Launch Screen of the App

Final Project Report

STARTUP WEEKEND APP

CS3217: Software Engineering for Modern Application Platforms

Members:

- Dinh Viet Thang - A0126513N
- Jeremy Jee De Sheng - A0139963N
- Li Xiaowei - A0142325R
- Shi Xiyue - A0146123R

Overview

REQUIREMENTS

Startup Weekend Singapore (SWSG) is an annual entrepreneurship event where over 150 attendees would gather together and within 54 hours pitch a business proposal towards a panel of judges. The event is targetted at technologically-enabled startup enthusiasts and people who want to network with like-minded individuals.Throughout the 3 day event, participants would be involved in idea creation, peer voting, keynote speeches, mentor consultations and finally presenting their own idea. Our application, will be tailor made to be an ideal event application that works for both participants and organizers alike. On the participant's end, our application is targetted to make their experience more seamless and convenient by allowing them to easily form teams with the appropriate skillsets, receive updates and notifications, see the schedule of the event, book mentor consultation slots and communicate within the team. On the organizer's end, our application aims to automate the manual administrative processes of the event by simplifying processes like daily registration, announcements and keeping attendees up to date while no longer having to deal with things like mentor booking which have now been automated. In summary, we aim to create an application that would streamline the workflow and experience for both participants and organizers of Startup Weekend through our application.

FEATURES

In line with the requirements we set out to meet, our application provides a Schedule feature, allowing users to see the schedule of events during SWSG as well as add them to their Calendar, a Mentor Consultation Booking feature, allowing users to see available time slots and reserve them, a Team Formation feature, allowing users to look for team mates as well as indicate what sort of members a team is looking for, a Chat System, allowing users to direct message other users, create private group chats and allowing organizers to create public channels available to all users, an Idea Voting feature, allowing users to post ideas and people to up or down vote ideas, a People Search feature, allowing users to search for other users by name and message or favourite them, and a Registration feature for Organisers that allows them to use QR Codes to scan participants and register them automatically.

REVISED SPECIFICATIONS

In terms of specifications, the Idea Voting Platform was initially designed for Teams to post ideas but instead has been changed to allow individuals to post ideas.

For the Schedule, initially we were looking to integrate Google Calendar within the application, but we decided to use iOS Calendar instead since it suits and integrates better with the iOS Eco-System and all users would be able to access it on their device without any additional accounts.

A People Feature was added, that would allow you to search for other users, view their profiles and message them from their profile as well. This Features takes advantage of the existing Profile System that initially allowed people to view their own profile, and was then extended for use a View that could display anyone's profile to be reused within the People Feature.

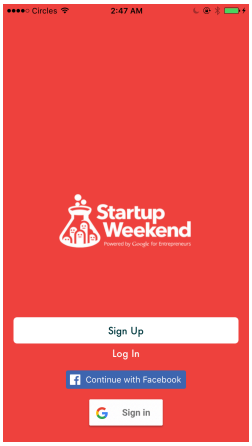
The App was initially conceptualised with a simple Registration Feature as well, however along the way the feature was forgotten and fell by the wayside. We have now added a Registration Feature into the application, that takes advantage of QR Codes for organizers to quickly register participants without the need to key in anything.

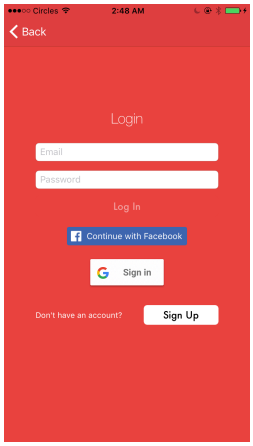
For the Home Screen, the initial concept was to have some form of dashboard that would allow users quick access to dynamic information and it was the last feature to be worked on. The Home Screen now displays all events for the day, automatically showing the next event as the initial view and allows the user to swipe left and right to see past or upcoming events for the day. The Home Screen also displays that latest chats that the user is involved in, automatically organising chats by their latest messages and showing the chat with the latest message by default.

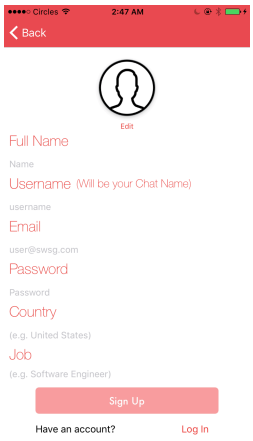
Integration of Components was also a feature that was not properly conceptualised in the earlier documents, the initial idea was simply to have different modules that were solely accessible from the menu making each feature largely standalone. However, to provide for a more intuitive user experience, we have made several improvements to the UI/UX by allowing for relevant features to be accessed from different parts of the application. For example, the Mentor, Chat, Team and Profile Systems were all conceptualised separately and to work separately.

USER MANUAL

REGISTRATION & SIGN UP

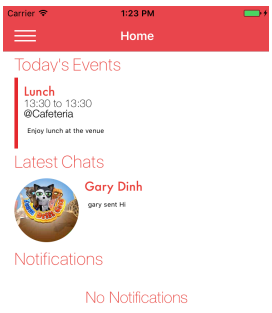


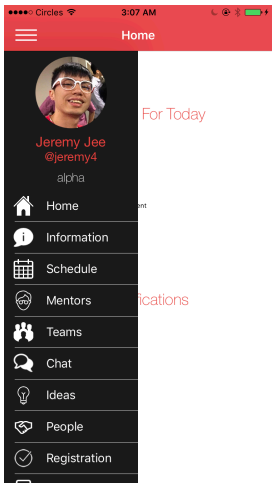




The first screen that the user sees when he starts up the application is the Start-Up Screen. From here, the user can select to Sign Up, Log In or proceed using Social Media Methods. Sign Up bring the user to the Registration Screen where the user would fill up his personal details and has the option of adding a Profile Image as an avatar. Proceeding using Social Media, the options available are Facebook and Google Account, for a user who has not registered before would then bring the user to the Registration Screen with his Name, Email and Profile Image automatically retrieved from the Social Media Account. If the user registers using a Social Media account, he can log in automatically using the Social Media Login in future. Log In would bring the user to the Log In page providing both the traditional email and password authentication and social media as well. If the user initially creates his account using an email and password account, then attempts to subsequently log in with his social media account, he would be prompted to log in with his email and password to authenticate, following which that social media account would be automatically linked to his existing account, providing multiple authentication methods.

