Trip Diary

Software Engineering (14:332:452)
Report 2

Team 5

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

Submission Date

March 22, 2020

Table of Contents

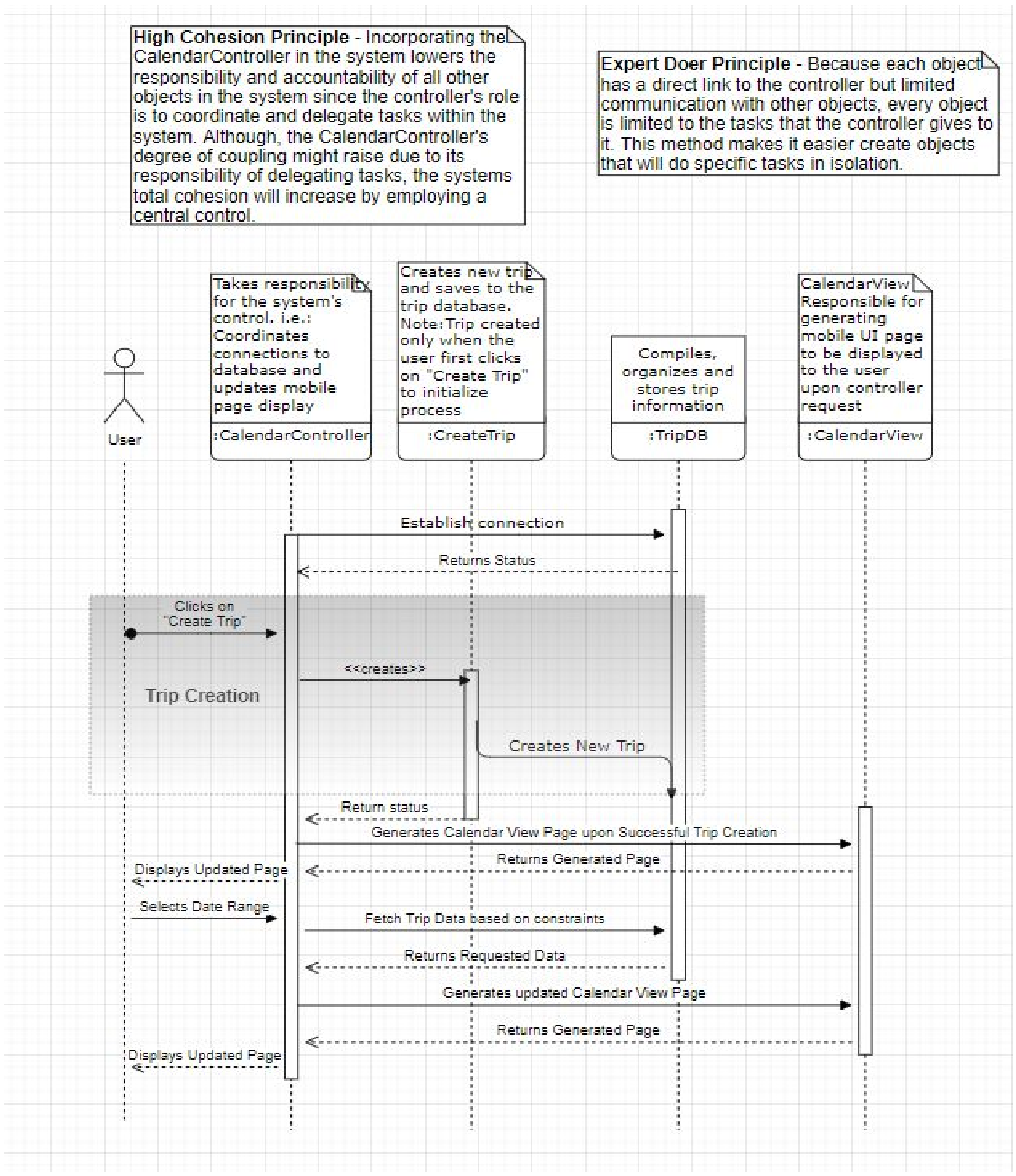
Contribution Breakdown

		Sam Z	Kinjal	Vincent	Nisha	Gaurav	Sam M	Yash	Jon
Project Management	Doc Merge	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
	Coordination	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
	Plan of Work	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Interaction Diagrams	UML Diagrams	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
	Description	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Classes + Specs	Class Diag/Descr	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
	Signatures	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
	Traceability	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Sys Arch + Design	Styles	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
	Pkg Diagram	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
	Map Hardware	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
	Database	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
	Network Protocol	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
	Global Flow	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
	Hardware Req	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Algorithms + Data Struct		12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
User Interface	App Appearance	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
	Prose Descr	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Testing Design		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

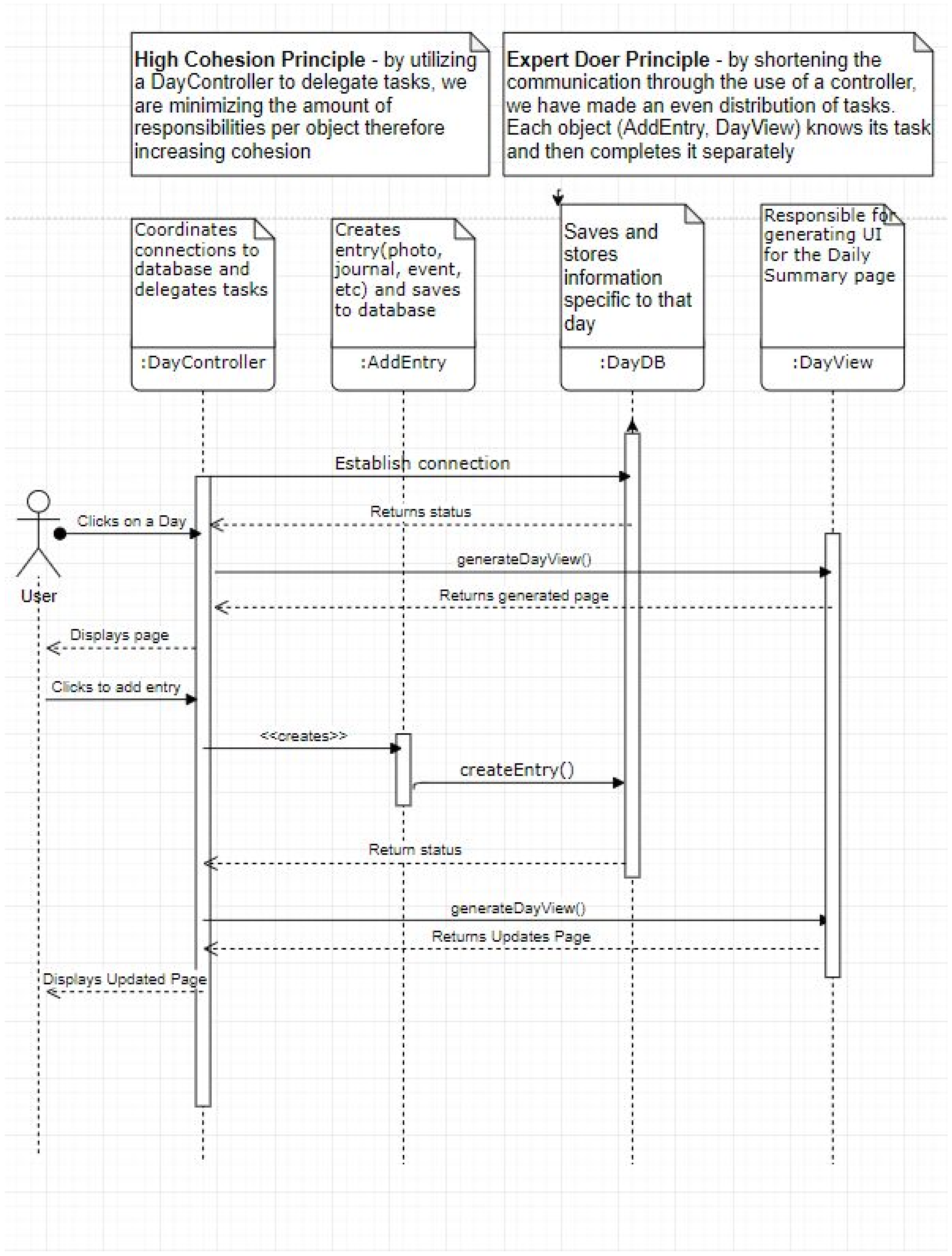
1. Interaction Diagrams

A. Sequence Diagrams / Descriptions

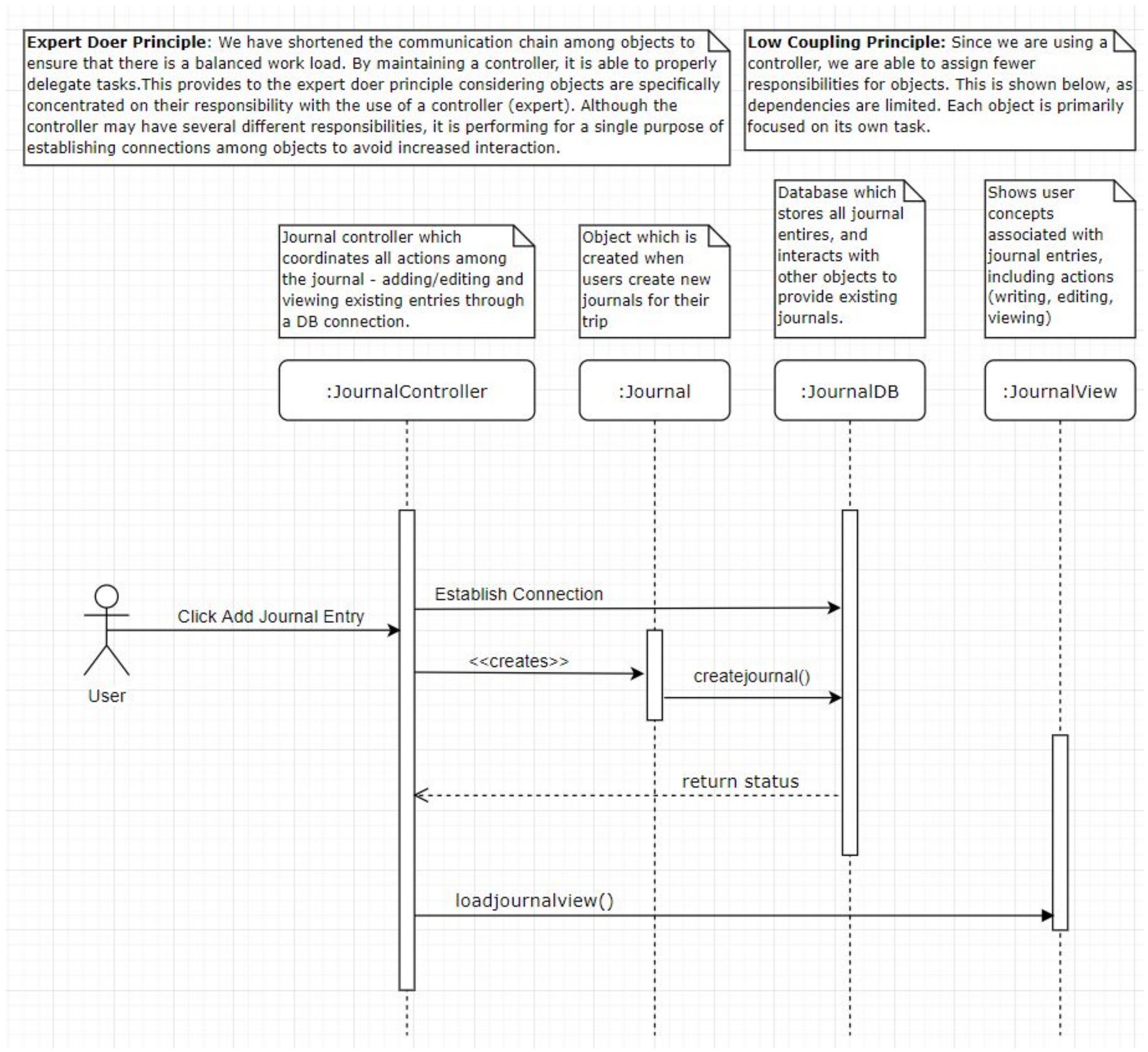
1. View Calendar/Trip Summary


High Cohesion Principle - Incorporating the CalendarController in the system lowers the responsibility and accountability of all other objects in the system since the controller's role is to coordinate and delegate tasks within the system. Although, the CalendarController's degree of coupling might raise due to its responsibility of delegating tasks, the systems total cohesion will increase by employing a central control.
Expert Doer Principle - Because each object has a direct link to the controller but limited communication with other objects, every object is limited to the tasks that the controller gives to it. This method makes it easier create objects that will do specific tasks in isolation.
Takes responsibility for the system's control. i.e.: Coordinates connections to database and updates mobile page display
Creates new trip and saves to the trip database. Note:Trip created only when the user first clicks on "Create Trip" to initialize process
Compiles, organizes and stores trip information
CalendarView Responsible for generating mobile UI page to be displayed to the user upon controller request
User
:CalendarController
:CreateTrip
:TripDB
:CalendarView
Establish connection
Returns Status
Clicks on "Create Trip"
Trip Creation
<<creates>>
Creates New Trip
Return status
Generates Calendar View Page upon Successful Trip Creation
Returns Generated Page
Displays Updated Page
Selects Date Range
Fetch Trip Data based on constraints
Returns Requested Data
Generates updated Calendar View Page
Returns Generated Page
Displays Updated Page

