# Harrison Smith (12000624)
# Software design Document

**Mobile apps development 2017**

**Introduction**

Almost everyone has had the issue of being on a holiday without knowing what to do or where to go. You've just arrived in a beautiful part of the country, but apart from travel guides (which can be spotty), there's not much to point you to the best local experiences. You could try and hunt down some locals, and decipher their instructions about exactly where to go (if they'll even give up their knowledge of the best spots). The difficulty and effort to find an adventure that will be the right one for you is pretty high.

Or maybe you're at home and are looking for something fun to do with your friends on the weekend. But organising a trip like this with your friends can be difficult. On top of finding a place to go that all of your friends will enjoy, there's always a flurry of messages, screenshots and web links that go along with it. Organising the event, confirming that people are coming, making sure people are available, rescheduling and so on is all part of the difficulty.

Adventure Squad covers you and your friends in both of these scenarios.

**1.1 Application Vision**

Adventure Squad gives you all the fun of going on new adventures with your friends and takes away the hassle of finding and planning them. Adventure Squad is an app that provides its users with a simple and effective way to find outings ('Adventures') that are aligned with their interests, and take into account how close an adventure is. An Adventure is any sort of activity that usually involves getting outside and experiencing nature at a relatively low cost. The app provides an easy list view of adventures that are relatively close to you, and that are tailored to your preferences, including the type of activity, difficulty and length of time.

The app will allow single users to plan adventures, and will also allow a user to form a squad made up of their friends and plan an adventure with their squad. Squads are usually formed amongst in-person friends or existing friend groups, but the app will allow you to search usernames to add an 'anonymous' user to your squad. The app is not meant to be used as a 'full-blown' social network, but instead is meant to complement existing frameworks and provide a better way to plan Adventures. The app would also ideally allow users to continue to tailor their own experience; as they go on more adventures the application would try and predict what sort of adventures a user prefers to go on over other adventures.

The idea is to focus an entire business around these apps.

- Show adventures that match with your
- Build a social network of adventurers who love to go on adventures

**1.2 Scope**

Each item within the scope is grouped under a certain 'level' of scope. The purpose of this grouping is to provide a better estimate of all of the items that need to be completed.

**Minimum scope –** To complete these would be the minimum features required to have the application meet its goals. These items are essential and must be completed to have any application at all.

- **1. Login & Registration**
    - Users can log in and register for the application with an email and password
    - Users can log out
- **2. Database**
    - Adventure details are stored in a database, accessed through the internet
    - Can retrieve a list of adventures given certain constraints
    - Adventure images are stored in a database (blob), accessed through the internet
- **3. Adventure Feed**
    - User can browse a list of adventures
    - User can view details of an adventure
- **4. My Trips**
    - User can add a single adventure to their plan
    - User must specify time and date for adventure
    - User can view a list of their trips

**'Full' Scope –** All items within this scope are not absolutely required, but the completion of these items are recommended for the application to achieve it's full potential. These should be prioritised after all items within the minimum scope are completed.

- **1. Login & Registration**
    - Users can allow location permission
    - Users can choose their preferred Adventure types
- **2. Database**
    - User details (username, adventure preferences etc) are stored in the database as an individual entry
    -
- **Adventure Feed**
    - List of adventures is found related to the user's preferences
    -
- **Squads**
    - User can create a squad and plan an adventure for their squad
    - User can join and leave multiple squads
    - Users are notified when an adventure is added to their squad
    - User can invite people to their squad by a simple username search
- **User Details**

    - User can edit their own profile
    - User can upload an image of themselves for their profile
-

**Optional Features ('Could')**

- **Login**
    - Facebook authentication and account creation
- **Adventure Feed**
    - User rating of adventures
    -
- **My Trips**
    - Google maps navigation intent starting
- **Adventure Feed**
    - Allow users to create & upload their own adventures for others to experience
    - User can do a more detailed search for items
- **My Trips**
    - User is notified when a planned adventure is going to start soon
- **Squads**
    - User can add people to their squad by having them scan a QR code with their phone (ZXing API)
- **UI**
    - Customised UI drawable objects e.g. rounded corners
- **User Details**
    - Preferences system to further tailor items such as notification preferences, adventure feed preferences

**Out of Scope**

- Inter-user messaging system
- User reporting other users system

**1.3 Document Version History**

5/06/2017 – Created document, added headings

5/06/2017 - Added Introduction, application vision from previous document. Started creating the scope and feature summary from this.