HOME SCREEN



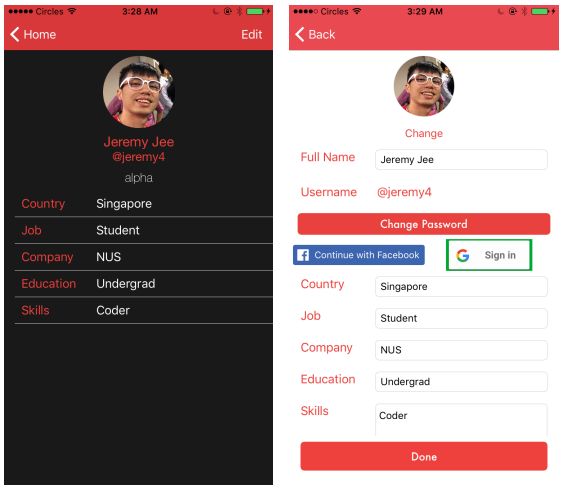


After Registration and Log In, users will be brought to the Home Screen that provides a quick dynamic look at information. Users will be able to see the next upcoming event for the day, and be able to swipe back and forth to see previous and upcoming events for the day. Tapping on an event, would bring the user into the Events System. Users can also see the latest message that they have received and swipe back and forth through their chats, that are organised by the timestamp of the

latest message. Tapping on a chat, would bring the user into the Chat System. Most importantly, by tapping on the Menu Icon at the Left side of the Navigation Bar would reveal the Hamburger Menu for the App. From here, users can access the different features of the app, users can also see their profile image, name and username while tapping on them would bring the user to their Profile page.

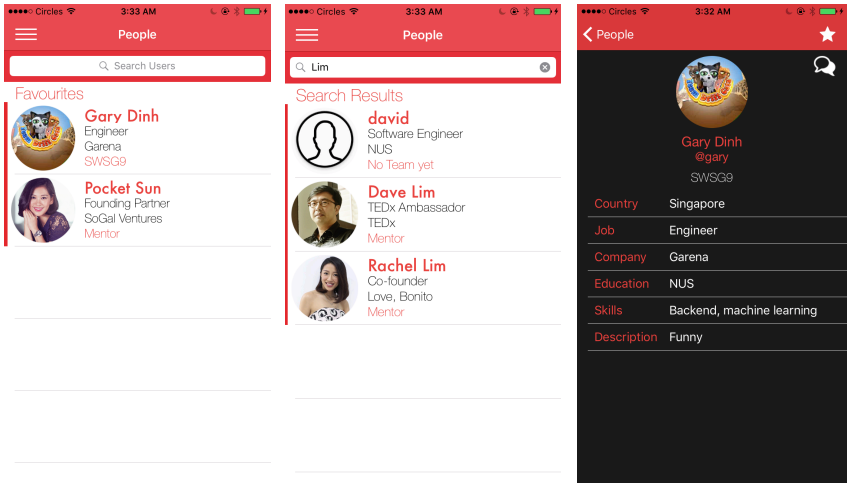
PROFILE SYSTEM

The Profile System shows the user's public profile page that other users can view as well through the People System. Users are able to edit their own profile, all details except username and email are editable including profile picture and authentication methods. Users are able to add a password if they registered through Social Media and link or unlink Social Media accounts to their existing accounts.



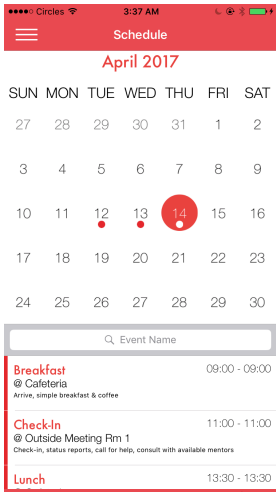
PEOPLE SYSTEM

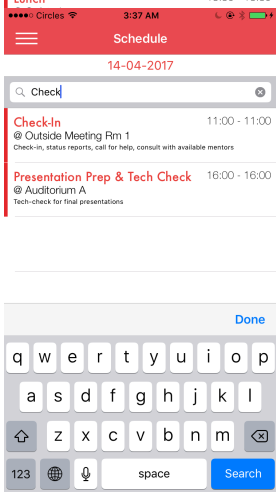
The People System starts by showing your anyone that you have favourited previously and allows you to search for other users of the application using either their name or their username. Tapping on a user would bring you to his public profile. From the Public Profile, you would be able to Favourite him or start a direct message with him via the Chat System.

