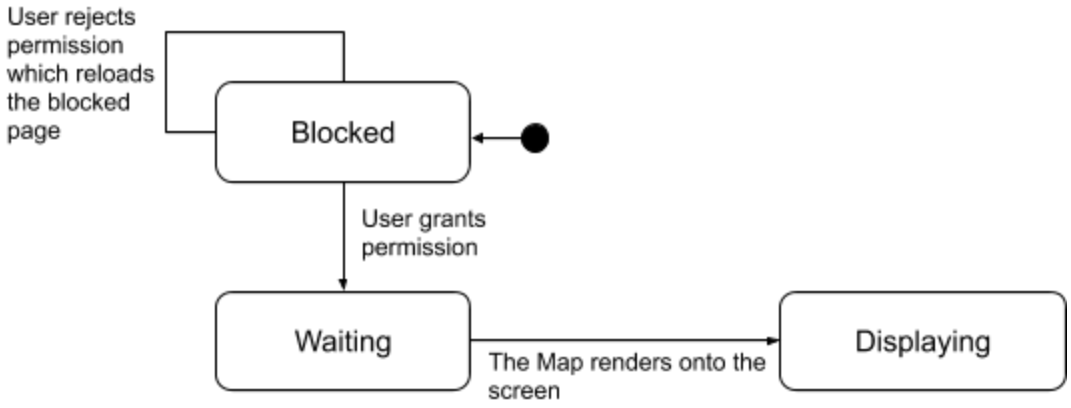
Define the following states:

1. “Blocked” = defined as:
   (hasLocationPermissions == false)
2. “Waiting” = defined as:
   (hasLocationPermissions == true) && (location == null)
3. “Displaying” = defined as:
   (hasLocationPermissions == true) && (location != null)

Transitions between states occur when triggered by the following events:

1. User grants permission to use his/her location
2. User rejects permission to use his/her location

For these states and events, we have the following state diagram:



This test has to do with the front-end side of the map. It makes sure that the view responds correctly to the user's actions. When the user rejects permission, then we cannot load the map because we won't have a location on the system side to focus on. Hence, we would expect that a loading screen appears with a spinning bar. A useful React-Native testing capability is that we are able to get a snapshot of our view at any time, so once we trigger an event that causes a transition in the state, we can simply check that the proper view has rendered onto the screen. The snapshot of the view consists of a tree of components and we can confirm that the correct components have been rendered onto the map. Therefore, to test all possible states we can trigger transitions and confirm what we expected.

TC-6 - Tests Photo Metadata when Uploading a Photo

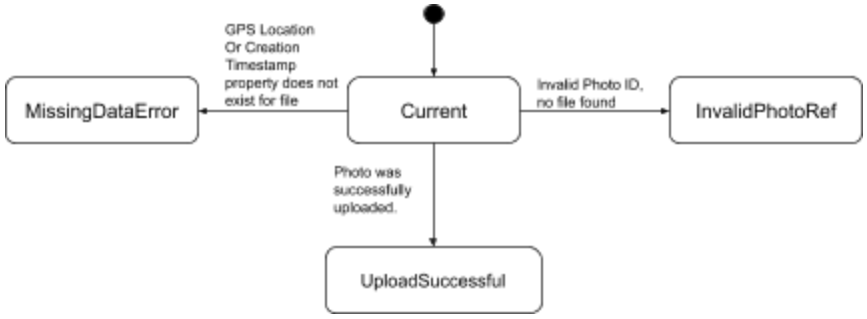
State-based Testing for photo data when uploading:

Define the following states as combinations of attribute values:

1. "Current" = defined as:
   (assetInfo == null && photo == null)
2. "MissingDataError" = defined as:
   (assetInfo != null &&
   (assetInfo.location == null || assetInfo.creationTimestamp == null)) && (photo != null)
3. "InvalidPhotoRef" = defined as:
   (assetInfo == null) && (photo != null)
4. "UploadSuccessful" = defined as:
   (assetInfo != null && assetInfo.creationTimestamp != null && assetInfo.location != null && photo.uploaded == true)

Define the relevant events:

1. User taps on an un-uploaded photo tile and the system attempts to process the metadata before it uploads the file and creates a database reference.



The current state represents the point in the application before a user interaction explicitly asks the system to upload a photo file to the Firebase storage bucket and store it as part of their current trip, as a result, both assetInfo and photo are null references. Once a photo reference is passed into the system, it should then be able to fetch the metadata for the file using the ID provided by the device. The metadata is stored in assetInfo, and if this is returned as null, then we recognized an error where we passed in an incorrect ID to the device as it was not able to fetch the file.

If, on the other hand, it's successful, the system then checks for the two primary data points in the metadata file - location and timestamp. If these don't exist, the system moves into an error state of MissingDataError. If it is able to find the information, it uploads the file, and moves into UploadSuccessful, which is then reflected back to the user with a change in opacity to the image tile.

TC-7 Sort By Rating

Class A	Class B
System sorts in descending order by rating	System sorts in ascending order by rating

For a user to sort attractions by rating, he or she needs to press the ratings button in the attractions view. This action can be treated as a boolean input, since the attraction can be either be sorted by ascending view or descending view.