5/06/2017 – Added API section, refined scope and added minimum scope / optional /

5/06/2017 – Refined feature list, scope, software flow diagram

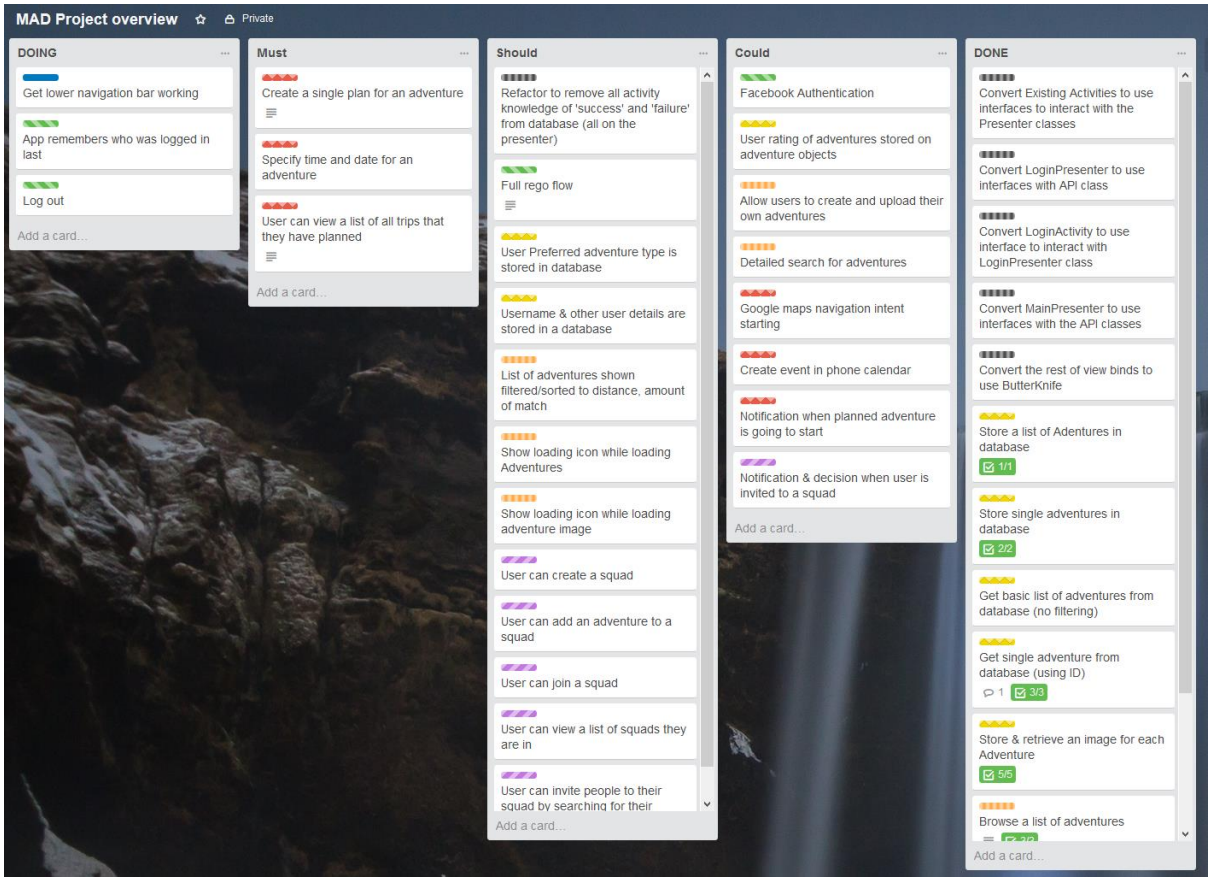
5/06/2017 – Updated scope to better match trello board

## 2 Functionality Overview

This section details the various functions that a user can perform within the application.