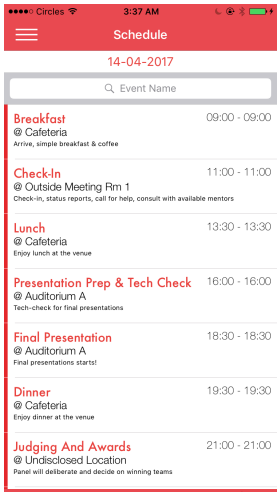


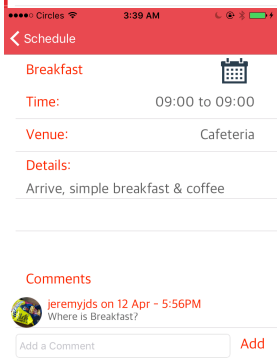
SCHEDULE SYSTEM

The Schedule System provides an overview of events for SWSG. At a glance in the calendar view, users can see which days have events occurring on them. Tapping on the day would review a scrollable list of events below. Double tapping would maximise the list to full-screen. Users can also use the search bar to filter the events for the day, to find the event that they are looking for on that day.



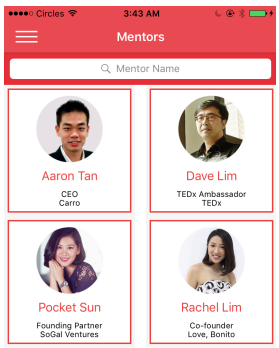


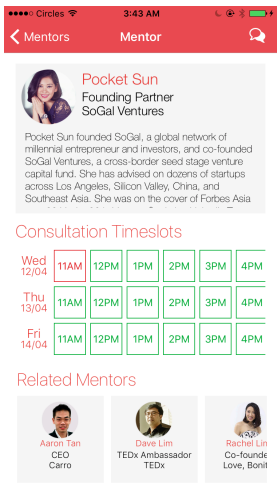




MENTOR SYSTEM

The Mentor System showcases all the mentors participating in SWSG, users can tap on a mentor to see more details or use the search bar to search for a mentor.

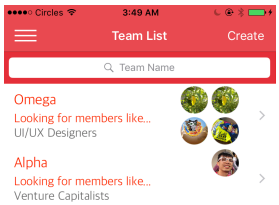


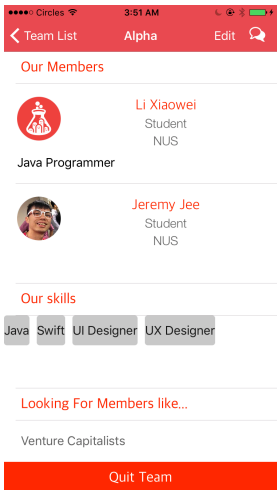


In the Mentor Details Page, tapping on the mentor description on top brings the user to the mentor's Profile page. Tapping the Chat Icon on the right side of the navigation bar starts a direct message with the mentor in the Chat system. Users can scroll horizontally and vertically on the Consultation Time slots to see all available times for all days, the day column is stickied so that users can scroll the slots while still seeing which day it is on. Tapping on a day would

bring up a prompt to book that slot, bookings can only be performed as part of a time on a vacant slot. Green represents vacant, Red represents booked and Grey represents unavailable. Mentors shown at the bottom are involved in the same field, and tapping on them would bring the user to that mentor's detail page. As a mentor, viewing your own mentor page would provide more details, for example tapping on a booked slot would reveal which team booked the slot and gives the option to view the team's profile in the Team System. Tapping on a vacant or unavailable slot would provide the option of changing the status of the slot to unavailable or vacant respectively.

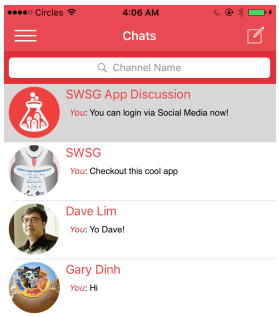
TEAMS SYSTEM

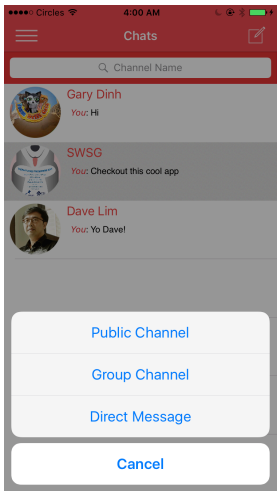




The Teams System allows you to create a team, listing down the skills your team currently possesses and the type of people who you are interested to have on your team. Participants can then look through the list of teams that are open and freely join or quit teams. Each team also has its own Team Chat, as part of the Chat System, that can be accessed by tapping the Chat Icon in the Team Details Page. The Team Chat dynamically updates its members based on the members of the team. Participants can also edit the team details if they choose to do so, and can also search for teams based on team names.

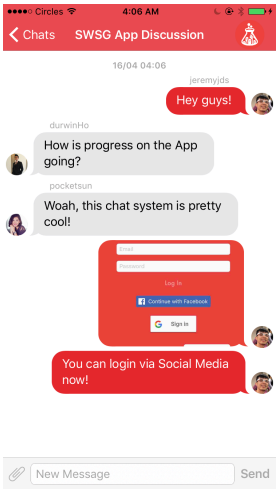
CHAT SYSTEM

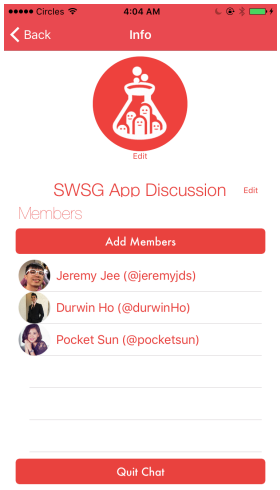




The Chat System is a messaging system within the app that allows you to message any other user within the app. In addition to direct messaging, you can also create Group Channels that are like private chats allowing you to add multiple members into a single chat, while organizers are allowed to create Public Channels that all users can see. Users can also search for chats by chat name, or if it is a direct message, by the other user's name. Users can also swipe across direct message chats to delete them, while private chats can be quit from

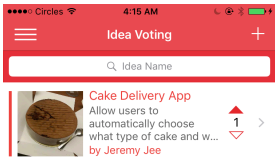
the Channel Info page. Tapping on a chat brings the user into the Channel page, where the user can send text messages and photo message as well as see typing indicators if other users are typing as well. Tapping on the top right icon on a group would bring users into the Channel Info page allowing users to change the Channel Icon, name, add members to the chat or quit the chat. Swiping across members on the list would provide the option to delete them from the chat. For team chats, none of the Channel Info can be changed as it is controlled by the system. For direct chats, tapping on the top right icon in the Channel would bring you to the user's profile page.