Therefore, we can split the input parameters into two input groups, one ascending and one descending. We can write two unit tests by calling the sortRating() method on the array of attractions saved in the component state. One test will iterate through the array and check to see if every rating is less than the next item's rating, and the other test will iterate through the array and check to see if every rating is greater than the next item's rating

TC-8 Sort By Price

Class A	Class B
System sorts in descending order by price	System sorts in ascending order by price

For a user to sort attractions by price, he or she needs to press the price button in the attractions view. This action can be treated as a boolean input, since the price can either be sorted by ascending view or descending view.

Therefore, we can split the input parameters into two input groups, one ascending and one descending. We can write two unit tests by calling the sortPrice() method on the array of attractions saved in the component state. One test will iterate through the array and check to see if every price rating is less than the next item's price rating, and the other test will iterate through the array and check to see if every price is greater than the next item's price.

TC-9 Save Attraction

Class A	Class B	Class C
System has not saved attraction	System has already saved attraction	System tries to save attraction again

For a user to save an attraction, he or she needs to press the save button on an attraction. This action can be treated as a boolean input, since the attraction can be either saved if the attraction has not been previously saved or failed to save if the attraction has already been saved.

Therefore, we can split the input parameters into three groups: attractions that haven't been saved, attractions that have been saved, and attractions that were attempted to be saved again. We can test to see if an item is saved by creating a test item and then calling the saveItem() function, which takes the item to be saved as an input. We expect the length of the array of saved items to therefore be 1. This test not only checks class B, but also inherently tests class A since only the item we saved is in the array, and no others belong to the array. For the second test case, we can try to call the saveItem() function again on the same item, yet still expect the array to be of length 1 because the system detects that the item is already in the array and prevents redundant action.

TC-10 Delete Attraction

Class A	Class B
System has not deleted attraction	System has deleted attraction

For a user to delete an attraction, he or she needs to long press on the attraction card in the saved items view and delete the attraction. This action can be treated as a boolean input, since the attraction can either be deleted or not deleted, and we can divide this action into two groups.

We only need to write one test case, however, because to check whether the system has not deleted an attraction is trivial. By saving some test item to delete and checking the length of the saved items array, we already understand that no item was deleted. Therefore, after saving this test item we can call the deleteHandler() method, which takes in item to be deleted, and then expect the length of the saved items array after to be zero.

TC-11 Add Budget

Class A	Class B
System has not added the budget	System has added the budget

In order for the user to add a new budget, the user must press to add the new budget and proceed to agree to adding the budget on the add budget page. In this case, there are really only two possibilities: the budget is added or the budget is not added. Since it is split up into two opposite results, we can use booleans to describe these.

To test this, we simply have to press to add the budget and return to the budget page to see if this exists. There are two ways to recognize if the budget has been saved by the system. Firstly, it must be displayed as the total budget in the budget page. Secondly, the budgeting per day is calculated by taking the total budget and dividing it by the number of the days in the trip. This is the second indicator if the budget has been saved, even if it hasn't been displayed.

TC-12 Add Purchase

Class A	Class B
System has not added the purchase	System has added the purchase

In order for the user to add a new purchase, the user must press to add the new purchase and proceed to agree to adding the purchase on the add budget page. In this case, there are really only two possibilities: the purchase is added or the purchase is not added. Since it is split up into two opposite results, we can use booleans to describe these.

To test this, we simply have to press to add the purchase and return to the budget page to see if this exists. There are two ways to recognize if the purchase has been saved by the system. Firstly, there already exists an array of purchases, which may be equal to zero. These purchases will be displayed as a history of purchases and the new purchase that has been

added must be added to both the array and to the display that shows this array. Secondly, we can see if the system saved the purchase through the budgeting function. If a new purchase is added, the budget is subtracted from and the budget becomes smaller by the purchase amount. This can show if the system is saving the budgeting information, even if it is not being displayed.

TC-13 - Checking the Calendar View

Define the following equivalence classes for viewing calendar:

Class A	Class B	Class C
Single trip within the specified date range	Multiple trips within the specified date range	No trips within the specified date range

In order for the calendar view to be displayed correctly, the user will tap on the calendar icon from the bottom toolbar for the calendar view to display. This test case will cover three classes of tests. One of which is when there are no trips on the generated calendar view in which case, the desired outcome is that there is no trip data contained in the calendar. The second class of tests is when the specified date range only encompasses one trip. This test case should yield a calendar view that contains only one calendar cluster of dates that contain information about the trip (and there will be only one trip summary showing the correct trip information). The third class of tests is when there are multiple trips contained in the calendar date range. In this case, the desired outcome is to have more than one cluster of days that has trip data associated with it as well as multiple trip data per date

TC-14 - Checking the Daily View Summary

Outcome A	Outcome B	Outcome C	Outcome D
Daily View Summary is displayed correctly with the corresponding data	Daily View Summary is displayed correctly without the corresponding data	Daily View Summary is not displayed correctly but has the corresponding data	Daily View Summary is not displayed correctly and does not have the corresponding data

For the user to view the Daily View Summary, he or she could first navigate to a Calendar View that contains one or more trips, then click on a day that has a daily summary. There are four outcomes from this test and only one is the desired. Outcome A is the desired, in which the Daily Summary is displayed according to the User Interface Specifications and the data displayed corresponds to the date for which it was inputted originally. Every other outcome is not desired because either the display is not according to User Interface Specifications or the

data shown does not correspond to the date and is incorrect. Please note that there might be more than one corresponding trip per daily summary if there are trip dates that overlap (this case is also tested within TC-14).