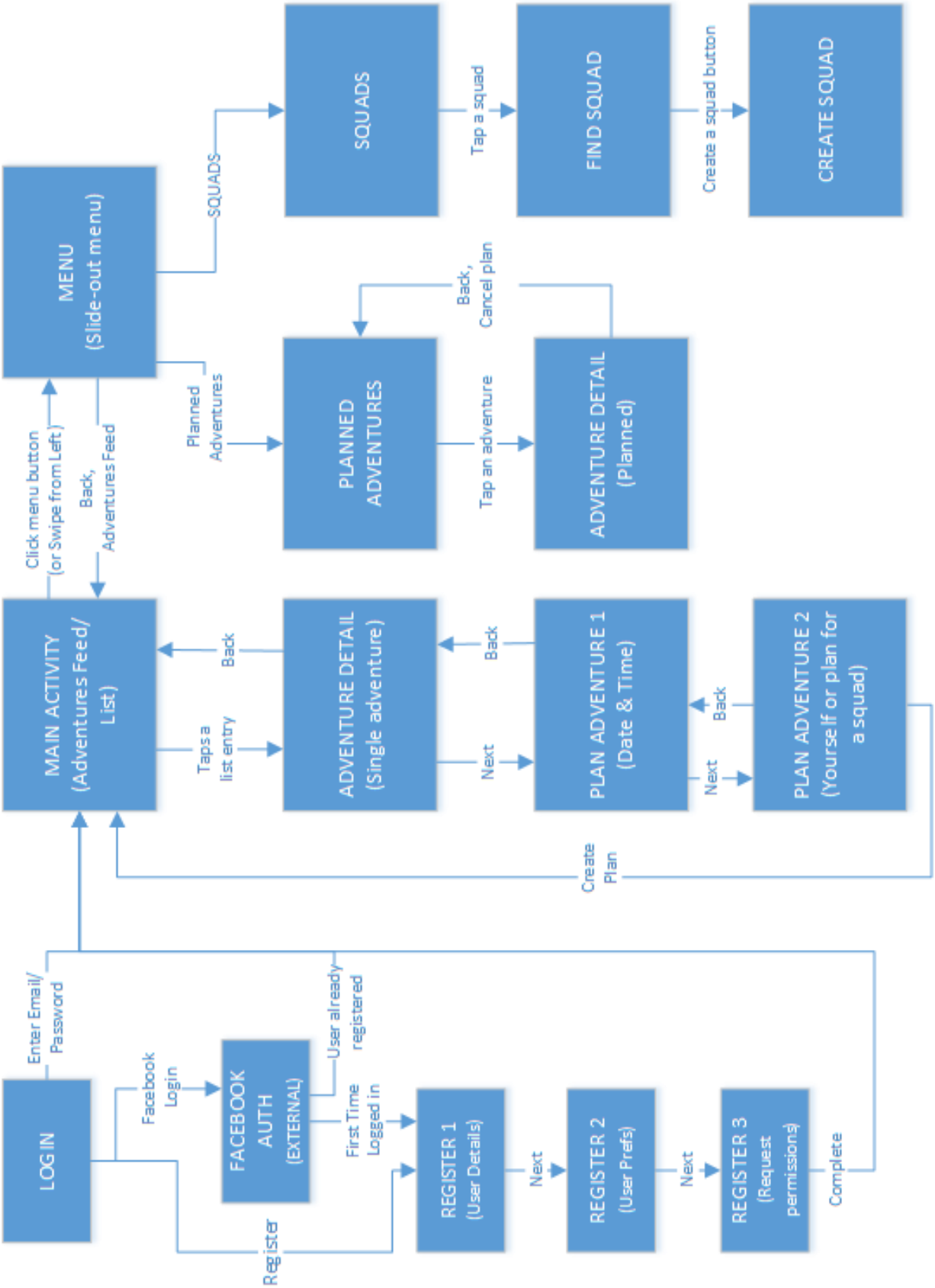
### 2.1 Feature Summary

- **Registration**
    - A user can choose to register from the initial 'login' page
    - User is then taken through a flow of registration which includes inputting information such as:
        - Email address
        - Password (and second time for verification)
        - User name (Must be unique)
        - Full name (optional)
        - Various adventure types that are suited to them
    - Registration should also request certain permissions such as location to allow filtering of trips
- **Login**
    - A user will be able to enter their email address and password in the login screen, and then tap a 'Login' button
        - If the user enters either their email or password incorrectly, the screen will change to show a message reflecting this
    - The user will also be able to tap a 'Facebook Login' button to login with a facebook account instead of email/password. This will take the user through a custom Facebook login flow.
- **Adventure Feed and Adventure Detail**
    - User will be able to browse a list of adventures that are displayed as a 'card-style' interface
        - These will display distance in km as well as difficulty as a 'loading bar' style graphic
    - User will be able to tap on an individual card to go to a single adventure screen detail
- **My adventures**
    - User will be able to see a list of all adventures that they have planned, both personally and for any that are part of a squad
    - User can click a single object to view the details of this object
- **Squads**
    - User will be able to view a list of all of the squads that they are in
    - User can join a new squad from this screen, or create one
    - User can select a squad to view the squad details including members, previous and planned adventures




## 2.3 Application Flow Diagram

Diagram showing the basic flow of how people will use the app.




**3 Design**

### 3.1 Overall Components overview

Generally, the application has similar basic components to other, similar event planning & database-driven applications.

- Login screen with registration flow
- Main screen with a 'cards'-style RecyclerView
- Slide-out left-hand-side menu for navigation
- Multiple other views for lists of objects within the application, e.g. Squads, Planned Adventures, as well as appropriate detailed views for each individual object

### 3.2 Mockups

**Description (with Figure 1)**

Primarily, Adventure Squad shows a collection of new adventures that a user can plan (fig 1. *'Adventures'*). Displayed adventures are selected according to each user's preferred activities (e.g. hiking, cycling, mountain biking, kayaking), and can also take into account the maximum travel distance to an adventure. Users can also refine the list by searching for a particular adventure, or filtering the list to only include certain adventures depending on time, distance or the type of activity.