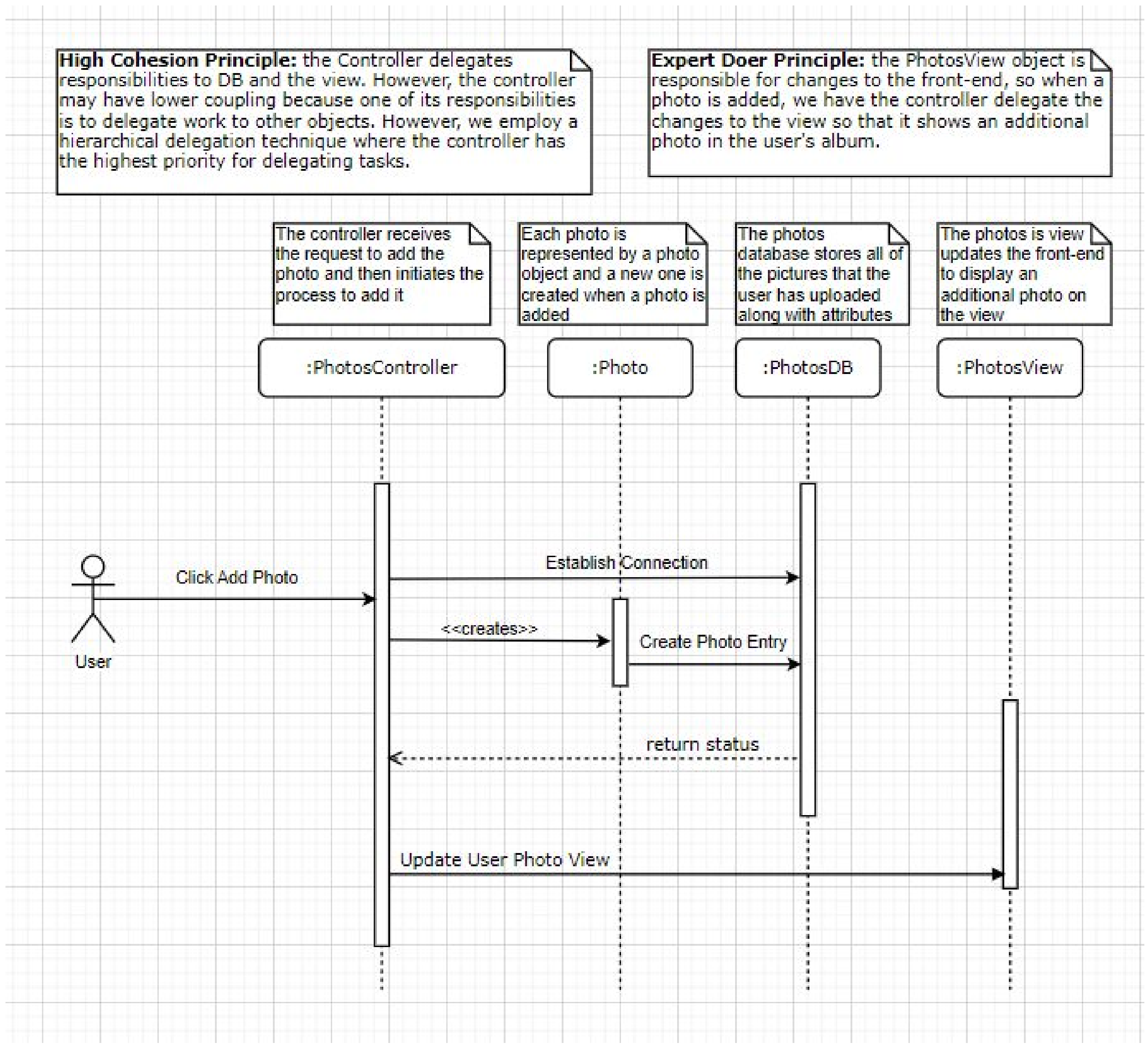
2. View Daily Summary


High Cohesion Principle - by utilizing a DayController to delegate tasks, we are minimizing the amount of responsibilities per object therefore increasing cohesion
Expert Doer Principle - by shortening the communication through the use of a controller, we have made an even distribution of tasks. Each object (AddEntry, DayView) knows its task and then completes it separately
Coordinates connections to database and delegates tasks
Creates entry(photo, journal, event, etc) and saves to database
Saves and stores information specific to that day
Responsible for generating UI for the Daily Summary page
:DayController
:AddEntry
:DayDB
:DayView
User
Establish connection
Clicks on a Day
Returns status
generateDayView()
Returns generated page
Displays page
Clicks to add entry
<<creates>>
createEntry()
Return status
generateDayView()
Returns Updates Page
Displays Updated Page

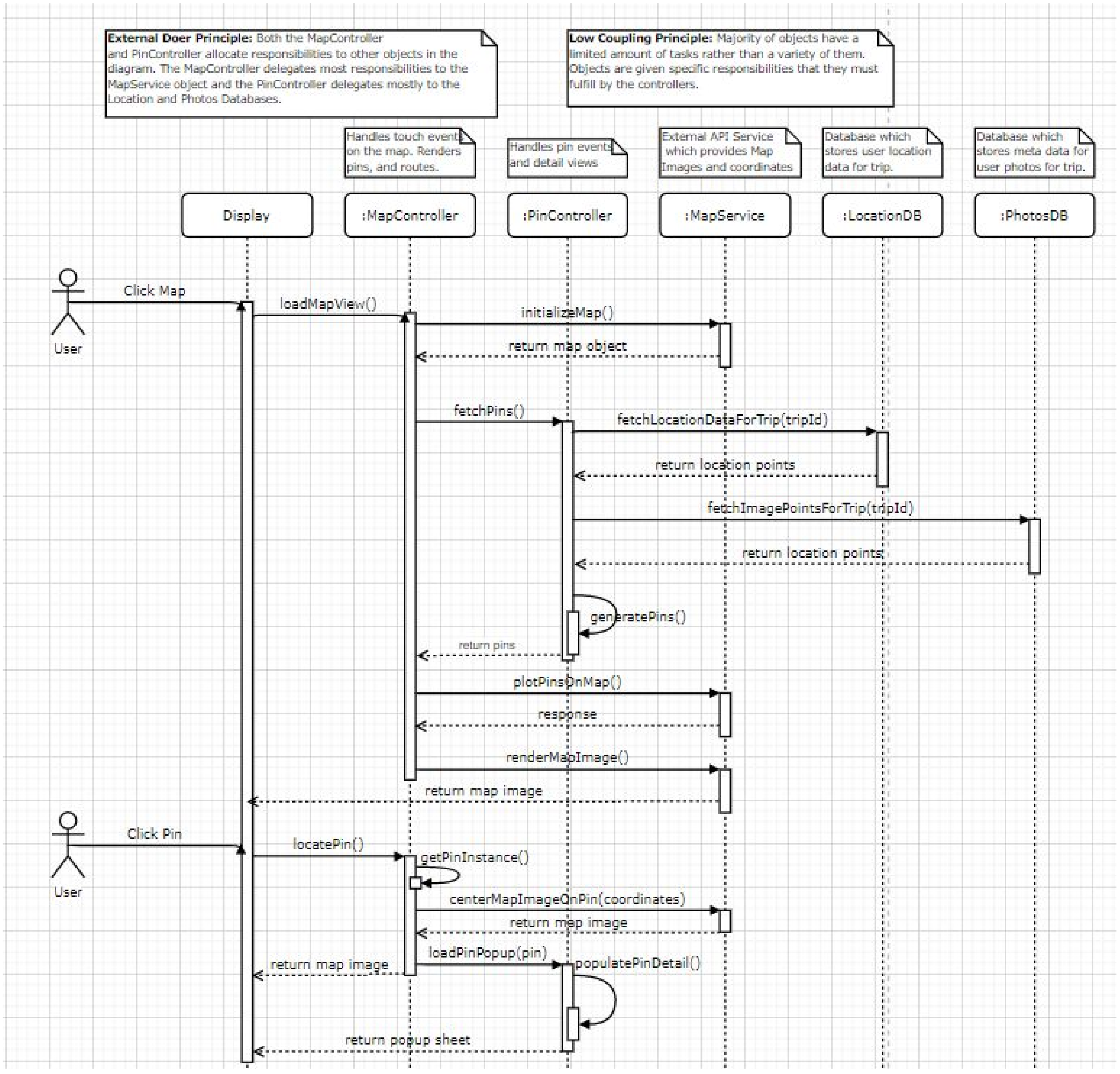
3. Add Journal Entry


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.
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.
Shows user concepts associated with journal entries, including actions (writing, editing, viewing)
:JournalController
:Journal
:JournalDB
:JournalView
User
Click Add Journal Entry
Establish Connection
<<creates>>
createjournal()
return status
loadjournalview()

4. Add Photo


High Cohesion Principle: the Controller delegates responsibilities to DB and the view. However, the controller may have lower coupling because one of its responsibilities is to delegate work to other objects. However, we employ a hierarchical delegation technique where the controller has the highest priority for delegating tasks.
Expert Doer Principle: the PhotosView object is responsible for changes to the front-end, so when a photo is added, we have the controller delegate the changes to the view so that it shows an additional photo in the user's album.
The controller receives the request to add the photo and then initiates the process to add it
Each photo is represented by a photo object and a new one is created when a photo is added
The photos database stores all of the pictures that the user has uploaded along with attributes
The photos is view updates the front-end to display an additional photo on the view
:PhotosController
:Photo
:PhotosDB
:PhotosView
User
Click Add Photo
Establish Connection
<<creates>>
Create Photo Entry
return status
Update User Photo View

5. View Map



External Doer Principle: Both the MapController and PinController allocate responsibilities to other objects in the diagram. The MapController delegates most responsibilities to the MapService object and the PinController delegates mostly to the Location and Photos Databases.

Low Coupling Principle: Majority of objects have a limited amount of tasks rather than a variety of them. Objects are given specific responsibilities that they must fulfill by the controllers.

Handles touch events on the map. Renders pins, and routes.