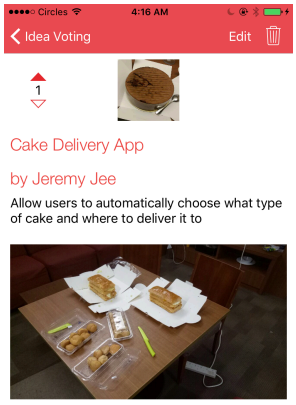




IDEAS SYSTEM

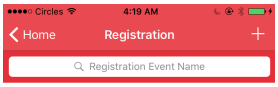
Users can create and post ideas, which other users can up or down vote. Each user gets a single vote, and users can change their vote at any time. Users can also search for ideas by name. Tapping on an idea brings the user to the Ideas Detail page that allows the user to see more details including additional pictures and embedded videos. Users can also tap on the Idea Poster's name to see his profile and chat with him from the Profile.

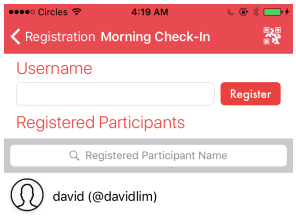




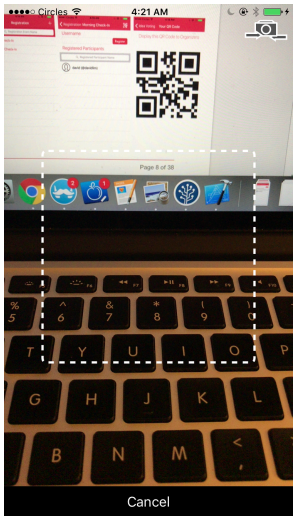
REGISTRATION SYSTEM

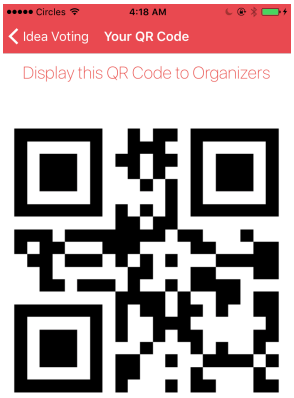
The registration system allows organizers to create registration events, in which they can use to track registered participants. Organisers can simply tap the QR Code Button at the top right of the Registration Event Details to bring up the QR Code scanner. Scanning the participant's QR Code will automatically register him into the event. For participants, simply accessing the





Registration System would automatically generate and display his QR Code that he then needs to let the organiser scan.





PERFORMANCE

The main resource that the App requires for operation is an Internet connection, given that many of the functions are closely integrated with Firebase, a lack of network connection would greatly cripple the application. The application primarily uses algorithms that run in Linear Time at O(n), especially in terms of processing time. In terms of space, the application tends to call data and query only data that is required from the Firebase Server, allowing the Firebase server to perform the appropriate filtering before sending the data to the application. Hence the application typically handles O(log(n)) in terms of space for the different components since it does not store the entire database locally.

Testing

STRATEGY

Our Testing Strategy focuses primarily on Bottom Up Black Box Testing, given that such is most commonly what a user would himself experience. We would perform Black Box Testing on our individual components, ensuring that the individual components are working as according to the specifications first in their individual branches, before combining them into the master branch. We then perform Integration Testing, ensuring that our different components are able to work well together and would not produce any errors in normal operation. We created a list of tests to perform for each component based on the specification, and we also utilise this list for regression testing. Each time we iterate and add an additional feature, we ensure that our existing features are still working properly while also checking that our new feature works correctly in conjunction with the old features.

We also use defensive programming techniques, making sure to use guard statements and checks to ensure that errors from unwrapping optionals or array out of bounds do not happen. We primarily use these Compile-Time Checks to ensure that our App works properly rather than Run-Time Assertions that may cause an App to suddenly crash if something unexpected happens. By using our Compile Time Checks, we ensure that there is not unexpected situation that is unaccounted for.

We opted not to use Glass Box Testing since the majority of our code is intricately linked with Firebase, it is not practical to write Unit Tests that test how our App works with Firebase. Instead we rely on Compile Time Checks to ensure that data that is serialised and deserialised from Firebase are properly done and handled. While using Black Box Testing to test scenarios and then compare if the data on the Firebase Console is indeed correct.

Due to not performing Glass Box Testing, we did not adopt any form of Test Metrics other than that of our Black Box Testing and ensuring that all the tests we listed pass. Primarily our list of tests come from a user's common experience with the app, starting with a list of normal operations that an "informed" user would do and then expanding to a list of "extraordinary" operations that a user may use to test or crash the App. This ensured that we covered many of the edge cases and that there were no situations whereby we were caught unawares.

We expected to find errors that a user would encounter in his normal operation of our application, as well as any more extreme errors or violations that the user may take if he was trying to test the limits of our applications. Yet it may be hard to fathom all possible methods by which the user would try to test the system, as well as the fact that due to the lack of glass box testing, it may be hard to pinpoint logic errors in the App. Errors that may be caused by the connection to Firebase, or the lack of a network connection or a slow network connection may also be hard to detect due to the asynchronous nature of Firebase connections.