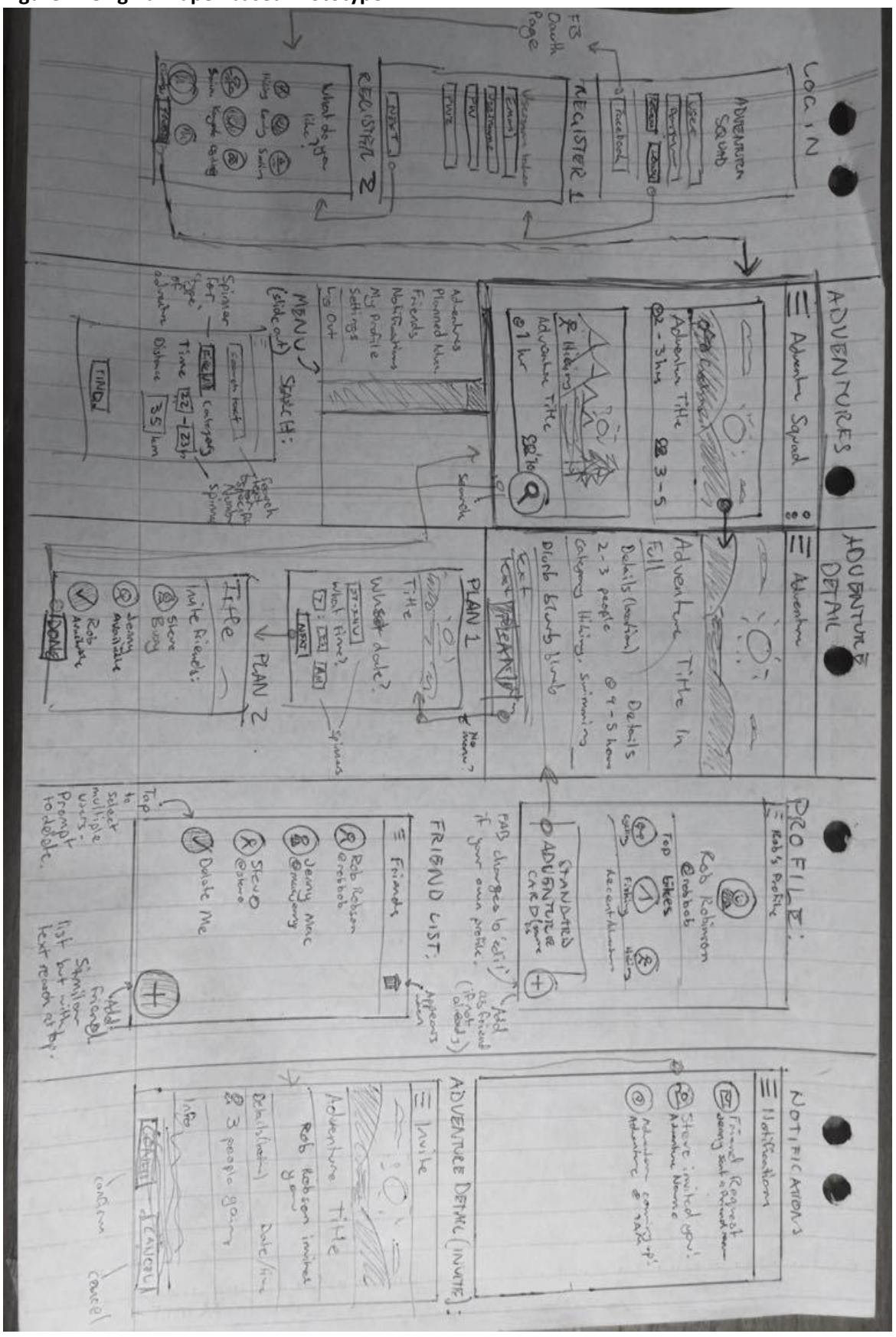
Once a user has found an adventure that they would like to go on, the user can view details about that adventure (fig 1. *'Adventure Detail'*). This includes info on how many people are recommended, the approximate amount of time to complete, and a short text description & extra info. The user can choose to plan this adventure on a certain date and time, and can then invite their friends to a planned adventure (fig 1. *'Planning'*).

Before the user starts using the app they have to create an account or log in using email & password (fig 1. *'Login'*). Users can create an account and must enter multiple details (fig 1. *'Register 1'*). Users are recommended to choose some interests (fig 1. *'Register 2'*). Users can also login using their Facebook account (fig 1. *FB Auth page*).

Users can navigate by using the slide-out menu from the taskbar (fig 1. *'Menu'*). This allows them to access other parts of the app e.g. their profile or their friends list. From the friends list (fig 1. *'Friends List'*), the user can add new friends or remove current ones. If their friend doesn't have an account, they can send them a message prompt to download the app. The user can also click a friend's name to view more (fig 1. *'Profile'*). The user can also view their own profile from the menu.

When any significant activities happen within the app (e.g. a friend invites you to go on an adventure), the app will display them in the notifications (fig 1. *'Notifications'*) section. From here the user can tap on any of the notifications and it will take them to the appropriate screen (e.g. tapping on an invite will take them to the '*Adventure Detail (Invite)'* screen).

**Figure 1: Original Paper-based Prototype**



### 3.3 Used Application Programming Interfaces

Currently in use

- Butterknife
    - Allows extremely simplified view binding & click handling
    - Reduces length of code and makes more readable
- Glide Api
    - Provides dynamic image loading for image views in lists
    - Also provides animations and caching
- hdodenhof's CircleImageView
    - Used to display circular images for Profile view
- Firebase
    - Email & password registration & authentication
    - Firebase real-time database
    - Firebase Storage for images related to adventures
- Standard Java utilities
    - Dictionary
    - ArrayList / List
- Android classes
    - Custom RecyclerView adapters for dynamic lists
    - Fragments and a custom Fragment Manager within 'Create a plan' flow

Planned APIs

- Google maps
    - Use Google Maps to lookup locations and determine distances
    - Create Google maps intents
- Location
    - Use current location to determine distances between you and an adventure
- Zebra Crossing < https://github.com/zxing/zxing >

## 4 Application Structure

Throughout the AdventureSquad application, I have endeavoured to build it in such a way that is scalable and loosely coupled. This was due to my use case – I have had to keep in mind that I will likely change the underlying data retrieval or the overlaying View technology at some point. As such, the application has a few notable features in its structure which are detailed below.