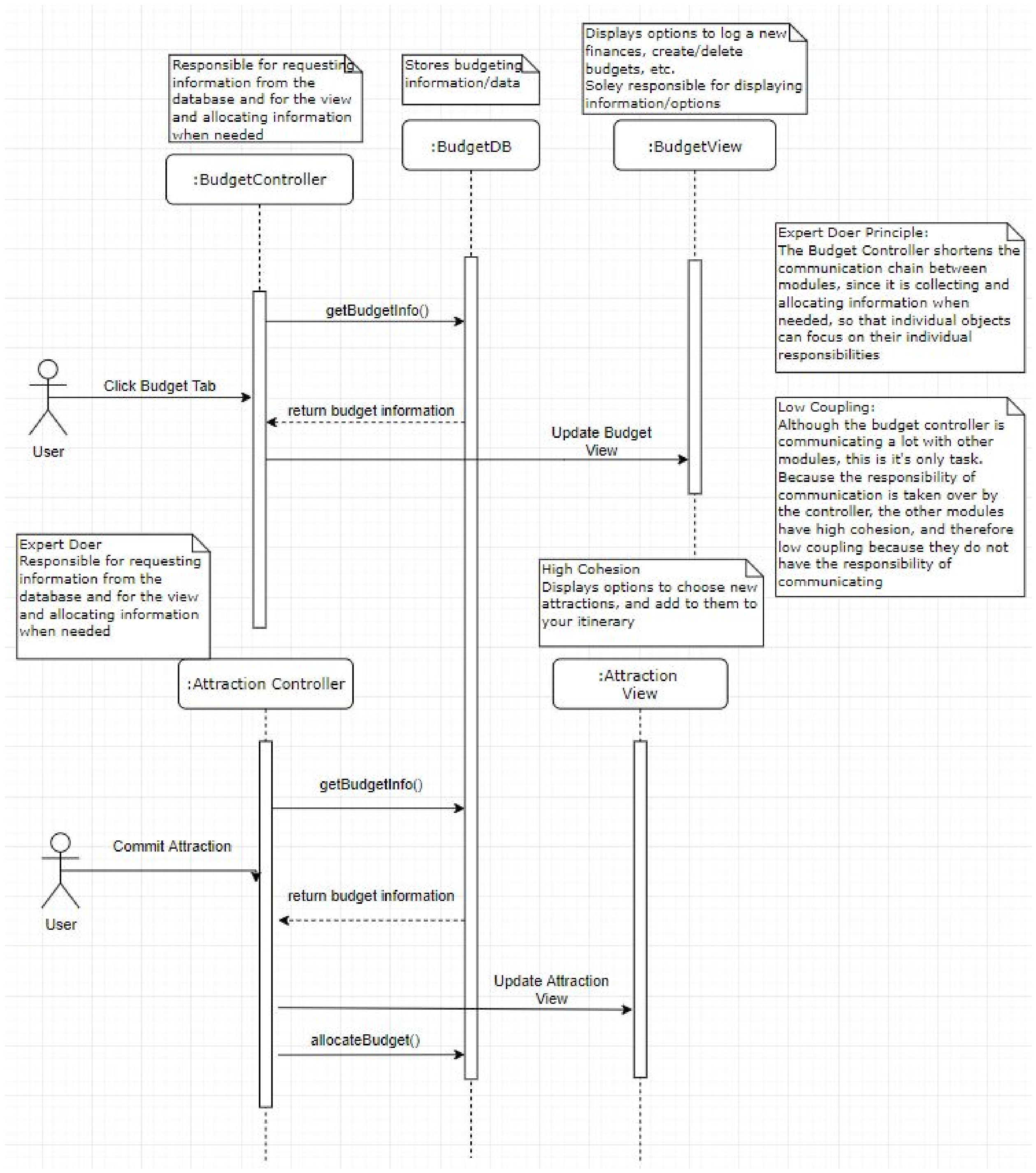
Handles pin events and detail views

External API Service which provides Map Images and coordinates

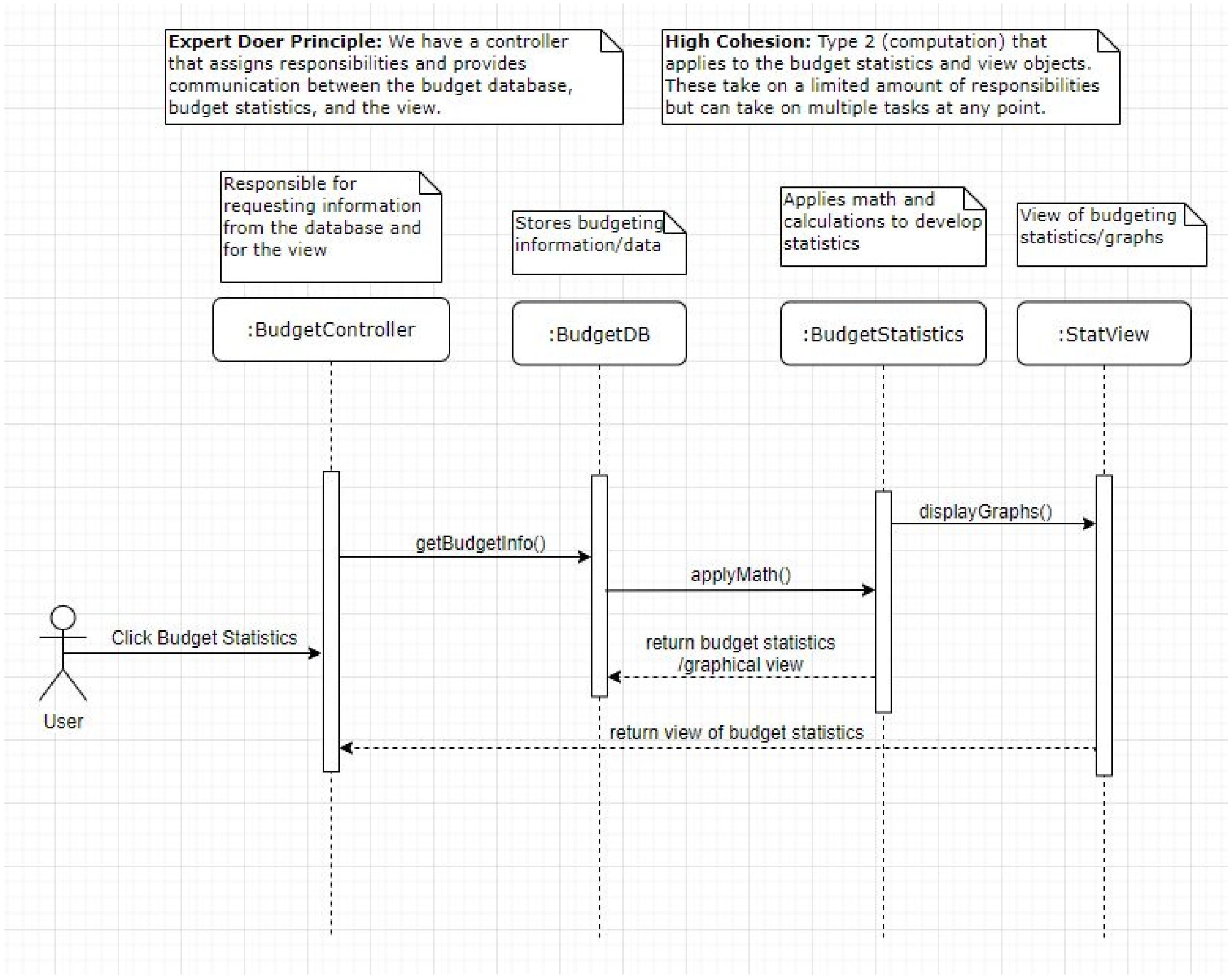
Database which stores user location data for trip.

Database which stores meta data for user photos for trip.

6. View Budget Overview

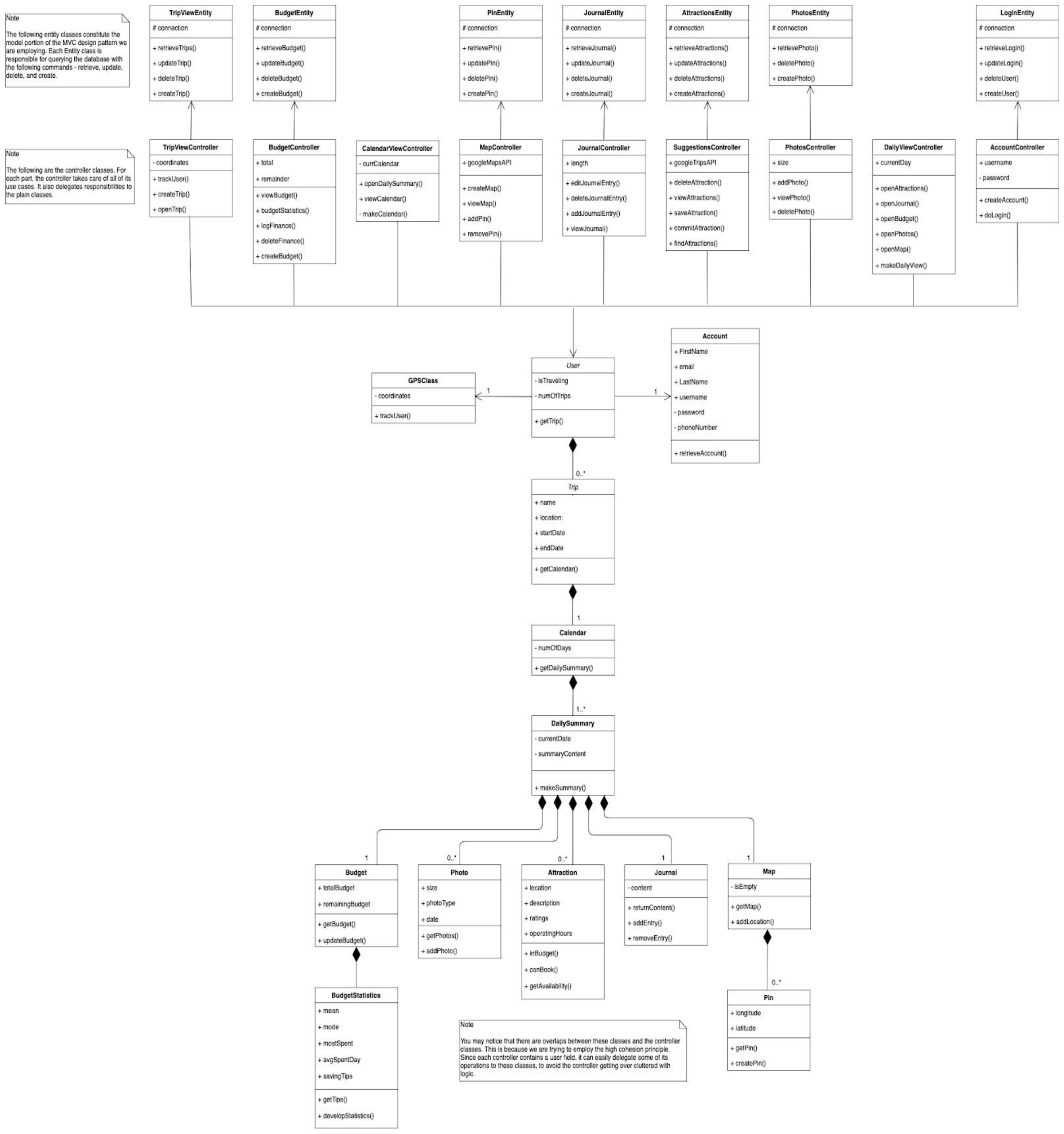

Responsible for requesting information from the database and for the view and allocating information when needed
Stores budgeting information/data
Displays options to log a new finances, create/delete budgets, etc. Soley responsible for displaying information/options
:BudgetController
:BudgetDB
:BudgetView
Expert Doer Principle: The Budget Controller shortens the communication chain between modules, since it is collecting and allocating information when needed, so that individual objects can focus on their individual responsibilities
getBudgetInfo()
Click Budget Tab
User
return budget information
Update Budget View
Low Coupling: Although the budget 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
Expert Doer Responsible for requesting information from the database and for the view and allocating information when needed
High Cohesion Displays options to choose new attractions, and add to them to your itinerary
:Attraction Controller
:Attraction View
getBudgetInfo()
Commit Attraction
User
return budget information
Update Attraction View
allocateBudget()

7. View Budget Statistics


Expert Doer Principle: We have a controller that assigns responsibilities and provides communication between the budget database, budget statistics, and the view.
High Cohesion: Type 2 (computation) that applies to the budget statistics and view objects. These take on a limited amount of responsibilities but can take on multiple tasks at any point.
Responsible for requesting information from the database and for the view
Stores budgeting information/data
Applies math and calculations to develop statistics
View of budgeting statistics/graphs
:BudgetController
:BudgetDB
:BudgetStatistics
:StatView
displayGraphs()
getBudgetInfo()
applyMath()
Click Budget Statistics
return budget statistics /graphical view
User
return view of budget statistics

2. Class Diagram and Interface Specification

A. Class Diagram


Note
The following entity classes constitute the model portion of the MVC design pattern we are employing. Each Entity class is responsible for querying the database with the following commands - retrieve, update, delete, and create.

Note
The following are the controller classes. For each part, the controller takes care of all of its use cases. It also delegates responsibilities to the plain classes.

Note
You may notice that there are overlaps between these classes and the controller classes. This is because we are trying to employ the high cohesion principle. Since each controller contains a user field, it can easily delegate some of its operations to these classes, to avoid the controller getting over cluttered with logic.