Particularly due to the networked nature of our App and the reliance on libraries, it makes it very hard for glass box testing to be done and only facilitating manual black box testing. Since manual black box testing is also done by people, and the list of tests is created by people there would tend to be areas that are overlooked. In order to mitigate this, we got friends who were not involved in the App nor IT savvy to try and use the application blindly to see if they encountered any unexpected behaviour or cause any errors. This has helped us track down and root out numerous bugs as well.

TEST RESULTS

Black Box Testing of data serialisation and deserialisation has been properly tested and checked to work, any method that involved sending data, editing data or receiving data from Firebase has been tested and checked to work. Since these cases were not coverable by Glass Box, we made sure that during the implementation phase we tested all the cases through some sort of “manual unit tests” by making sure that for every component we tested each CRUD (Create Read Update Delete) Operation in each module.

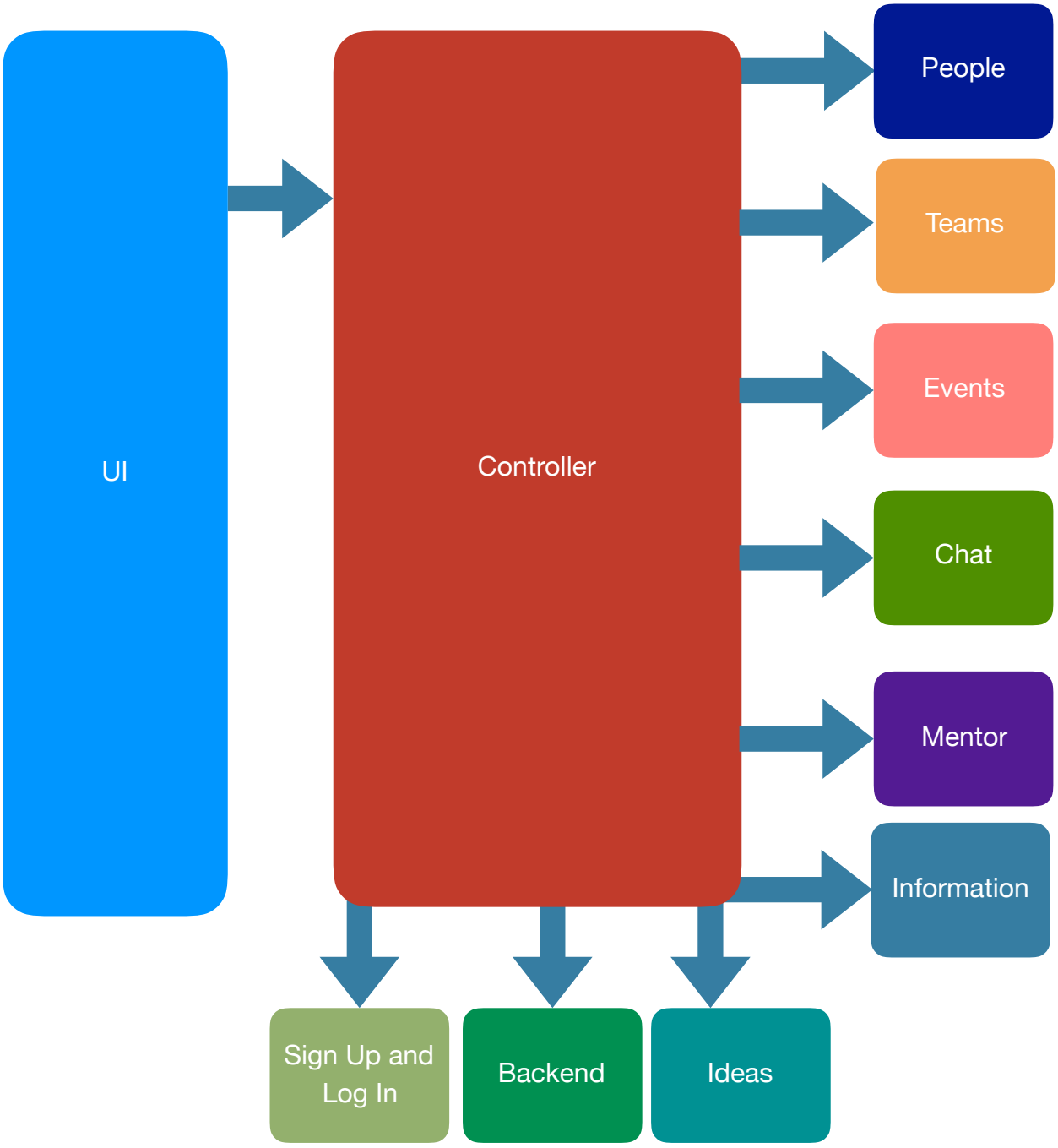
All modules have been tested, though the thorough cover of testing may differ from module to module. The Home, Schedule, Mentors, Chat, People and Registration have been thoroughly tested with Black Box Testing to ensure that the code works as expected. Faults that have been eliminated include any form of array out of bounds error, a slow asynchronous callback from the Firebase server, any form of improper user inputs e.g. empty strings, any combination of actions that may result in a user getting “stuck” and being either unable to go back or unable to close the keyboard, ensuring that the correct data is sent while jumping from module to module. Errors that may be left behind would most probably be those that the user would not encounter in a normal operation of the App, or layout or formatting errors that sometimes occur due to a slow network connection. This slow connection could lead to data not being loaded, and subsequently not being passed

along during a segue. Certain errors such as the asynchronous callbacks loading wrong images on table cells, due to the content having changed since the asynchronous calls do not seem to have any easy solutions.