.

TC-15 - Verifying Journal Entries

Outcome A	Outcome B	Outcome C	Outcome D
Journal Entries are displayed correctly with the corresponding data	Journal Entries are displayed correctly without the corresponding data	Journal Entries are not displayed correctly but has the corresponding data	Journal Entries are not displayed correctly and does not have the corresponding data

For the user to view the list of journal entries from a specified day, he or she should first navigate to the Journal Page and have logged at least one trip. There are four outcomes from this test and only one is the desired. Outcome A is the desired, in which the Journal Entries are displayed according to the User Interface Specifications and the data displayed corresponds to the date for which it was inputted originally. Every other outcome is not desired because either the display is not according to User Interface Specifications or the data shown does not correspond to the date and is incorrect.

C. Integration Testing Strategy

1. TC-1

Test-case Identifier:	TC-1
Use Case Tested:	UC-8
Pass/fail Criteria:	The journal entry that was made shows up on the journal page
Input Data:	Journal Entry

Test Procedure	Expected Result
1. Press to add a journal entry 2. Add a journal entry 3. Return to the journal page and check if the journal entry inputted is displayed on the page	1. System takes user to add journal page 2. System saves a reference for the new Journal object with the given data (title, text, etc.) 3. System retrieves saved entry and displays the journal entry on the journal page

2. TC-2

Test-case Identifier:	TC-2
Use Case Tested:	UC-7
Pass/fail Criteria:	The date set in creating a trip is set correctly in the calendar page
Input Data:	Date

Test Procedure	Expected Result
1. Press to create a new trip 2. Set the start and end date and press to create the new trip 3. Press the calendar tab to view the calendar page 4. View the calendar to confirm the correct dates are set	1. System takes the user to the new trip page 2. System saves the trip with the name, location, and start and end dates as a trip object 3. System opens the calendar page 4. System retrieves the trip object and displays the selected dates

3. TC-3

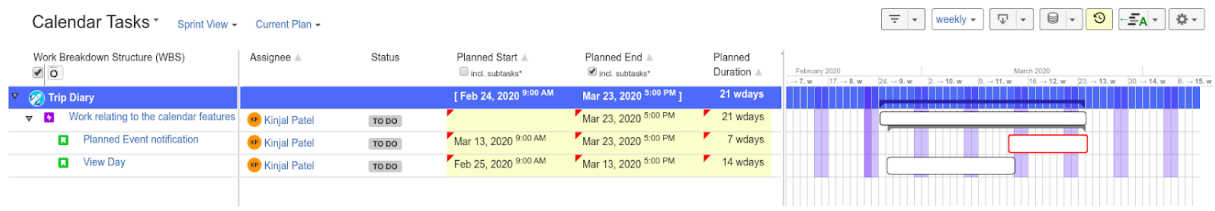
Test-case Identifier:	TC-3
Use Case Tested:	UC-5
Pass/fail Criteria:	The purchase added shows up on the budget page
Input Data:	Budget Item

Test Procedure	Expected Result
1. Press the budget tab on the trip page 2. Press to add a new purchase item 3. Input a purchase amount and a purchase type and press to add new purchase 4. View the purchase on the budget page in history of purchases	1. System retrieves the budget information stored for that trip 2. System takes the user to the new purchase page 3. System saves a new reference for a purchase with the type and amount and returns to budget page 4. System retrieves the purchase information and displays it