B. Data Types and Operation Signatures

1. TripViewEntity

a. **Attributes**

    i. connection : string - connection to needed database stored as URL

b. **Operation**

    i. retrieveTrips(username : string) : List - returns List of all trips for specified user
    ii. updateTrips(username : string, oldTrip : trip, newTrip : trip) : void - update old trip object for a certain user with new trip object
    iii. deleteTrip(username : string, delTrip : trip) : void - delete specified trip for specified user
    iv. createTrip(username : string, newTrip : trip) : void - inserts new trip into specified user profile

2. BudgetEntity

a. **Attributes**

    i. connection : string - connection to needed database stored as URL

b. **Operation**

    i. retrieveBudget(username : string, trip : trip) : budget - returns budget for one of trips of a specified user
    ii. updateBudget(username : string, trip : trip, oldBudget : budget, newBudget : budget) : void - replaces old budget with updated budget for a specified trip
    iii. deleteBudget(username : string, trip : trip, delBudget : budget) : void - deletes budget for a specified trip
    iv. createBudget(username : string, trip : trip, newBudget : budget) : void - save new budget details for a specified trip

3. TripViewController

a. **Attributes**

    i. coordinates : array - array storing two ints which represent latitude and longitude coordinates of user

b. **Operation**

    i. trackUser() : array - returns array of gps coordinates from google maps API
    ii. createTrip(username : string, newTrip : trip) : void - inserts new trip into specified user profile
    iii. openTrip(username : string, openTrip : trip) : void - goes to page which displays basic trip info

4. BudgetController

a. **Attributes**
   i. total : int - total budget in dollars user has set for trip
   ii. remainder : int - amount of remaining funds for user to spend from budget

b. **Operation**
   i. viewBudget(username : string, trip : trip, viewBudget : budget) : void - opens budget view page for the specified budget
   ii. budgetStatistics(username : string, trip : trip) : void - stores the budget stats calculated for a specific trip when change is made
   iii. logFinance(username : string, trip : trip, editBudget : budget) : budget - add a purchase to a specified budget and return updated object
   iv. deleteFinance(username : string, trip : trip, editBudget : budget) : budget - remove a purchase to a specified budget
   v. createBudget(username : string, trip : trip, newBudget : budget) : void - save new budget details for a specified trip

5. CalenderViewController

a. **Attributes**
   i. currCalendar : Calendar - A specific calendar for a user's trip

b. **Operation**
   i. viewCalendar(username : string) : void - returns calendar page for user with all trips organized by date
   ii. viewDailySummary(username : string) : void - fetches daily summary for user from the trip they are currently on
   iii. makeCalendar(username : string) : void - fills in calendar for specified user with all trips he has gone on

6. PinEntity

a. **Attributes**
   i. connection : string - connection to needed database stored as URL

b. **Operation**
   i. retrievePin(username : string, trip : trip) : pin - returns pin for one of trips of a specified user
   ii. updatePin(username : string, trip : trip, newPin : pin) : void - replaces old pin with updated pin for a specified trip
   iii. deletePin(username : string, trip : trip, delPin : pin) : void - deletes pin for a specified trip
   iv. createPin(username : string, trip : trip, newPin : pin) : void - save new pin details for a specified trip

7. MapController

a. **Attributes**
   i. googleMapsAPI : string - URL to query google maps API and get desired image of map from given coordinates

b. **Operation**
   i. createMap(pinArray : array, coordinates : array) : void - creates map around appropriate coordinates with pins user placed displayed
   ii. viewMap() : void - renders map image onto screen
   iii. addPin(username : string, trip : trip, delPin : pin) : void - deletes pin for a specified trip
   iv. removePin(username : string, trip : trip, delPin : pin) : void - deletes pin for a specified trip

8. JournalEntity

a. **Attributes**
   i. connection : string - connection to needed database stored as URL

b. **Operation**
   i. retrieveJournal(username : string, trip : trip) : journal - returns journal for one of trips of a specified user
   ii. updateJournal(username : string, trip : trip, newJournal : journal) : void - replaces old journal with updated journal for a specified trip
   iii. deleteJournal(username : string, trip : trip, delJournal : journal) : void - deletes journal for a specified trip
   iv. createJournal(username : string, trip : trip, newJournal : journal) : void - save new journal details for a specified trip

9. JournalController

a. **Attributes**
   i. length : int - number of characters of a journal entry

b. **Operation**
   i. viewJournal(username : string, trip : trip) : journal - returns journal for one of trips of a specified user and moves to view journal page
   ii. editJournalEntry(username : string, trip : trip, newJournal : journal) : void - replaces old journal with updated journal for a specified trip
   iii. deleteJournalEntry(username : string, trip : trip, delJournal : journal) : void - deletes journal for a specified trip
   iv. addJournalEntry(username : string, trip : trip, newJournal : journal) : void - save new journal details for a specified trip

10. AttractionEntity

a. **Attributes**
   i. connection : string - connection to needed database stored as URL

b. **Operation**
    i. retrieveAttraction(username : string, trip : trip) : attraction - returns attraction for one of trips of a specified user
    ii. updateAttraction(username : string, trip : trip, newAttraction : attraction) : void - replaces old attraction with updated attraction for a specified trip
    iii. deleteAttraction(username : string, trip : trip, delAttraction : attraction) : void - deletes attraction for a specified trip
    iv. createAttraction(username : string, trip : trip, newAttraction : attraction) : void - save new attraction details for a specified trip

11. SuggestionsController

a. **Attributes**
    i. googleTripsAPI : string - URL to query google trips API and get recommended attractions around that area

b. **Operation**
    i. viewAttraction(username : string, trip : trip) : attraction - returns attraction for one of trips of a specified user
    ii. commitAttraction(username : string, trip : trip, newAttraction : attraction) : void - replaces old attraction with updated attraction for a specified trip
    iii. deleteAttraction(username : string, trip : trip, delAttraction : attraction) : void - deletes attraction for a specified trip
    iv. saveAttraction(username : string, trip : trip, newAttraction : attraction) : void - save new attraction details for a specified trip
    v. findAttraction(coordinates : array) : arraylist - returns list of attractions based on given coordinates by sending request to google trips API

12. PhotosEntity

a. **Attributes**
    i. connection : string - connection to needed database stored as URL

b. **Operation**
    i. retrievePhoto(username : string, trip : trip) : photo - fetches photo being requested from database
    ii. createPhoto(username : string, trip : trip, newPhoto : photo) : void - uploads new photo object to database for storage
    iii. deletePhoto(username : string, trip : trip, delPhoto : photo) : void - deletes selected photo from database

13. PhotosController

a. **Attributes**
    i. size : - number of photos being managed Array - tells us the number of photos the user is storing in a daily summary instance

b. **Operation**

i. addPhoto(username : string, trip : trip, dailySummary : DailySummary, newPhoto : photo) : void - sends request to upload the new image to the trips library
ii. viewPhoto(username : string, trip : trip, dailySummary : DailySummary, photo : Photo) : photo - requests to view a photo in higher definition
iii. deletePhoto(username : string, trip : trip, delPhoto : photo) : void - sends request to delete selected photo

14. User

a. Attributes

i. isTraveling : boolean - true if user is currently in a trip, false otherwise
ii. numOfTrips : int - total number of trips user has been on

b. Operation

i. getTrip(username : string, tripName : string) : trip - returns trip object of trip user has selected

15. Trip

a. Attributes

i. name : string - name of trip created by user
ii. location : string - city trip takes place in
iii. startDate : datetime - start date of the trip as selected by user
iv. endDate : datetime - end date of rip as selected by user

b. Operation

i. getCalendar() : void - takes user to calendar page custom for current trips dates

16. GPSClass

a. Attributes

i. coordinates : array - array of two int values that return the coordinates of where a trip took place

b. Operation

i. trackUser() : array - returns array of gps coordinates from google maps API

17. Account

a. Attributes

i. FirstName : string - first name of user
ii. LastName : string - last name of user
iii. email : string - email id of user for contact info/notifications
iv. username : string - username for user to login to account
v. password : string - password for user to login to account

   vi. phoneNumber : int - phone number of user for contacting/push sms notifications

b. **Operation**
   i. retrieveAccount() : Account - returns an instance of the user's account, encapsulating all of the information relevant to logging in

18. Calendar

a. **Attributes**
   i. numofDays : int - number of days in the month being displayed

b. **Operation**
   i. getDailySummary() : DailySummary - returns instance of daily summary

19. DailySummary

a. **Attributes**
   i. currentDate : datetime - current date the app is being loaded up on
   ii. summaryContent : String - amalgamation of all of the main features of app user has added data too that day

b. **Operation**
   i. makeSummary(username : String) : String - returns data

20. Budget

a. **Attributes**
   i. totalBudget : int - total budget in dollars user has set for trip
   ii. remainingBudget : int - amount of remaining funds for user to spend from budget

b. **Operation**
   i. getBudget() : budget - returns instance of budget
   ii. updateBudget(newBudget : budget) - updates given budget with new object to replace old info

21. Photo

a. **Attributes**
   i. size : array - array of x and y dimensions of the number of pixels in the image
   ii. photoType : string - file type of picture that has been uploaded by user
   iii. date : datetime - date and time photo was taken

b. **Operation**
   i. getPhotos() : photo - returns instance of photo class

22. Attraction

a. **Attributes**
   i. location : string - city location is a part of

ii. description : string - detailed description of the event the attraction and what it entails
iii. ratings : int - rating out of 5 based on other tourists reviews who have visited the attraction
iv. operatingHours : string - time the attraction is open on different days of the week

b. Operation

i. inBudget() : boolean - checks if attraction can fit into budget of user for the trip selected and returns false if it goes over budget and true otherwise
ii. canBook() : boolean - checks if user can book attraction and returns true if there is still space available/it is open and false otherwise
iii. getAvalibility() : string - returns availability reservation timings of the attraction selected

23. Journal

a. Attributes

i. content : String - text user types up in journal

b. Operation

i. returnContent() : void - displays text has typed up with background of handwritten journal
ii. addEntry(String addContent) : void - adds additional text to the content
iii. removeEntry(int start, int end) : void - removes text from content from specified positions

24. Map

a. Attributes

i. isEmpty : boolean - returns true if no pins are on map, false otherwise

b. Operation

i. getMap() : map - returns instance of map class

25. BudgetStatistics

a. Attributes

i. mean : int - average money spent per day
ii. mode : int - day with most money spent
iii. mostSpent : datetime - most user has spent on one day
iv. avgSpentDay : int - average user spends per day during a trip
v. savingTrips

b. Operation

i. getTips() : arraylist - returns an arraylist of potential advice that can be given for different budgets and spending habits

ii. developStatistics() : calculates all the values of the above attributes and stores them accurately updating whenever changes are made

26. Pin

a. **Attributes**
   i. longitude : int - longitudinal coordinate of pin
   ii. latitude : int - latitudinal coordinate of pin

b. **Operation**
   i. getPin() : Pin - returns the instance of a Pin object
   ii. createPin() : void - creates a pin at the instance's longitude and latitude coordinates

C. Traceability List | Report 1

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/Trip Summary:** Our group decided to add this interaction diagram because we thought that viewing the calendar and trip 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 trip summary and calling 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

3. System Architecture + System Design

A. 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’.

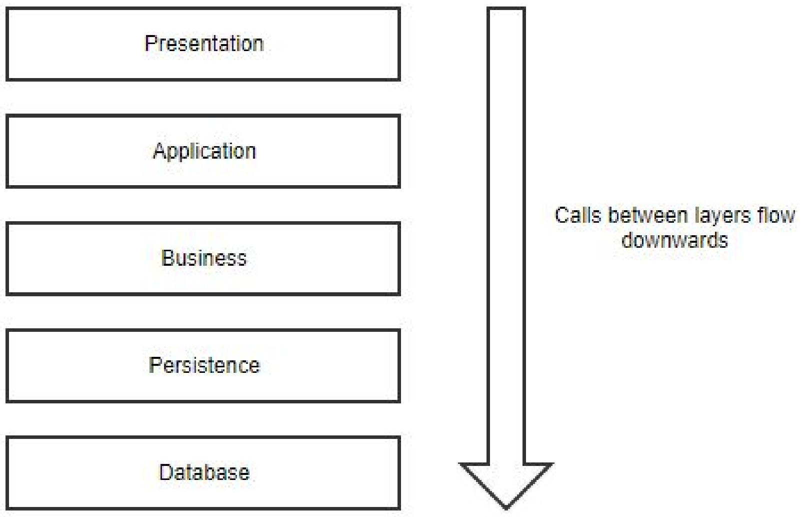


Figure 1

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 trip 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.

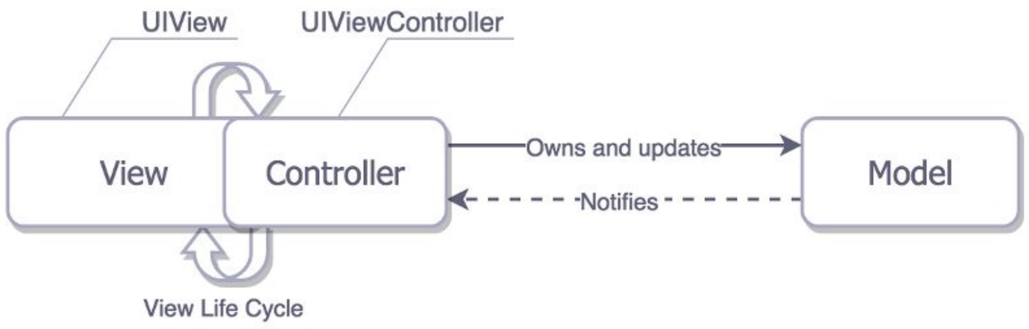


Figure 2

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.