Design

OVERVIEW

TOP-LEVEL DESIGN



Our application is organised into modules or packages, as we wanted to make our application extensible. Particularly due to its nature as being an event management app, we believe that there is great potential to add on additional features to our application and we wanted to make it easy to further extend our application. The two largest packages would be the UI, which contains all classes related to the Views, and the Controller, which would be the classes that control the Views like the ViewControllers. Then we would have multiple smaller modular packages, one package for each module that corresponds with a feature, such as Users that would handle log in, profiles etc. or Messaging that would handle all aspects of the group chat.

INTERESTING DESIGN ISSUES

The use of Firebase as a backend gave particularly interesting design issues especially with respect to cohesion, initially we had planned to have all the backend methods in a single Client that would manage all connections to the backend. However, to take advantage of Firebase as a Real-Time Database to update our App's Info in real-time, we had to do most of the deserializing within our Controllers themselves instead of the Firebase Client. Hence this has led to a higher than expected coupling.

The design of the User class has particularly gone through many iterations, given that we have multiple user types we wanted a model that could easily support that. Hence we had the UserTypes being a struct of Bools, that would allow you to indicate more than one type of user role. However, we had overlooked this for the actual implementation of the class. Our initial concept was to use a form of Protocol Implementation, with User being Protocol and the different roles being implementations, for example Participant and Mentor would be classes that implemented the User Protocol with additional type specific values such as Profile or Consultation Slots. This ended up leading to issues in that, when a user had multiple types, he would have more than one implementation of the User Protocol which would end up to be extremely messy since we would then have to keep track of multiple protocols. We then transitioned to a simpler model whereby we would have a User class that would contain solely the authentication details, email and password, with a Profile class that would contain all the description of Users and any optional classes. We then discovered that this had low cohesion since a Profile class should not have irrelevant details inside like Consultation Slots etc. Hence we arrived at our current solution, a User class with the appropriate User level variables such as email, team, profile and optionals for type specific

values such as consultation slots, while the Profile became solely a struct to store more specific information.

One of the biggest design issue we faced was with regard to storage, our initial idea was to have a Storage class that would handle all the data locally while the backend was being worked on. We would then swap out all the local methods within Storage with Firebase ones, and the app would still work fine as before. However, as it turns out that the methods and principles we used to design the local storage ended up to be too different from how Firebase worked, and we ended up having to redesign the storage from the ground up.

USE OF LIBRARIES

We used several libraries, particularly a number of them for UI purposes and most importantly backend as well. Our most important library used for Firebase, as our Authentication and Database backend, complemented with the Facebook and Google SDKs for the Social Media Logins. We used One Signal as our Library for Notifications, and we used two libraries for Registration, one to scan QR Codes (QRCodeReader) and one to generate QR Codes from text (QR Code). The remaining libraries were for UI Purposes such as SwiftSpinner, JSQMessagesViewController, JTAppleCalendar, RSKImageCropper and TextFieldEffects

PROBLEMS WITH DESIGN

Much of the additional hassles that we faced with design had to do with how our App would work with Firebase, particularly we had to take into account the lag of the connection as well as how to download as little data as possible. In terms of performance, we created a local cache for the different images such as profile images, icons etc. that would be stored on system and checked each time whether the image URL had changed or not and then still needed to be re-downloaded and updated, else the App would simply take it from the cache. This has led to additional space requirements, since we now need to store the images, while reducing the possible lag time that users experience in images being loaded. There is also still the check being run to check for changes that would also take up so bandwidth, but we deem that to not be very significant.

The decision to have a single file to act as the main interface for the backend has also led to the issues of a massive file, currently the FirebaseClient class is extremely long since it contains the functions for all the different features. Refactoring it would take an extremely long time given the coupling of many features to the FirebaseClient file, hence it was a

design mistake to have all the backend implementations within a single file. We could have alternatively looked at ways for the single file to just be the interface with the implementations being done elsewhere in the Backend package.

RUNTIME STRUCTURES

USERS

User

- `User` is a class that represents a user of the app. It contains information about the user: profile: email: String, Profile, type: UserTypes, team: String, mentor: Mentor?, uid: String?, favourites: [String]?. Its representation invariant are:
  - email has to be valid
  - profile and type have to meet their representation invariant
  - If user is not a participant, the user cannot have a team
  - If user is not a mentor, the mentor attribute has to be nil

Profile

- `Profile` is a class that represents the profile of a user. It contains information about the profile: type: name: String, username: String, image: UIImage?, job: String, company: String, country: String, education: String, skills: String, desc: String. Its representation invariant is that
  - except for image, none of its fields can be empty

UserTypes

- `UserTypes` is a struct that represents the type of a user. A user can be multiple types (any of isParticipant, isSpeaker, isMentor, isOrganizer, isAdmin), but must have at least one type. Its representation invariant is that it cannot be neither isParticipant, isSpeaker, isMentor, isOrganizer, nor isAdmin. `UserTypes` is immutable.

EVENTS

Event

- `Event` is a class that represents a particular event. Event attributes include event image, event name, event date and time, description, details and venue.

Comment

- `Comment` is a class that represents a particular comment posted by an active user. Comment attributes include words, and the user's ID.

TEAMS

Team

- `Team` is a class that represents a particular team formed. Advised team size is 4 members. Team attributes include team id, members, team name, team info, type of skills that the team is looking for, and the visibility of the team, mainly whether the team is a private or public team.
  - rep invariant: members.count != 0 AND team name != empty AND team skill tags != empty AND skills the team is looking for != empty

IDEAS

Idea

- `Idea` is a class that represents an idea of a user. Idea implements ImagesContent and TemplateContent protocols. Idea attributes include id, name, user, description, mainImage, images, videoLink, votes, imagesState. Its representation invariant is that a user can only get a single vote.