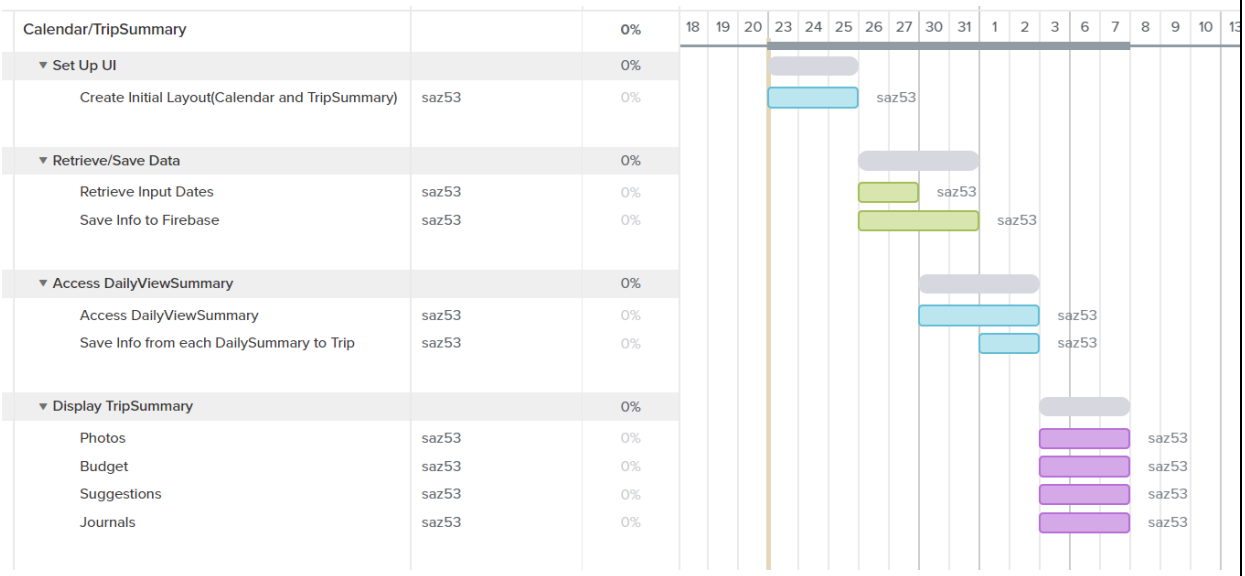
13. History of Work

A. Calendar

Report #1



Report #2



1. Deadline Updates

In the first report, broader goals were set as we were still designing what the calendar would include. However, in report two, the goals are much more specific and clearly state what we plan to implement. Throughout the first and second report, the main focus was setting up the calendar. From here, we need to work on setting up the daily summary to display the data of the current selected day.

2. Current Status

Currently, we have organized the journals and photos by having a field in the database. When the calendar queries the index, it filters all of the information so that it only fetches the information (photos and journals) from a specific day. We do this through unix timestamps.

Additionally, the calendar has implemented a weather feature. After asking the user's permission to access their location, the calendar retrieves the user's latitude and longitude coordinates. It then calls an API, giving it the user's current latitude and longitude to retrieve the weather information. It then displays this weather information underneath the calendar. It can only display the current real-time weather, so selecting different days has no effect on the weather call.

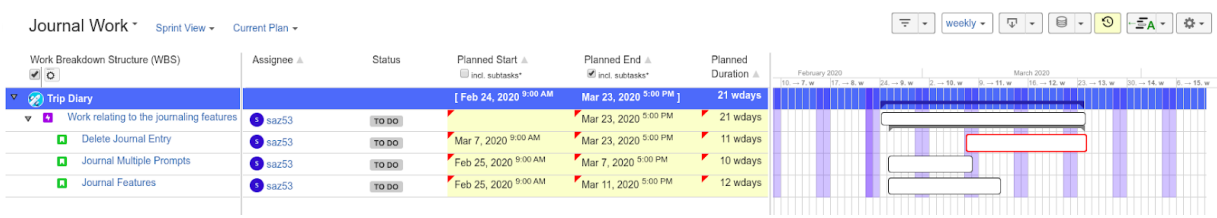
3. Future Work

Future work entails implementing budget and attractions into the daily summary. When a user clicks on a future day in the calendar, they should be presented with attractions that they have saved and the daily budget allocated for that day.

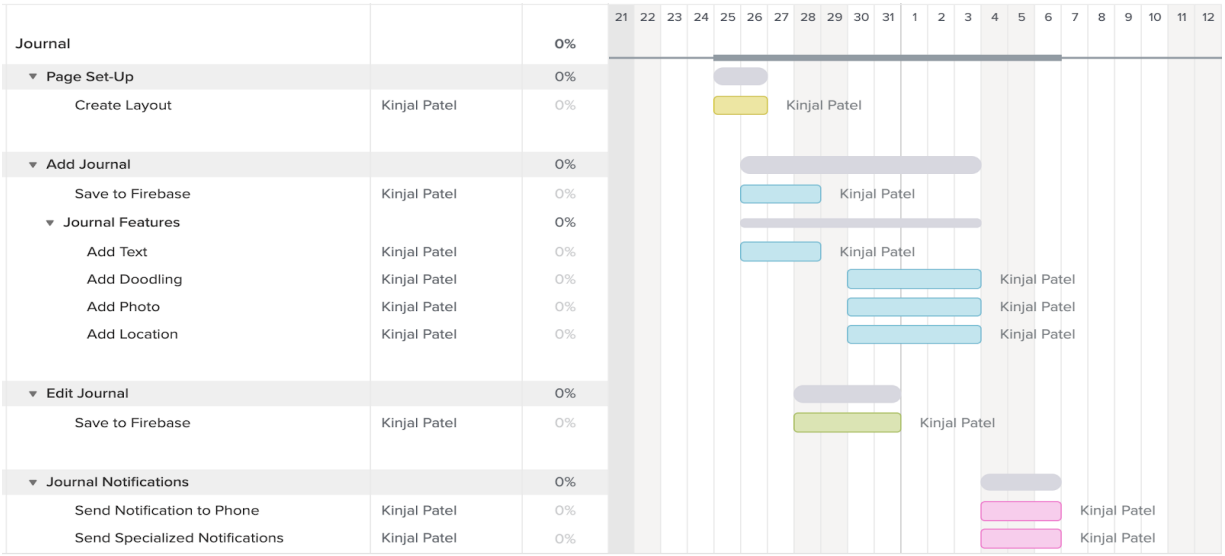
Additionally, I would like to implement a planning feature where users can schedule any event they want at a certain time. When this event is in the near future, they would receive a notification.