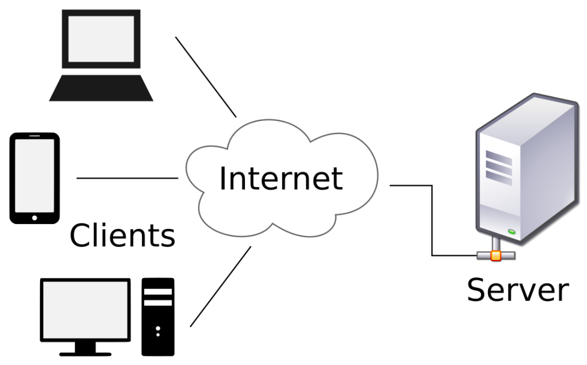
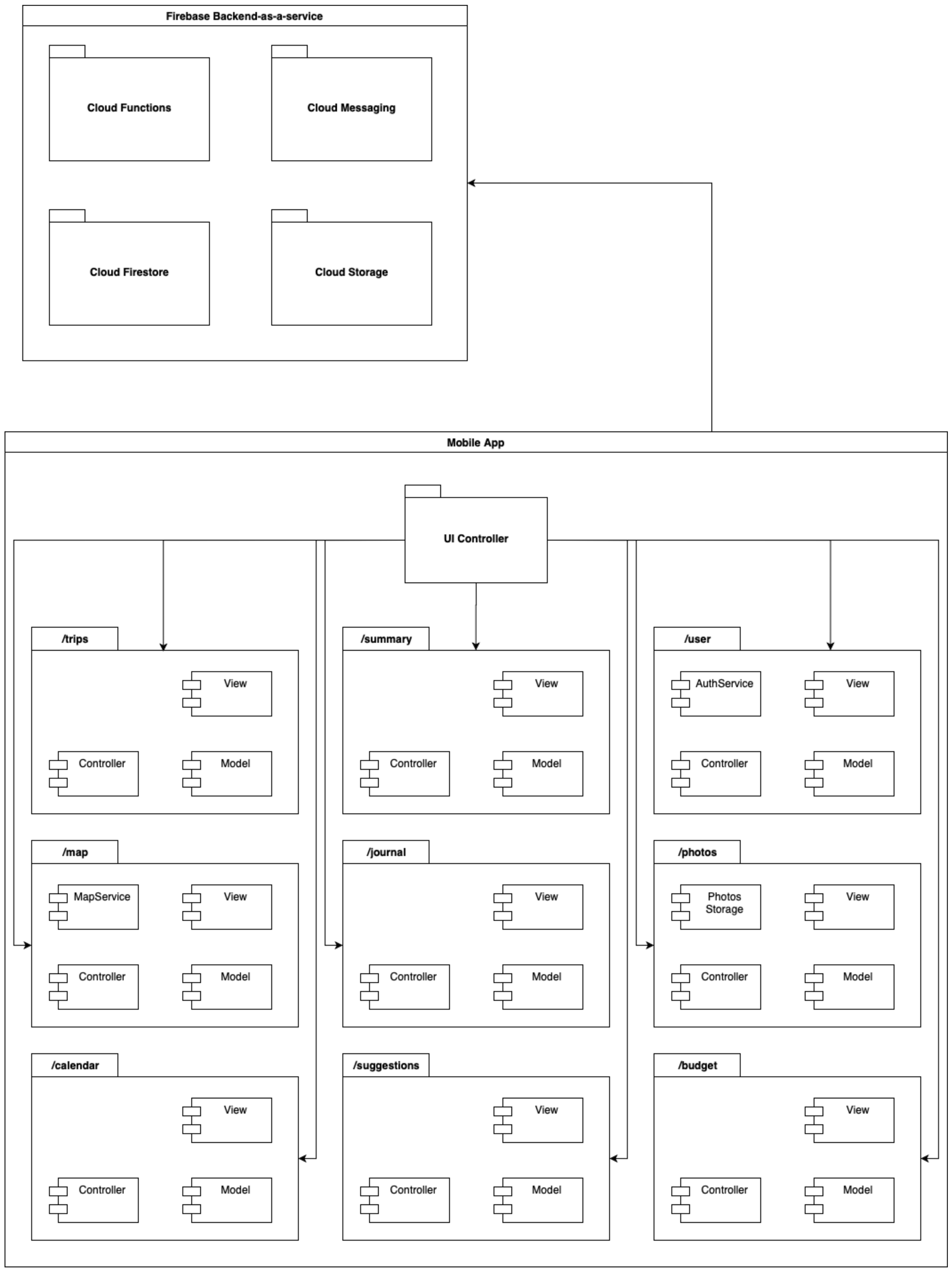


Figure 3

B. 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 provided 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 provided 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

C. 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.

D. 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

E. 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.

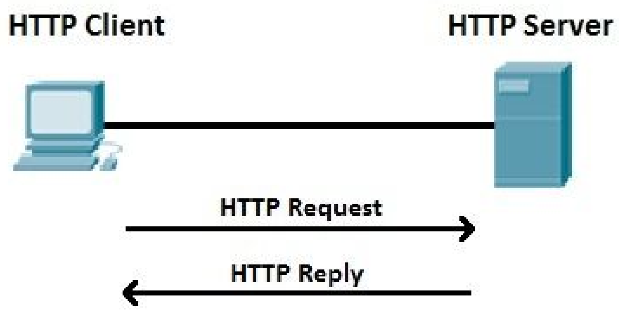


Figure 3

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

4. 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

Each trip will have multiple day objects that will be displayed to the user in the form of a calendar. Therefore, we have decided to use an **array** to hold all of these day objects. An array is the best way to store these objects since it offers the fastest way to retrieve information since we will know the index of the desired object when the user clicks it. There is no need to waste time and search for it in the array. Additionally, since we are having the user input the start and end of their trip, we can create an array of the correct size. We will not need to resize or grow the array.

2. DailySummary

Each daily summary will have an **arraylist** of photos that were uploaded to that day of the trip which allows for easy access/sorting/inserting of the list to create multiple organized views of the photos depending on how the user wants to view the photos from their day. Also, each object of dailysummary will have a map object as a canvas to display where the user went all day.

3. 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).

4. 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.

5. **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).

6. **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.

7. **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).

8. **Budget**

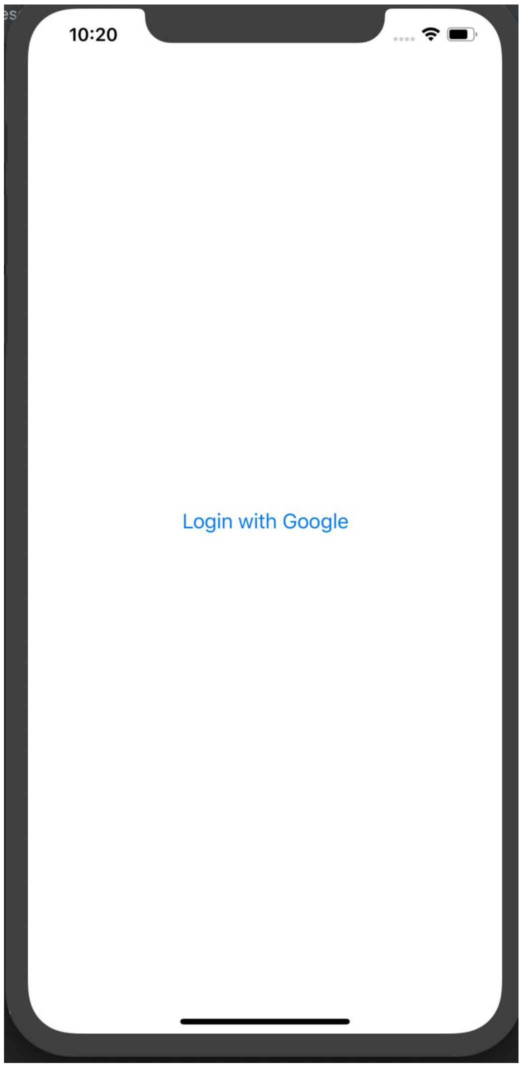
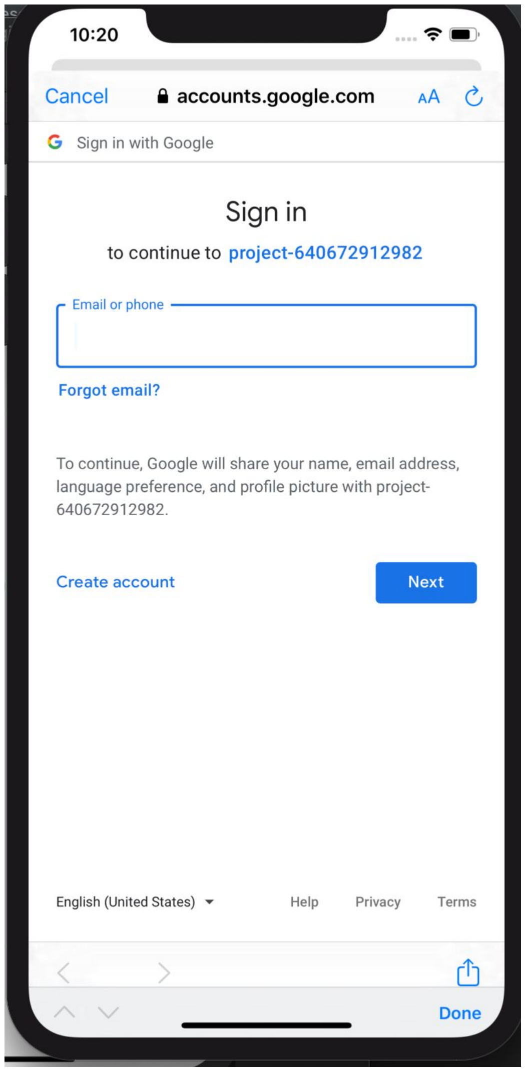
   The budget statistics will use two **arraylists** to store tips and tricks or encouragement to continue saving, that are presented at random for the user. If the user is low on their budget, it will use the list that represents tips and tricks and if the user has not used much of their budget, then the 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.

9. **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.

5. UI Design + Implementation

A. Log-In Page

Landing Page	Google Sign-In
	

10:20

Cancel accounts.google.com AA

Sign in with Google

Sign in

to continue to project-640672912982

Email or phone

Forgot email?

To continue, Google will share your name, email address, language preference, and profile picture with project-640672912982.

Create account Next

English (United States) Help Privacy Terms

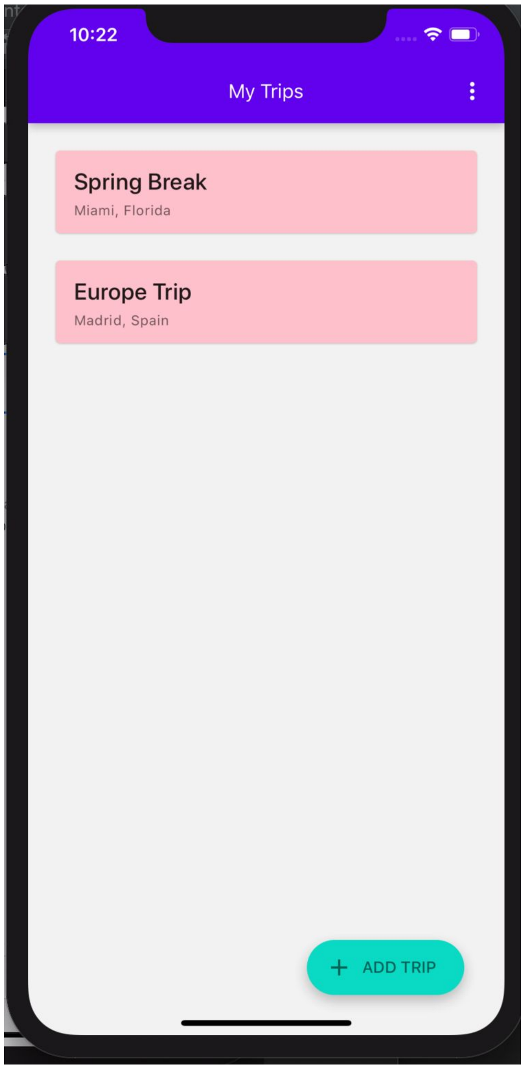
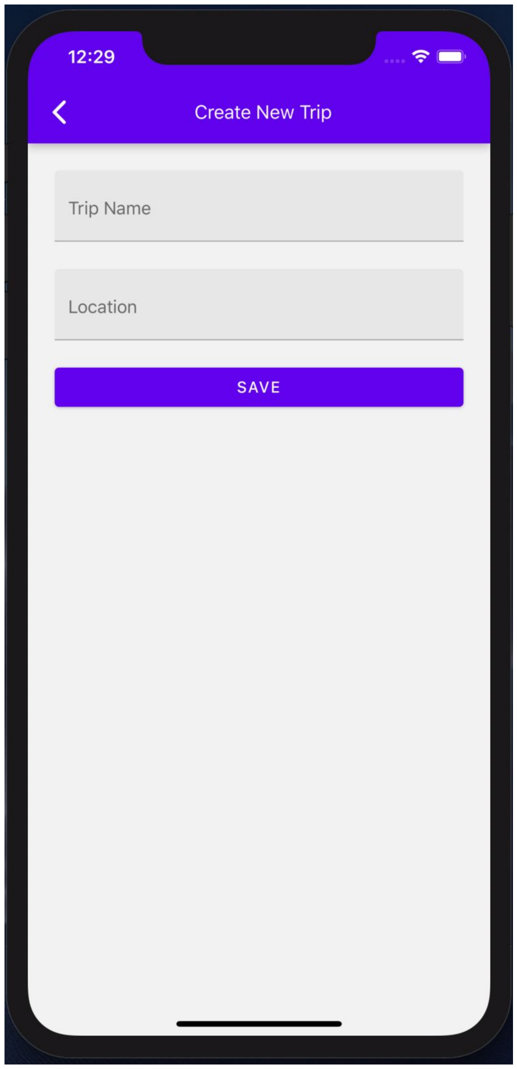
Done

As presented above, the user is able to log into the application through a Google account. This allows their information to be identified through the Google platform, and for them to bypass creating a new account.