### *4.0.1 Object Oriented Design*

The AdventureSquad application aims to be as object-oriented as possible while still considering practicality. On the suggestion of some knowledgeable Android developers, I have endeavoured to implement the Model-View-Presenter (MVP) software architecture as much as possible. This has a few benefits, the chief benefit being that when implemented correctly, either a view class or an API class can be swapped out for another one without affecting the flow of the program. Within Android, to implement a successful MVP architecture I followed the following guidelines:

- A **View** is implemented as either an Activity class or a Fragment within an activity class.
    - The view should contain all Android-related code, including click listeners, view bindings, and so on
    - Also includes RecyclerView or other list Adapters that need to display data.
    - The view handles all input receiving & data entry, but passes these down to a presenter to handle the actual logic
    - Implements a standard interface so that the presenter can access it's callback methods
- A **Presenter** is implemented as a standalone Java class that implements the business logic.
    - It acts as an abstraction layer between the View and the underlying model
    - It does not have any knowledge of Android specific methods or libraries
    - It only has knowledge of generic view methods, for example 'displayMessage' or 'completeAction', that are provided to it through a java interface
        - These methods are then implemented by the view to provide view functions
    - Generally holds the Model once it has been retrieved from the data source (with the exception of Recycler views)

The model is implemented in two parts: The Model classes, and the APIs to store & retrieve the model from an external data source.

- **Model classes**:
    - These are 'plain old java objects' (POJOs) representing the basic data structure of the application
    - They are populated when downloading information from the database and passed throughout the application in order to display and work with the data
- **API classes**:
    - These are essentially 'helper' classes for interacting with a data source.
    - Currently all API classes are implemented to store and retrieve data from Google's Firebase system.
    - All API methods generally take a call-back class as an argument, which will be detailed further on in the document.

- 

I also ran into some problems and restrictions with using MVP structure.

- A problem with how the Adapters worked meant that I needed to add a call to the actual presenter from an Adapter.
    - While this worked, it broke the application in that the Adapter was communicating directly with the Presenter, instead of being handled by the View (Activity) class.
    - This also meant that every time a new list item was recycled, it would retrieve the URI and image again, leading to an amount of slow down
    - The solution to this would be to cache the image URI in the model object, so the RecyclerView Adapter class could juts use that.

#### *4.0.2 Call-back Methods*

To reduce coupling between the model/API classes and the presenters, AdventureSquad extends the concept of call-back methods that is common with Firebase programming and applies it to the MVP structure.

- The API usually takes an interface as a 'call-back'. When a task is complete it will then call the relevant call-back method,
    - This can allow simple callbacks with anonymous inner classes, or more complex ones where it requires a presenter class
    - e.g. when retrieving a list of data is complete, it will call 'onListRetrieve(List of objects)' on the original class that was calling it
- The callback system allows methods to be executed asynchronously within the Firebase classes, and to then notify upper classes when an action is completed.

Call-backs could be more abstracted. E.g. instead of taking a presenter interface for callbacks, it should take a more specific interface just for that specific task.

- I began to move towards this way of doing things later in development but did not get around to implementing it everywhere.

#### *4.0.3 Dependency injection*

Another software feature that I have worked to implement is dependency injection. This technique essentially refers to providing an object with its objects as necessary, instead of allowing the object to create its own instances of classes that it needs. This has a few benefits, the chief benefit being that it enables you to inject mock versions of the dependencies to an object for unit testing purposes.

I have implemented dependency injection within the Presenter and API classes. This allows another class (e.g. an Activity) to easily initialise a new Presenter with all of it's necessary API objects that it needs to retrieve data. It also means that

This also had some disadvantages, as in my implementation it increases the amount of downloading that the API has to do from the database. This could be improved by passing existing API objects

between activities instead of creating new ones for each activity & presenter, or by moving to a Singleton pattern for these APIs (similar to the recommended use of a Database helper class).