B. Journal

Report #1



Report #2



1. Deadline Updates

Considering the deadlines set above, we were able to implement functionality for all use cases as defined from Report 1. Below summarizes the deadlines set from Report 1 and 2:

Report 1	Deadlines Accomplished Currently
1. Delete journal entry 2. Multiple prompts asked 3. Doodling feature	1. Delete journal entry

Report 2	Deadlines Accomplished Currently
1. Create page layout 2. Save journal to firebase 3. Add journal features 4. Send notifications	Allowed for **basic functionality of all journal use-cases** 1. Page layout created 2. Saved journal to firebase 3. Journal can be deleted 4. Journal entries are displayed to user on main page 5. Journal can be edited

2. Current Status

Currently, we have taken feedback into account and are trying to implement cross-functionality between features. This included reconsidering how the journal will be able to be more sophisticated to provide automation for the user.

Report 3	Deadlines Accomplished Currently
1. Add a journal commit chart 2. Journal sorting based on date 3. Add autofill location tags 4. Add pictures associated with journal to entry card	1. Added journal commit chart, which offers journal sorting functionality 2. Added location tags, which autofill based off of map pins and saved attractions 3. Allowed for the user to view a picture associated with their journal card based on existing photos uploaded or saved attractions

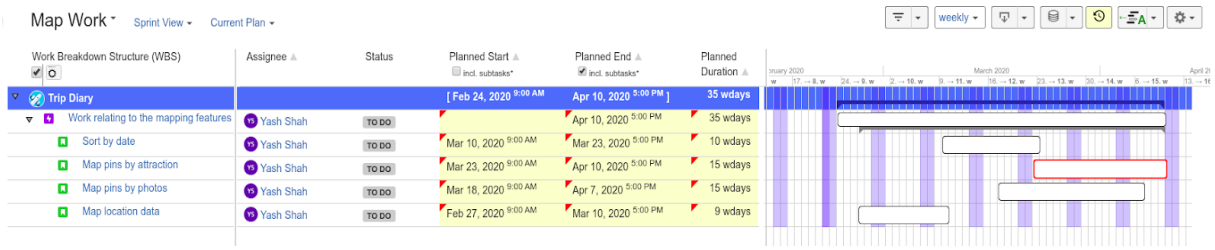
3. Future Work

The journal feature will be fully cross-functional within other features in the application. The cross functional features that we anticipate on completing are the following:

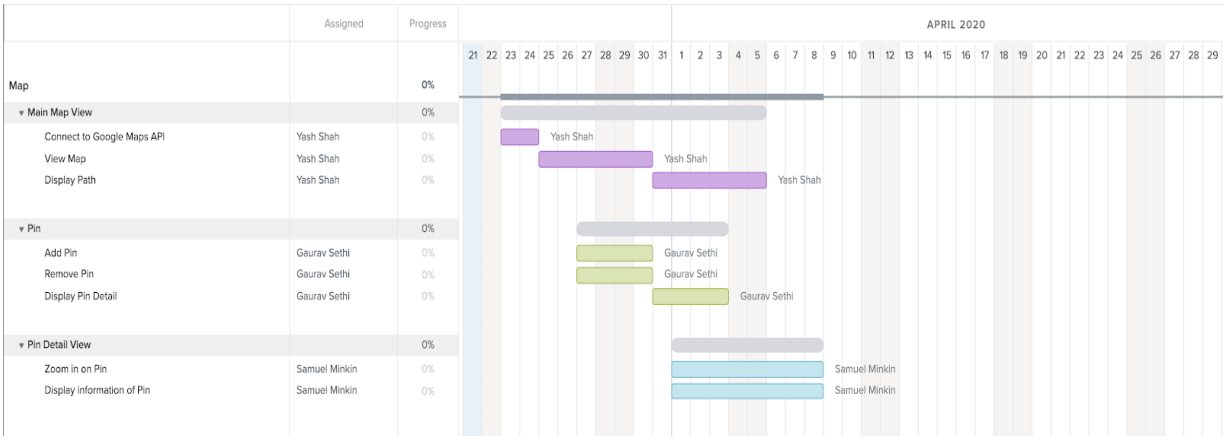
1. Sending notifications for the user to be reminded to write their journal entry

C. Map

Report #1



Report #2



1. Deadline Updates

In our first report, we listed bigger picture goals for the map and by the time we got to the second report we had a much better picture of a better progression to eventually get to those bigger picture goals. Hence, many of the deadlines from report #1 have been pushed off to after the deadlines in report #2. We focused on building the basic parts of Map, Pin, and Pin Detail View first, and our next steps were to make the map more class-platform dependent by adding pins by attraction, fetching nearby user photos, and displaying routes.

2. Current Status

By the time of final submission, we have implemented almost all features of the map and pins and have included some of the more advanced features such as displaying a custom callout for both types of pins - user-generated and auto-generated based off of attractions.