1. **Landing Page**
   The user will be presented with this page in order to sign in. We anticipate to add further details which will display the essential “Log-In with Google” option.

2. **Google Sign-In**
   Provided through the Google platform, TripDiary is able to be accessible (on Android and iOS) through a user's already existing Google account.

B. User Trips Page

My Trips	Create Trip
 My Trips; Spring Break, Miami, Florida; Europe Trip, Madrid, Spain; ADD TRIP	 Create New Trip; Trip Name; Location; SAVE

Every user will encounter their 'My Trips' screen. This is the location in which they will be able to interact with different trips the user has logged previously. Although this is a raw implementation from the mock-up, we anticipate on enhancing the UI to fit a common theme throughout the application.

1. **My Trips**
   Through separate blocks, the user is able to distinguish between different trips. They are then able to click on their trips, to be led to their 'Trip Summary/Calendar' page, which is in the process of being implemented.

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

C. Unimplemented Features

Considering features unimplemented, the application has the following remaining items:

1. Calendar
2. Daily Summary
3. Journal
4. Map
5. Photos
6. Suggestions
7. Budget

This section was meant to be a discussion of already implemented features, which are presented above. Once the interface for the remaining features are implemented, it will be included in the demo/next report. **Regardless, we plan on following the mock-ups presented in Report 1.**

6. Design of Tests

A. Test Cases

1. TC-1

Test-case Identifier:	TC-1
Use Case Tested:	UC-1: CreateAccount
Pass/fail Criteria:	The test passes if the user enters valid credentials for username and password and the account does not already exist
Input Data:	Text
Test Procedure	**Expected Result**
Step 1: Type in a username which already exists	System sends an alert saying account exists
Step 2: Type in an invalid username	System sends an alert specifying the valid format for a username
Step 3: Type in a valid username and password which do not exist	System returns to the login view so that the user logs in

2. TC-2

Test-case Identifier:	TC-2
Use Case Tested:	UC-2: 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

3. TC-3

Test-case Identifier:	TC-3
Use Case Tested:	UC-3: 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

4. TC-4

Test-case Identifier:	TC-4
Use Case Tested:	UC-5: AddPhoto
Pass/fail Criteria:	The test passes if the user enters an image with the correct format
Input Data:	File
Test Procedure	**Expected Result**
Step 1: User uploads an unsupported file from the file system	System sends an alert saying file is of the wrong format
Step 2: User uploads a .jpg, .png, or supported file type	System adds the photo to the list of photos in the day for that trip and re-renders the photos view

5. TC-5

Test-case Identifier:	TC-5
Use Case Tested:	UC-7: AddPin
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 25 foot radius

Input Data:	User presses the add 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

6. TC-6

Test-case Identifier:	TC-6
Use Case Tested:	UC-11: SaveAttraction
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

7. TC-7

Test-case Identifier:	TC-7
Use Case Tested:	UC-12: CommitAttraction
Pass/fail Criteria:	The test passes if the user commits an attraction successfully
Input Data:	User presses the commit button

Test Procedure	Expected Result
Step 1: User commits an attraction that has been approved for their budget	System successfully commits an attraction
Step 2: User commits an attraction that is not within their budget	System displays a warning saying the attraction is not in the user's budget

8. TC-8

Test-case Identifier:	TC-8
Use Case Tested:	UC-20: ViewCalendar/TripSumary
Pass/fail Criteria:	Test passes if the user navigates to the calendar view, inputs desired dates and all day objects for the selected dates are correctly displayed with the associated data structures followed by the correct trip summary/summaries below.
Input Data:	Date Range

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 the last day of the month (based on the date range)
Step 2: User specifies date range of their desired trip(s)	System generates all contained trip summaries beneath the calendar

9. TC-9

Test-case Identifier:	TC-9
Use Case Tested:	UC-21: ViewDailySummary
Pass/fail Criteria:	Test passes if the user navigates to the calendar view, inputs desired dates, selects a specific day and the Daily Summary is correctly displayed with all associated data (i.e. photos, journal entries, suggestions, maps)
Input Data:	Date Range, Date Selection

Test Procedure	Expected Result
Step 1: User clicks on “calendar” **Step 2:** User specifies date range of their desired trip(s)	System generates a calendar view of days that begin at the beginning of the month and extend to the last day of the month (based on the date range)
Step 3: User clicks on a specific day of a trip	After a selection of a trip day, the associated daily summary will be displayed

10. TC-10

Test-case Identifier:	TC-10
Use Case Tested:	UC-22: Journaling
Pass/fail Criteria:	Test passes if the user navigates to the dailyView for a specific trip day, selects Journal Entries and all journal entries are correctly displayed
Input Data:	Date selection

Test Procedure	Expected Result
Step 1: User navigates to the dailyView page **Step 2:** User clicks on “Journal Entries” **Step 3:** User clicks on a specific Journal Entry	System generates a dailyView page After the user selects a specific journal entry, the associated journal entry will appear along with any photos, drawings or videos that were added previously

11. TC-11

Test-case Identifier:	TC-11
Use Case Tested:	UC-23/UC-24: Add/Delete/EditJournalEntry
Pass/fail Criteria:	Test passes if the user makes a change to a journal entry and the change is reflected in the Daily Summary page
Input Data:	N/A

Test Procedure	Expected Result
Step 1: User makes a change to a journal entry from a specific day **Step 2:** User navigates to Daily Summary for the same day **Step 3:** User clicks on “Journal Entries”	System generates the list of journal entries after “Journal Entries” is selected and the journal entry that was added/edited will appear or the journal entry that was deleted will no longer appear

12. TC-12

Test-case Identifier: TC-12 Use Case Tested: UC-18/UC-19 Pass/fail Criteria: The system displays the budget correctly based on user input Input Data: User’s budget	
Test Procedure	**Expected Result**
Step 1: User inputs valid budget or changes	System generates a display for the budget
Step 2: User inputs invalid budget or changes	System generates error

B. Test Coverage

1. TC-1 - Information for Creating an Account

Define the following Equivalence Classes for each input:

Class A	Class B	Class C
Consists of valid characters; Length is greater than 0; There is no existing username in the database	There is already a username in the database	Consists of some invalid characters not recognized by the system

First, we will show that the classes are disjoint. It is easy to show A is disjoint with B and C because class A consists of valid inputs and classes B and C consist of invalid inputs. B and C are disjoint because if there existed an input x common to both B and C, then that would imply that there exists a valid username in the database with invalid characters, which we know is impossible because class A prevents such a valid input by how it’s defined. Therefore, the classes A, B, and C are all disjoint.

It was important to show that A, B, and C are disjoint because this proves that A, B, and C are equivalence classes. This has high coverage because we may just select a candidate from A, B, and C, and by definition of equivalence classes we would expect the results to be uniform over all candidates in A, B, and C. The set A covers all valid inputs, and the sets B and C cover the two cases which would lead to failure.

To increase test coverage, we should also employ **boundary testing** for this test case. We have not yet covered the case where the input length is zero - just an empty string. In this case, the system would send an alert saying that the username cannot be empty. With this last form of testing, we should have covered all invalid and valid cases.

2. TC-2 - Logging Into the System

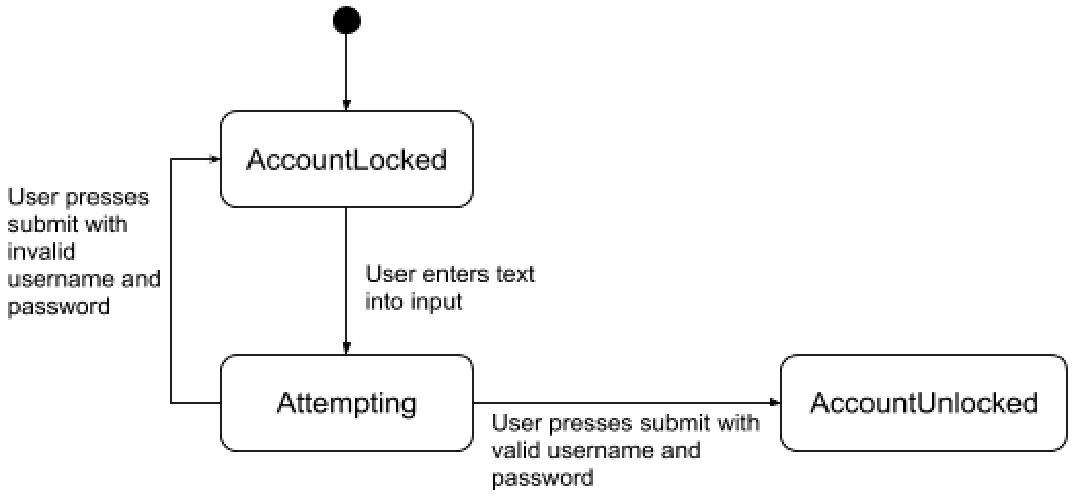
State-based Testing for Logging In:

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


AccountLocked
User presses submit with invalid username and password
User enters text into input
Attempting
User presses submit with valid username and password
AccountUnlocked

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

3. TC-3 - 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.

4. TC-4 - Adding a Photo

Define the following equivalence classes for adding a photo:

Class A	Class B
System supports this file type	System does not recognize this file type

For a user to add a photo, he or she needs to select a photo from their device's file system. However, it is possible for a user to accidently upload something such as a textbook pdf which would cause an error. We do not need to make separate cases for all unsupported file types because the error is the same in all cases.

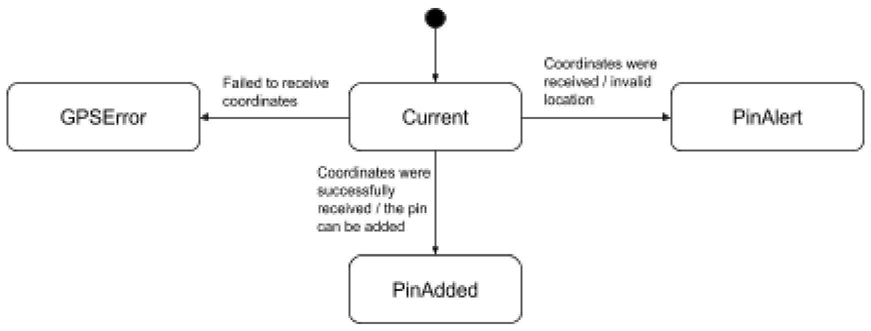
As a result, we can separate inputs into two classes - those which cause an error and those which are uploaded successfully. To support this test case, we can write two unit tests. One of which checks that the expected result for a valid input is success, and another which checks for failure when an unsupported file is uploaded.

5. TC-5 - Adding a Pin to the Map

State-based Testing for 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


GPSError
Failed to receive coordinates
Current
Coordinates were received / invalid location
PinAlert
Coordinates were successfully received / the pin can be added
PinAdded

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. Since we are also using the GPS component of our system, these tests will probably have to be integration tests.

6. TC-6 Saving an Attraction

Class A	Class B
System has not saved attraction	System has already saved attraction

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 two input groups, one valid and one invalid. We can write two unit tests to test whether saving an attraction that has not been saved produces a valid result, and whether saving an attraction that has already been saved produces an invalid result.

7. TC-7 - Committing an Attraction

Class A	Class B
Attraction is within a user's budget	Attraction is outside user's budget

For a user to commit to an attraction, he or she needs to press the commit button on an attraction. This action can be treated as a set input, since the attraction can be either inside the user's budget or outside the user's budget.

Since the user's budget is a range of values, we can split the input parameters into two input groups, one valid and one invalid. We can write two unit tests to test whether committing an attraction that is within a user's budget produces a valid result, and whether committing an attraction that is outside the user's budget produces an invalid result.

8. TC-8 - 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 must input a start date and an end date that will specify the date range 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 are no days that contain a daily summary and there will be no trip summaries beneath 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 cluster of dates that contain daily summaries 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 date range. In this case, the desired outcome is to have more than one cluster of days that have daily summaries as well as multiple trip summaries under the calendar.