### 4.1 Packages

Java packages are split up by following the general purpose of the class (e.g. Model (for the data model), Activity, Adapter and so on) and then where applicable (e.g. in the Activity package), these are divided further into a subsection of each app. Interfaces Currently existing in the application:

- **Activity**
    - Holds all of the
- **Activity/interfaces**
    - Holds interfaces related to activities
    - Also holds Fragments that are in use in some activites (this is an incorrect placement, should ideally be in a fragments package)
- **Adapter**
    - Holds any RecyclerView (or other) adapters for use within list views
- **Api**
    - Holds API classes that communicate directly with the database / data source
- **Interfaces**
    - Holds interfaces that are responsible for enabling presenters to apply changes to the views (Activities)
        - E.g. PresentableLoginView
- **Model**
    - Holds all the data model classes
- **Presenter**
    - Holds all presenter classed
- **Presenter/Interfaces**
    - Holds all interfaces for the presenter to communicate with the API (callback interfaces)

### 4.2 Activities

Please refer to ‘view’ section above for more information.

### 4.3 Services

AdventureSquad uses no explicit android services. However, in using Firebase, it does perform a signifivant amount of data setting and retrieval in the background on a different thread from the app’s UI. The app generally makes use of Firebase’s built-in task management system by adding custom completion listeners on top of Firebase’s standard listeners. This system allows all explicit Firebase logic to remain in the API class, while still using the benefits of its listener system to provide asynchronous background data.

**5 Data Structure**

The data structure of the application is defined from a set of Java POJOs:

- Adventure
- AdventureType
- Plan
- Squad
- User

Due to the nature of Firebase being a 'NoSQL' database, the structure of the database is comparable to using a large, scaleable JSON or XML file. This also meant that traditional SQL relations and queries were not possible.

As a result, throughout the data structure I have stored multiple references to other objects when a relation is necessary, often in both directions. This was possible due to Firebase's 'push' method, which generates a new data point with a unique identifier (UID). Thus, most objects within the data structure will contain references to other objects. This introduces some complexity and overhead when an object is created or deleted, but significantly reduces the overhead when retrieving files.

For example, when a Plan is created, it holds a reference to its Squad as well as to the relevant Adventure. Then, after a plan is created, the program will follow the reference to the particular Squad and will add the newly created Plan to the Squad's list of plans. This allows a two-way lookup: it allows a list of plans for a particular squad to be retrieved quickly, as well as allowing a Plan's Squad to be retrieved without having to search through the squad list for the correct squad.

Below is a snapshot of the FireBase data structure, showing how the POJOs interact with each other.

**FireBase database hierarchical structure:**

## 6 Testing

Originally I would have preferred to get Android Studio's inbuilt testing facilities working with the application. The application was coded in such a way to better facilitate unit testing by incorporating software architecture concepts like dependency injection, which allows for mock objects to be more easily created and passed to a unit.

Unfortunately I did not find enough time to experiment and learn Android Studio's inbuilt testing features. This meant that I generally used less complex methods to test the application: debug logs, step-through debugging, crash logs and checking data output on each side.

## 7 Conclusion

- Had a few issues with getting Firebase working
    - Had to download multiple files
    - Still wouldn't work, had to use the Android Studio SDK manager to download the latest version of the Google Repository
    - This was the final step in getting Gradle to correctly sync the latest version
- Changing ID fields to long instead of int
    - Had to change these to 'longs' as an integer has a positive limit of around 2.5 million. This may be a problem if the app grows significantly.
- Added GitHub version control, started using version control inside the application itself
- Created a Trello board and began using that to track progress of individual items

**8 References**

Indicate where you got snippets of code from and other references you used to build your app

- Stack Overflow
    - Best way to structure date objects in Firebase
    - Best way to add to dictionary objects
- Android Documentation:
- Java Oracle Documentation
    - http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html
- Firebase Documentation:
- Github
    - Hdodendof's circle image view
    - Glide api docs