First Report Completed Features:

- Map pins by photos
- Map location data
- Map pins by attractions

Second Report Completed Features:

- Add Pin, Remove Pin, Display Pin Detail
- Zoom in on pin, Display information of pin
- Connect to Maps API, View Map
- Display routes

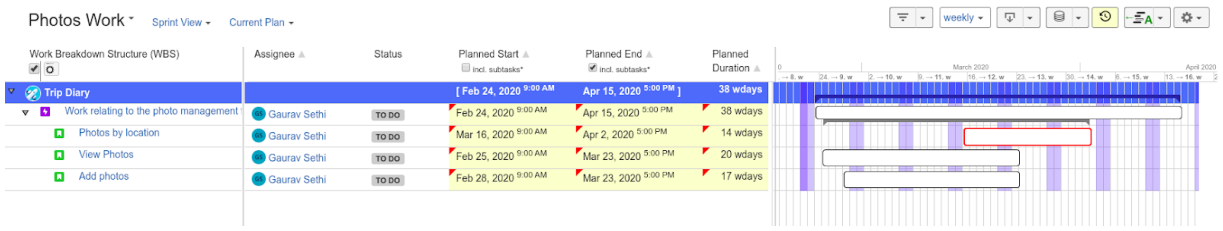
3. Future Work

In the future, we hope to be able to display the map by date. For each date, we will grab the locations of suggestions that the user has saved and render them as pins on the map to increase cross-platform interaction.

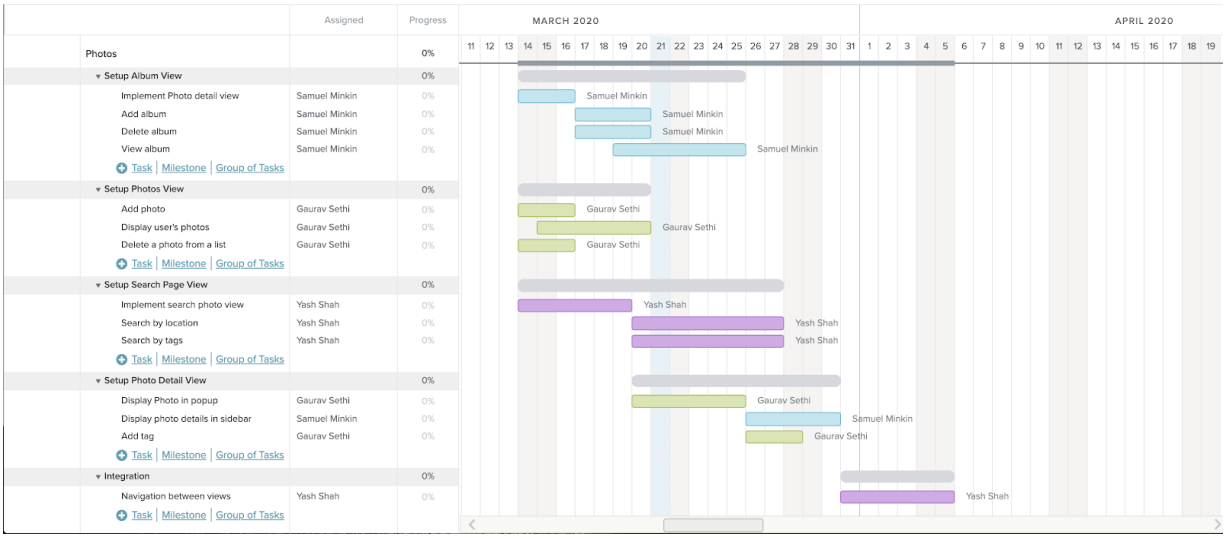
- Sort by date

D. Photos

Report #1



Report #2



4. Deadline Updates

In the second report, we had a lot more in depth use cases of setting up the structure of the photos feature as well as specifics of additional features that may enhance the user's experience, but during the demo, we were asked to revert to our original approach. We have one use-case defining this feature work, all the while encapsulating the user features that we defined earlier to make the system more comprehensive.

Rather than using a system which required the user to individually select each photo from their library to import, the system now fetches all photos that occurred during the duration of the trip and prompts the user to automatically upload them with a single tap. This significantly reduces our click rate, and makes for a better experience for the end user.

5. Current Status

We have implemented the bare features of
First Report Completed Features:

- View Photos
- Add Photos

Second Report Completed Features:

- Add photo, display users photos
- Navigate between viewers
- Display photos as pin on map

Third Report Completed Features:

- Grid view with ability to tag which photos were uploaded
- EXIF data processing to fetch location & creation timestamp
- Filtering photo library to display photos that were taken during a trip

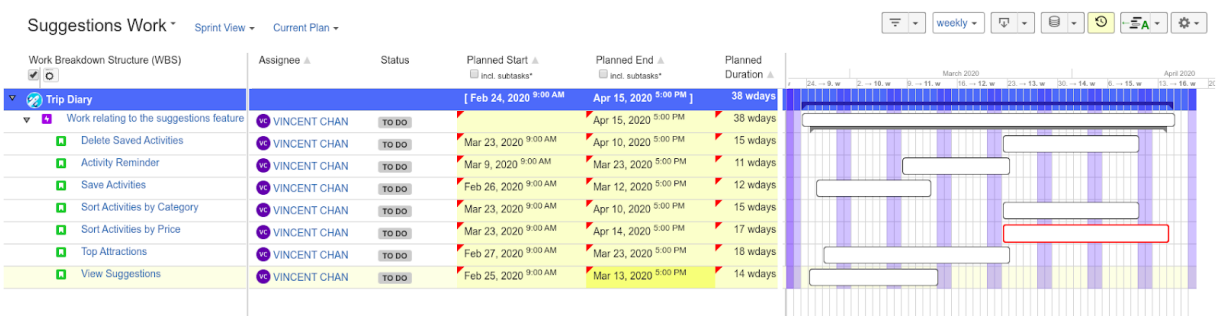
6. Future Work

In the later stages, we setup our data model to accept additional tags and search query terms for each photo. We hope to make use of an image classification network to process the classifications found in each photograph, and append them as a searchable list of strings. This will enable the user to browse for photos of a specific item they remember taking from their photo library.