MENTORS

Mentor

- 'Mentor' is a class that represents data for a mentor. A Mentor contains a Profile class and a Field that describes which field of work he is in, and an array of ConsultationDate.

ConsultationDate

- 'ConsultationDate' is a struct that represents data for the Consultation Slots for the day. It contains the Date for which it contains slots for and an array of ConsultationSlot.

ConsultationSlot

- 'ConsultationSlot' is a struct that contains the start time, duration, team and status of a Consultation Slot.

ConsultationSlotStatus

- ‘ConsultationSlotStatus’ is an enum that contains the three states, vacant, occupied and unavailable.

Field

- ‘Field’ is an enum that contains the types of working fields the mentor is in. It is user to group mentors in related fields together for display.

CHAT

Channel

- ‘Channel’ is a class that contains the id, ChannelType, icon, name, latestMessage and an array of Strings for members. For members, the array of Strings contain their User IDs that are unique. The extension of a ChannelType allows multiple different types of channels to be defined as well.

ChannelType

- ‘ChannelType’ is an enum that contains the 4 possible types of channels: Public Channel, Private Channel, Direct Message and Team.

Message

- ‘Message’ is a class that contains the attributes of the message that was sent such as the sender’s ID, the sender’s username, the timestamp, the message and a image if included.

REGISTRATION

RegistrationEvent

- ‘RegistrationEvent’ is a class that contains an id, name of the registration event and an array of Strings for registered Users. The array of Strings contains the UIDs for the users.

UI SUPPORTING

OverviewContent

- `OverviewContent` is a class that represents the overview information of SWSG. OverviewContent implements ImagesContent and TemplateContent protocols. OverviewContent attributes include id, description, videoLink, images and imagesState

ImagesState

- ‘ImagesState` is class to indicate whether images have been fetched or changed. We have such a data structure because images fetching and uploading are very time consuming. To achieve a better performance, when user edit an idea / overview content, if images are not changed, we won’t update images in the database. When user view an idea / overview content, if images have been fetched, we won’t fetch again.

ImagesContent and TemplateContent are helper protocols. By using them, we can improve the code reusability when displaying/editing ideas and overview.

ImagesContent

- `ImagesContent` is a protocol that represents a content which has multiple images. ImagesContent attributes include id, images and imagesState. It has some default implementation to fetch images.

ImagesContent

- `TemplateContent` is a protocol that represents a template content, which will be shown in TemplateViewController. TemplateContent attributes include description, images and videoLink.

BACKEND

PushNotification

- `PushNotification` is a class that represent a push notification. It contains information about the notification: type, message and additionalData. The type and additionalData store information that helps handling the notifications when we receive it, additionalData is a [String: Any] which is very flexible and we can store any kind of information we need in order to process the notification correctly.

FBRequest

- ‘FBRequest’ is a class that is used to deserialise data from Facebook’s Graph Response Protocol. It deserialises that response into a SocialUser class

SocialUser

- ‘SocialUser’ is a class that is used to store data from responses from Social Media Authentications, it has the user’s id, name, email, AuthType and url from his Social Media profile.

AuthType

- ‘AuthType’ is an enum that identifies all the available methods of Authentication such as Facebook, Google and Email