9. TC-9 - 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.

10. TC-10 - 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 a Calendar View that contains one or more trips, then click on a day that has a daily summary and then select "Journal Entries". 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.

11. TC-11 - Verifying Journal Entry Changes

Class A	Class B	Class C
A journal entry that was deleted no longer appears in the daily view summary	A journal entry that was created now appears in the daily view summary	A journal entry that was modified(in any way) now shows the edits in the journal entries in the daily view summary

There is more than one way for a user to add, edit or remove a journal entry. This is why it is essential that however it is done, the changes are reflected in the Daily View Summary from the Calendar View. Class A, B and C fully test any changes that could be made to the journal entries of any given day on the Trip Calendar. Class A is the result of a test done to ensure that a deleted entry appears nowhere in the journal entry list from the daily view summary; and also none of its data are found anywhere else in the system. Class B is a result of a test done to make sure that a new journal entry is now shown in the daily view summary (with all of its contents). Class C is the result of a test done to verify that modifications made to a journal entry is represented in the daily view summary (and that nothing else in the journal entry changed except the edits that were first made).

12. TC-12: Viewing Budget Changes

Class A	Class B
Budget includes negative numbers	Budget includes non-number values

When the user inputs a new budget or changes the current budget they have, the budget must be considered a valid budget. To be considered valid, the budget cannot include negative numbers and the budget cannot include non-number values. Both Class A and Class B cover any of the errors that the user could have while entering or changing the current budget. Class A prevents any errors with mathematical calculations for both committing to attractions (subtracting from the budget) and for developing budget statistics. Class B confirms that the budget is an actual number and that numerical calculations can be performed on the budget value. For when the user uses up certain amounts of the budget, to display the value of the budget remaining, we cannot have a negative or non-numerical value. By keeping the original budget numerical and positive, we also prevent these future errors.

C. Integration Testing Strategy

As per our initial decision of using a layered architecture for our software project, our sub-groups focus is on creating individual components which implement our various use cases. Each component has a root controller and it’s own set of models and views. When these components are integrated together, we implement integration tests to ensure that the controllers work together as intended in the scope of the application as a whole.

1. App → Authentication → Trips

This integration test ensures that once our application loads, it calls the authentication component, and upon a successful login, it then successfully loads the Trips component of our application.

2. Trip → Authentication

This integration test ensures that if a user clicks on “Sign Out” from the Trips view, that the application then logs them out and sends them back to the authentication component.

3. Trips → Calendar

This integration test ensures that from the Trips screen, that the calendar component is loaded to appropriately display the dates and other relevant information about the trip.

4. Trips → Trip Summary

This integration test ensures that from the Trips view, if a user clicks on a trip object, that our application is able to successfully call on the Trip Summary component which handles data loading of the trip itself and sets up the appropriate navigation bar for the various trip detail views.

5. **Calendar→ Daily View Summary**

    This integration test ensures that from the Calendar view, when a user clicks on a specific date, that the Daily View Summary component is loaded for the date that has been requested.

6. **Daily View Summary → Suggestions**

    This integration test ensures that from the Daily View Summary screen, that the suggestions component is loaded to appropriately display the suggestions generated for the trip.

7. **Trip Summary→ Photos**

    This integration test ensures that when a user clicks on the Photos tab on the Trip Summary navigation bar, that the Photos component is loaded for the trip that has been requested.

8. **Trip Summary→ Maps**

    This integration test ensures that when a user clicks on the Maps tab from the Trip Summary navigation bar, that the Map component is loaded for the trip that has been requested.

9. **Trip Summary→ Budget**

    This integration test ensures that when a user clicks on the Budget tab from the Trip Summary navigation bar, that the Budget component is loaded for the trip that has been requested.

10. **Trip Summary→ Journal**

    This integration test ensures that when a user clicks on the Journal tab from the Trip Summary navigation bar, that the Journal component is loaded for the trip that has been requested.

7. Project Management + Plan of Work

A. Merging Contributions

1. Editing Rotations

An issue encountered from the first report was that all merging was a single group member's responsibility. To alleviate and delegate this responsibility, we created a system to combine reports which involved each subgroup. Considering Report 2 was split into 3 parts, each subgroup had editing responsibilities for an individual part. Below, is a visual of the system created:

Part #	Subgroup Editing	Students
1	Budget / Suggestions	Vincent Chan, Nisha Bhat
2	Map / Photos	Sam Minkin, Yash Shah, Gaurav Seth
3	Calendar / Journal	Sam Zahner, Kinjal Patel, Jonathon Banks

2. Formatting Consistency

Adopted to ensure consistency throughout the report, a formatting schema was created for each editing subgroup to follow:

Size	Part	Color
20	Main Heading	🟦
16	Subheading	🟪
13	Minor Heading	⬛

B. Project Coordination + Progress Report

1. Use Cases Implemented

From our report 1, we have a total of 24 use cases that need to be implemented. Currently, UC-1: Create Account, UC-2: Login, and UC-3: Create Trip, have all been implemented in our app.

2. Progress

a. What is functional?

Firebase is set up and works currently, but we have not saved information or retrieved information from it yet. The user can create a new account or login to their current account through Google, which we use Google's login API to accomplish. After the user has logged in, a screen is displayed of all of their trips. At the bottom of the page, an add trip button is present, which when clicked, opens a new create trip page. In this create trip page, the user is prompted with the name and location of their trip. As of now, the current trips that the user has are hard-coded. In the future, we will have to pull this information from Firebase. What happens after the person has created a trip is now being worked on.

b. What is being worked on?

The calendar, trip summary, and daily summary user interface pages are the next step. However, the map, budget, photos, and journal are also being coded by their respective subgroup to increase efficiency.

3. Project Coordination: Meetings

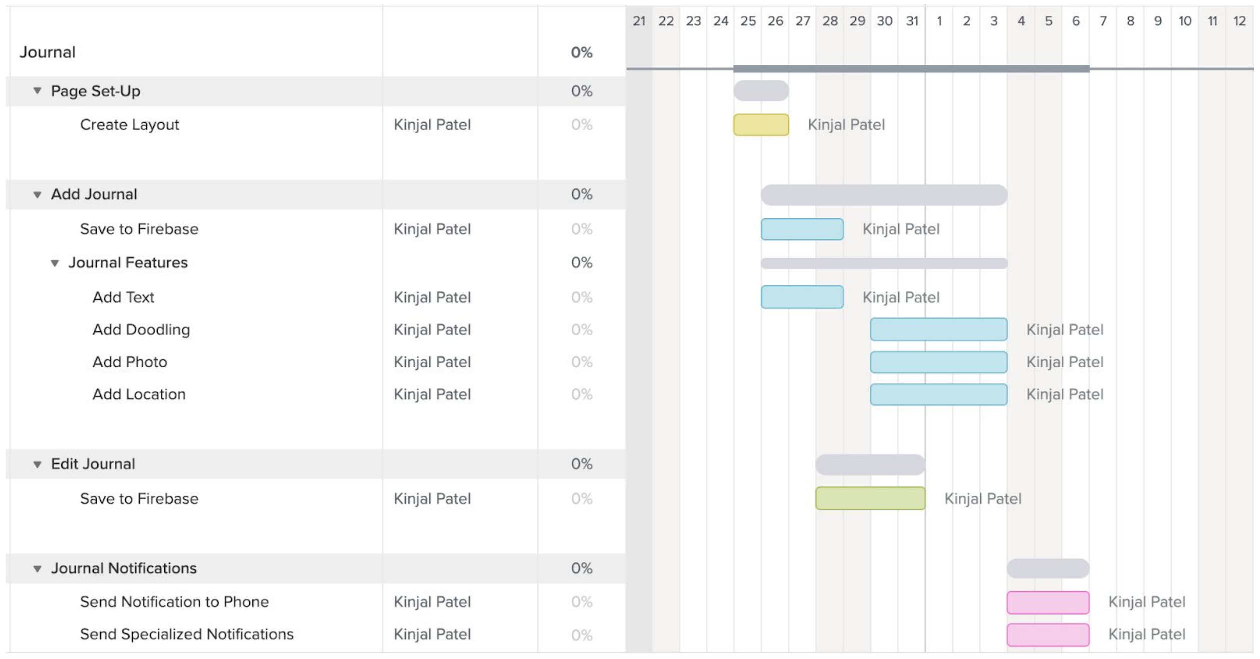
To coordinate how the project was going to be initially set up through ReactNative and Firebase, we met as an entire team. Additionally, since editing and merging documents for the first report was not distributed well, we discussed how to distribute it better. Below are links to our WhenToMeet and our meeting agenda.

1. Meeting Agenda/Notes
2. WhenToMeet

C. Plan of Work

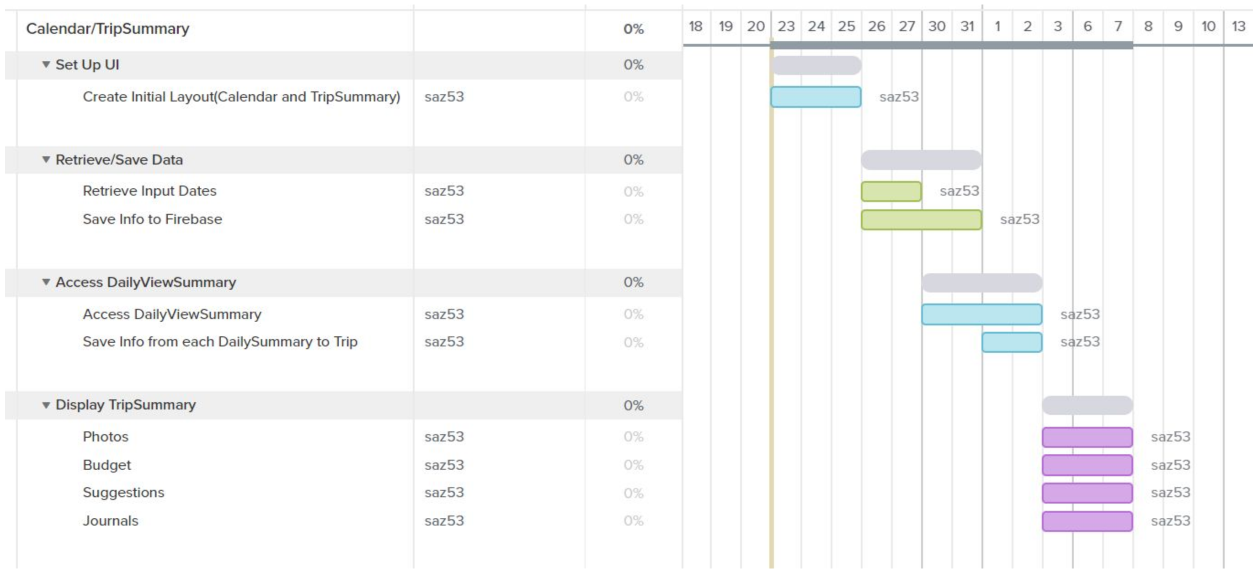
Sub-Problem	Team
Editing/Report Combining	Rotating Subgroups (explained above)
Calendar/Journal	Kinjal Patel, Samuel Zahner, Jonathan Banks
Map/Photos/Transportation	Gaurav Sethi, Samuel Minkin, Yash Shah
Suggestions/Budget	Vincent Chan, Nisha Bhat