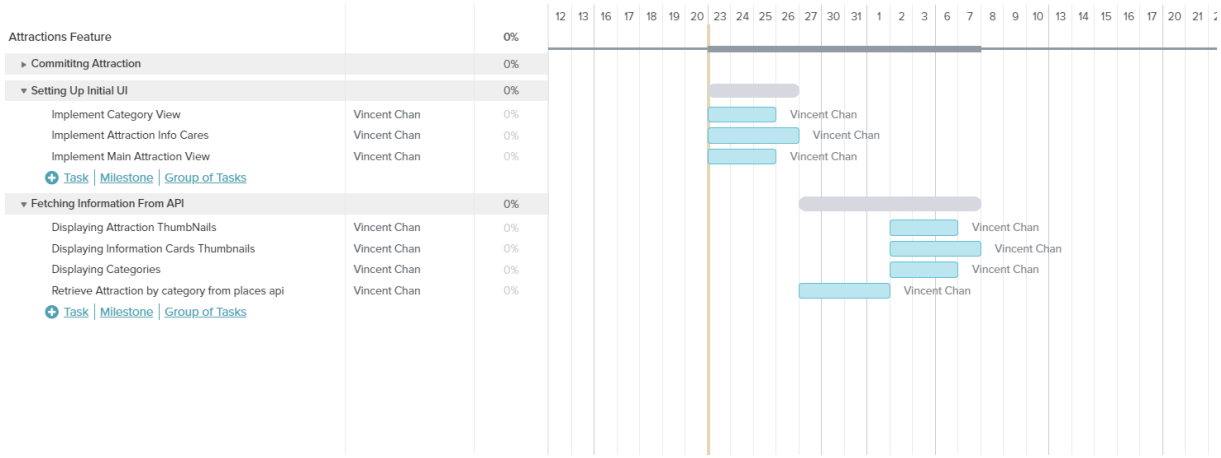
We also plan to create an auto upload feature. Currently, while the system presents the user with all their photos, we do not auto upload as a means to provide privacy features for the user, but we recognize the use case where a user may want to automatically process and upload new pictures they take.

E. Attractions

Report #1



Report #2



1. Deadline Updates

- We ended up integrating the saved attractions into the calendar feature, allowing users to plan for future days of their trip while browsing their trip history in the calendar. On top of that we also used the attraction information in the journaling and pins features as well.
- On top of this, we were able to successfully implement sorting by different tags and providing more information to the user when clicking on an attraction that they are interested in.

2. Current Status

Here is a summary listing of the features complete throughout the project:

First Report:

- Category, attraction, saveAttraction views

- Retrieve information from Google Places API
- Save attractions to saved attractions view
- Delete Saved Attraction

Second Report:

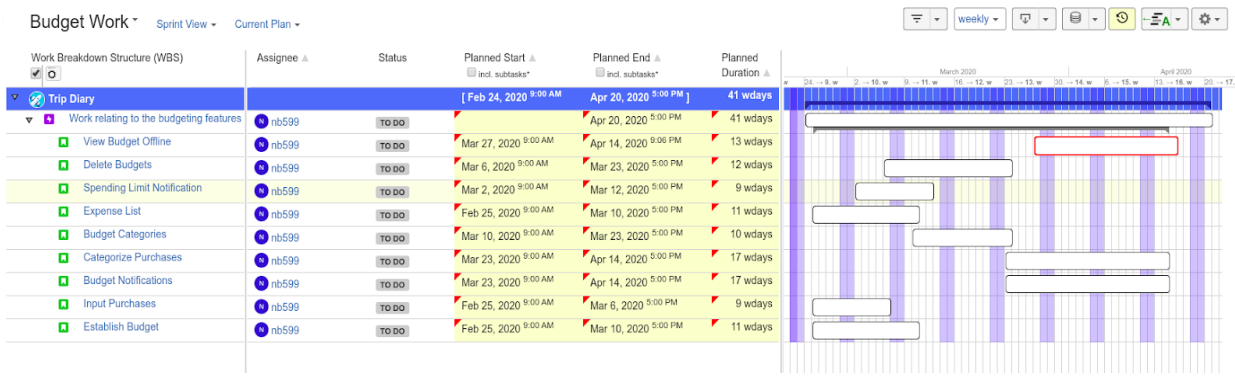
- Information Window view
- Retrieve more information from the google places api (photos, reviews, etc.)
- Display custom information for each business
- Sort by ratings and price
- Store saved attractions in firebase for other features to use

3. Future Work

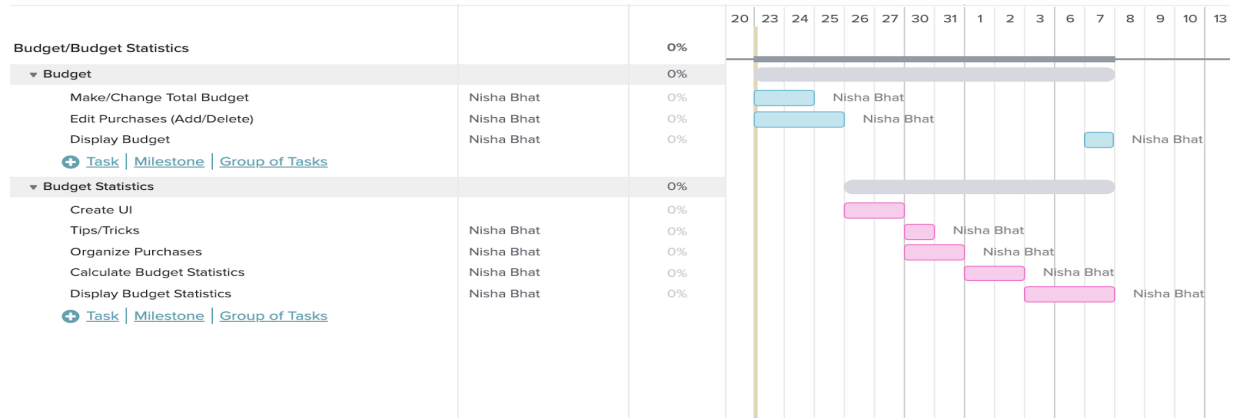
- In the future it would be nice to include a robust recommendation system perhaps by storing the information of frequently visited attractions and doing some processing to determine how to recommend new attractions
- More tags for sorting information such as ranged pricing, ratings, distance
- Search bar for custom attractions searches with autofill
- UI improvements

F. Budget

Report #1



Report #2



1. Deadline Updates

Post Report 1: In the first and second reports, many of the components were independent of each other and there was a lack of automation to make budgeting easier for the user. According to deadlines, the cross functionality and automation needs to be implemented.

Post Report 2: Everything that was meant to be completed by the previous deadline was implemented.

2. Current Status

Post Report 1: The current status of the budgeting functionality is fairly simple. The budget has been implemented to work in the most basic sense. The budget can be added and viewed by users.

Post Report 2: The complexity that was needed after Demo 1 was integrated into the budget functionality to encourage budgeting through a financial limit per day. This was to take some of the burden off the user and have some automation that can increase the ease of the budget function and increase the complexity of the spending section of the application.

3. Future Work

Post Report 1: Cross-functionality must still be implemented. Automation will be used to divide the budget equally into the weeks and days that span the trip. This automation will provide a set amount that the user should ideally spend per day. However, the user can go beyond a daily budget and if they choose to, the budget will be reallocated amongst the remaining days. This provides an estimation for the user to stay within a spending budget and to practice careful spending.

Post Report 2: The budget can be further integrated, especially with the attractions functionality. This would provide less burden on the user and having these two integrated would make the most sense for the user as attractions make up the majority of spending when on vacation or traveling.

G. Breakdown of Responsibilities

Subgroup	Person	Modules / Classes Responsible
Calendar / Daily Summary / Journal	Sam Zahner	Calendar Page 1. CalendarViewController Coordinate Create Trip Page with Calendar 1. TripViewController 2. TripViewEntity
	Kinjal Patel	Journal Page 1. Journal 2. JournalEntity 3. JournalController
	Jonathon Banks	Daily Summary 1. Daily View Controller
Map / Photos	Sam Minkin	Photos 1. Setup Album View 2. Album Controller 3. Photo Detail View Maps 4. Pin Detail View 5. Pin Detail Controller

	Yash Shah	Photos 1. Search Page View 2. Search Page Controller 3. Photos Navigation Maps 4. Map View 5. Map Controller
	Gaurav Sethi	Photos 1. Photos Page View 2. Photos Page Controller 3. Photo Detail Controller Maps 3. Pin View 4. Pin Controller
Budget / Suggestions	Nisha Bhat	Budget 1. Budget 2. Budget Entity 3. Budget Controller
	Vincent Chan	Attractions 1. Attraction 2. Attraction Entity 3. Attraction Controller

14. References

(1) https://en.wikipedia.org/wiki/FURPS**,** Used for non-functional requirements
(2) https://christineelder.com/top-10-benefits-of-travel-journaling/, Used to reference actors
(3) https://www.ece.rutgers.edu/~marsic/Teaching/SE/syllabus.html
(4) https://asana.com/resources/gantt-chart-basics, Used to learn gantt chart
(5) https://en.wikipedia.org/wiki/User_story, Used to write user stories
(6) https://en.wikipedia.org/wiki/FURPS, Used to write non-functional requirements
(7) Figure 1: https://dzone.com/articles/software-architecture-the-5-patterns-you-need-to-k
(8) Figure 2: https://www.intellectsoft.net/blog/mobile-app-architecture/
(9) Figure 3: https://en.wikipedia.org/wiki/Client%E2%80%93server_model
(10) Figure 4: https://www.csitquestions.com/what-is-http-and-https-explained-ccna-tutorial/
(11) System Architecture (Layered): https://medium.com/@mlbors/architectural-styles-and-architectural-patterns-c240f7df88a0