1. Journal



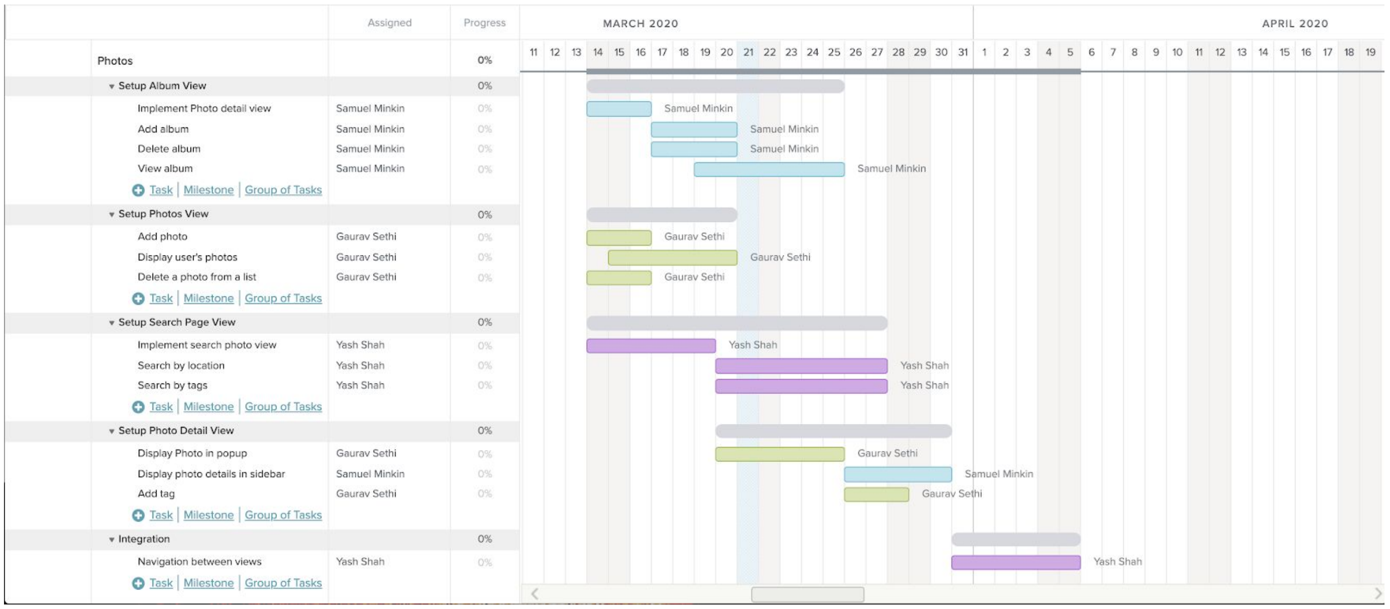
Journal		0%
▾ Page Set-Up		0%
Create Layout	Kinjal Patel	0%
▾ Add Journal		0%
Save to Firebase	Kinjal Patel	0%
▾ Journal Features		0%
Add Text	Kinjal Patel	0%
Add Doodling	Kinjal Patel	0%
Add Photo	Kinjal Patel	0%
Add Location	Kinjal Patel	0%
▾ Edit Journal		0%
Save to Firebase	Kinjal Patel	0%
▾ Journal Notifications		0%
Send Notification to Phone	Kinjal Patel	0%
Send Specialized Notifications	Kinjal Patel	0%

2. Calendar/TripSummary



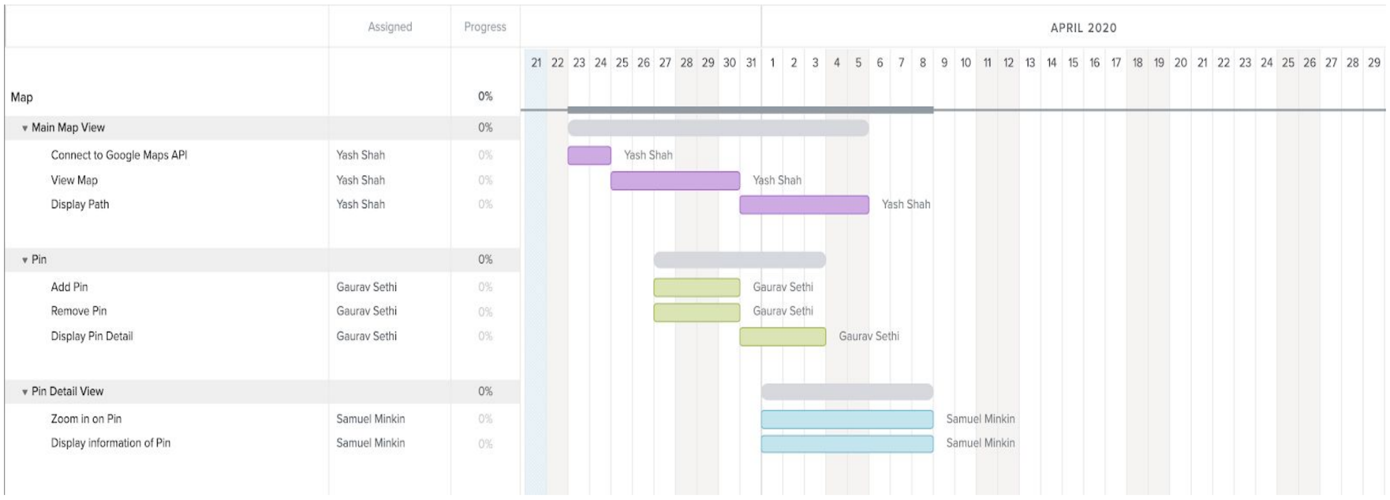
Calendar/TripSummary		0%
▾ Set Up UI		0%
Create Initial Layout(Calendar and TripSummary)	saz53	0%
▾ Retrieve/Save Data		0%
Retrieve Input Dates	saz53	0%
Save Info to Firebase	saz53	0%
▾ Access DailyViewSummary		0%
Access DailyViewSummary	saz53	0%
Save Info from each DailySummary to Trip	saz53	0%
▾ Display TripSummary		0%
Photos	saz53	0%
Budget	saz53	0%
Suggestions	saz53	0%
Journals	saz53	0%

3. Photos



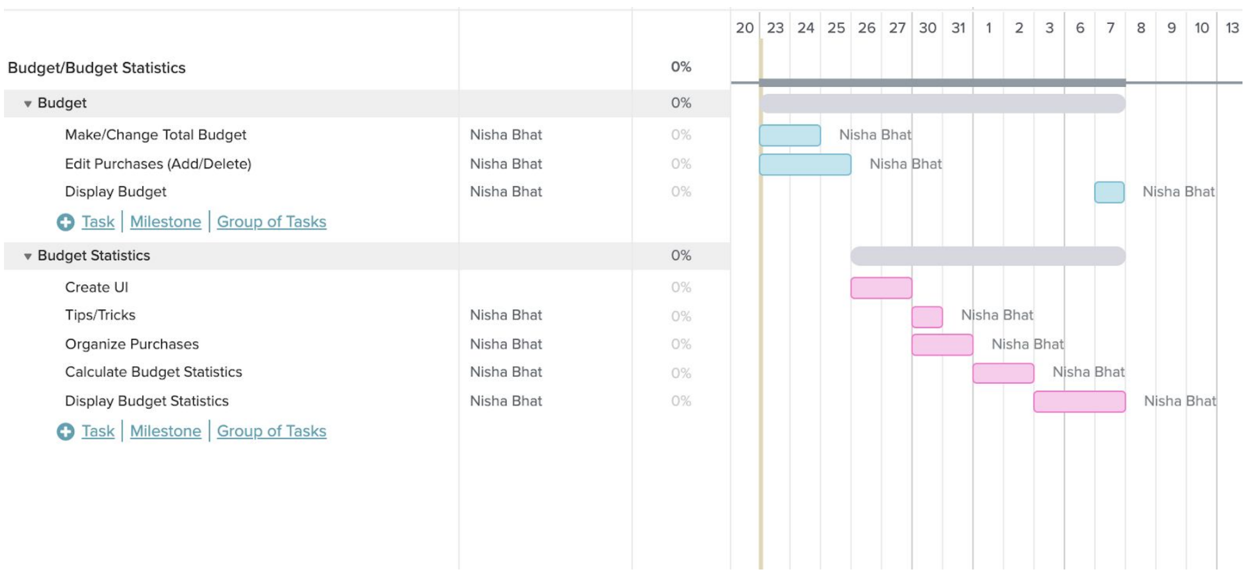
	Assigned	Progress
Photos		0%
▾ Setup Album View		0%
Implement Photo detail view	Samuel Minkin	0%
Add album	Samuel Minkin	0%
Delete album	Samuel Minkin	0%
View album	Samuel Minkin	0%
Task | Milestone | Group of Tasks		
▾ Setup Photos View		0%
Add photo	Gaurav Sethi	0%
Display user's photos	Gaurav Sethi	0%
Delete a photo from a list	Gaurav Sethi	0%
Task | Milestone | Group of Tasks		
▾ Setup Search Page View		0%
Implement search photo view	Yash Shah	0%
Search by location	Yash Shah	0%
Search by tags	Yash Shah	0%
Task | Milestone | Group of Tasks		
▾ Setup Photo Detail View		0%
Display Photo in popup	Gaurav Sethi	0%
Display photo details in sidebar	Samuel Minkin	0%
Add tag	Gaurav Sethi	0%
Task | Milestone | Group of Tasks		
▾ Integration		0%
Navigation between views	Yash Shah	0%
Task | Milestone | Group of Tasks		

4. Map



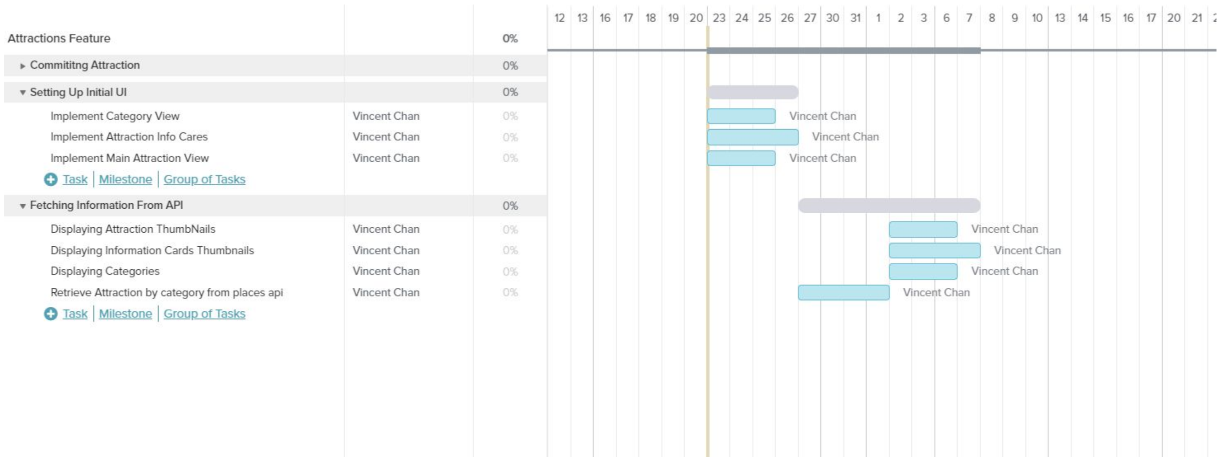
	Assigned	Progress
Map		0%
▾ Main Map View		0%
Connect to Google Maps API	Yash Shah	0%
View Map	Yash Shah	0%
Display Path	Yash Shah	0%
▾ Pin		0%
Add Pin	Gaurav Sethi	0%
Remove Pin	Gaurav Sethi	0%
Display Pin Detail	Gaurav Sethi	0%
▾ Pin Detail View		0%
Zoom in on Pin	Samuel Minkin	0%
Display information of Pin	Samuel Minkin	0%

5. Budget/Budget Statistics



Budget/Budget Statistics		0%
▾ Budget		0%
Make/Change Total Budget	Nisha Bhat	0%
Edit Purchases (Add/Delete)	Nisha Bhat	0%
Display Budget	Nisha Bhat	0%
Task | Milestone | Group of Tasks		
▾ Budget Statistics		0%
Create UI		0%
Tips/Tricks	Nisha Bhat	0%
Organize Purchases	Nisha Bhat	0%
Calculate Budget Statistics	Nisha Bhat	0%
Display Budget Statistics	Nisha Bhat	0%
Task | Milestone | Group of Tasks		

6. Attractions



Attractions Feature		0%
▸ Commititng Attraction		0%
▾ Setting Up Initial UI		0%
Implement Category View	Vincent Chan	0%
Implement Attraction Info Cares	Vincent Chan	0%
Implement Main Attraction View	Vincent Chan	0%
Task | Milestone | Group of Tasks		
▾ Fetching Information From API		0%
Displaying Attraction ThumbNails	Vincent Chan	0%
Displaying Information Cards Thumbnails	Vincent Chan	0%
Displaying Categories	Vincent Chan	0%
Retrieve Attraction by category from places api	Vincent Chan	0%
Task | Milestone | Group of Tasks		

D. 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

		4. BudgetStatistics
	Vincent Chan	Attractions 1. Attraction 2. Attraction Entity 3. Attraction Controller

Integration

Individual group members will be responsible to integrate their own work within the application. To easily allow for code sharing and updates, our team has used GitHub. For the application to connect with each other, the student responsible for ‘Daily Summary’ will coordinate the connection within the application (since you are able to reach different pages though that screen). We will set up different branches for each subgroup to work on and subgroups will only commit their updates to their branches. Then, each group will code review another group's branch before merging the branch to the master branch. This will allow a fresh set of eyes to look at the code and the structure of the code. That will be our process with integrating the branches. When it comes to integration tests, the groups whose code is being merged will be responsible for conducting tests for all components that connect to their code.

References

Figure 1: https://dzone.com/articles/software-architecture-the-5-patterns-you-need-to-k
Figure 2: https://www.intellectsoft.net/blog/mobile-app-architecture/
Figure 3: https://en.wikipedia.org/wiki/Client%E2%80%93server_model
Figure 4: https://www.csitquestions.com/what-is-http-and-https-explained-ccna-tutorial/
System Architecture (Layered):
https://medium.com/@mlbors/architectural-styles-and-architectural-patterns-c240f7df88a0