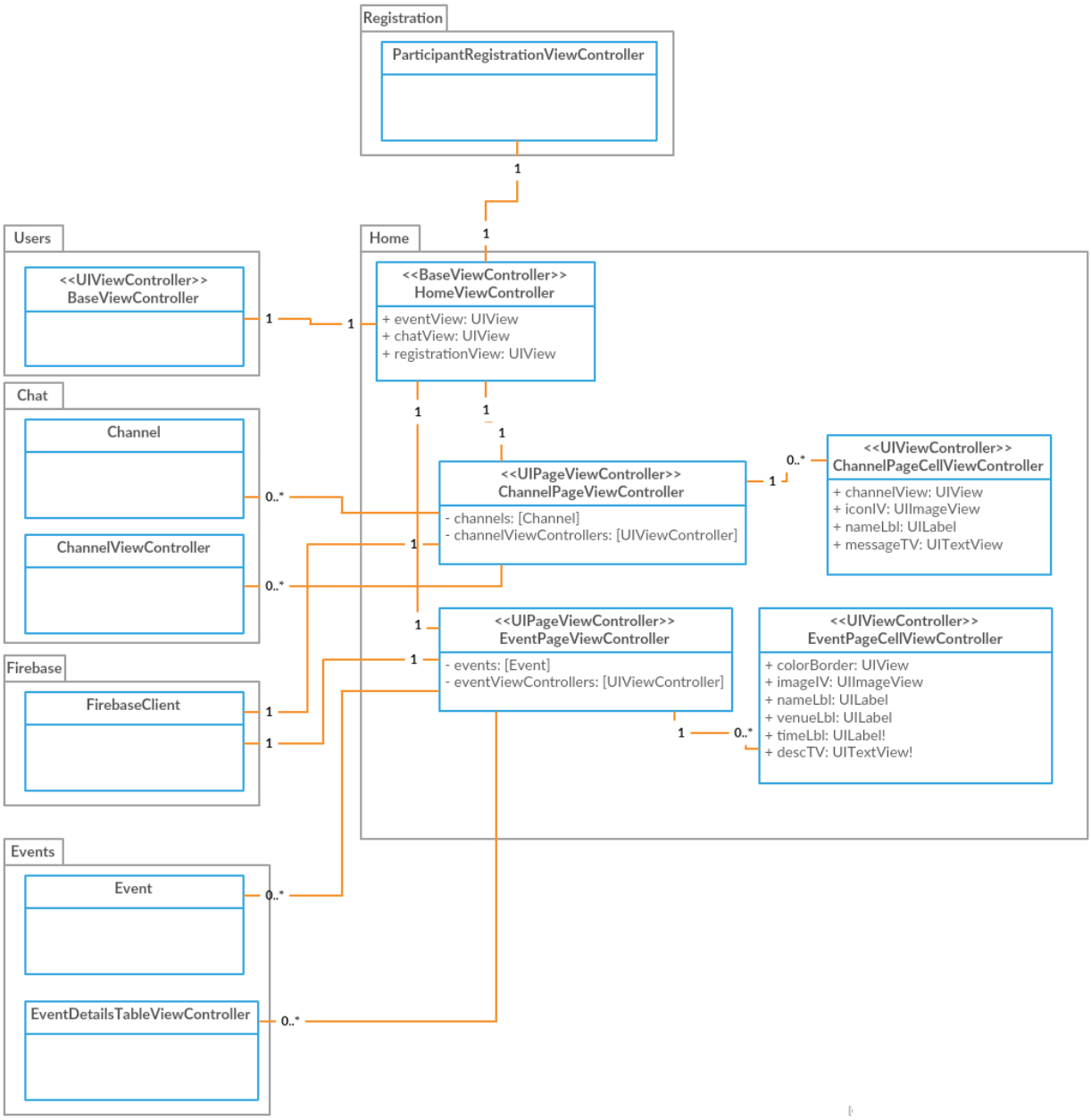
MODULE STRUCTURE

Our App utilises the Model-View-Controller (MVC) design principle, in order to decouple the Data, Presentation and Logic of the Application.

Most data types are represented as classes, as we want to leverage on the use of implicit sharing. Such as in the case of updating the profile of a participant, we want this update to reflect across all references to it such as in his team page as well. For Constants and Utility Methods or those that do not require sharing, we typically use either Structs or Enums for simplicity. Such as representing Config as a struct.

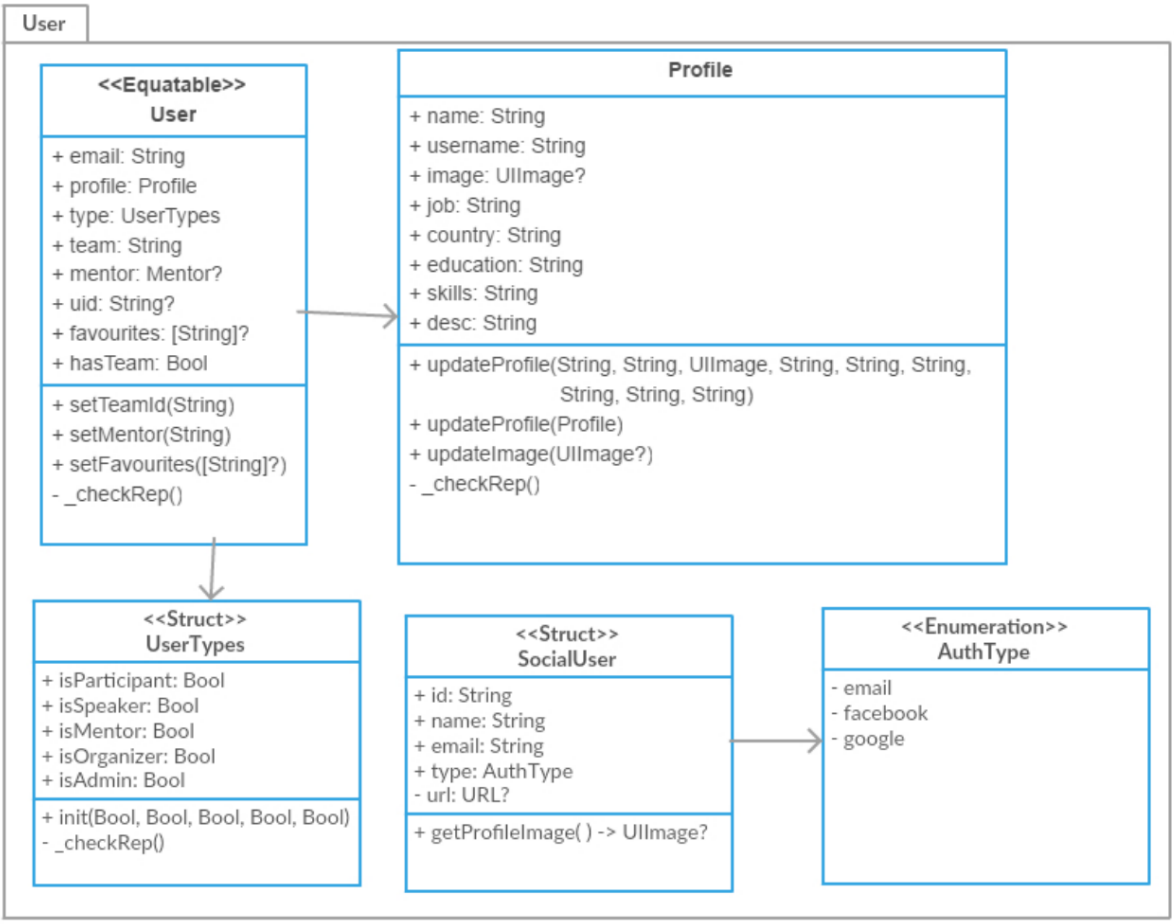
Each feature also has its own storyboard that is separated, rather than having a huge storyboard lumped together. We feel that this facilitates low coupling and high cohesion and also prevents a “Big Bang” when it comes to merging storyboards together. In terms of developing separate modules, we separate each feature into their own package thereby following the Separation of Concerns principle, reducing functional overlaps and reduces any ripple effect when changing one of the modules.

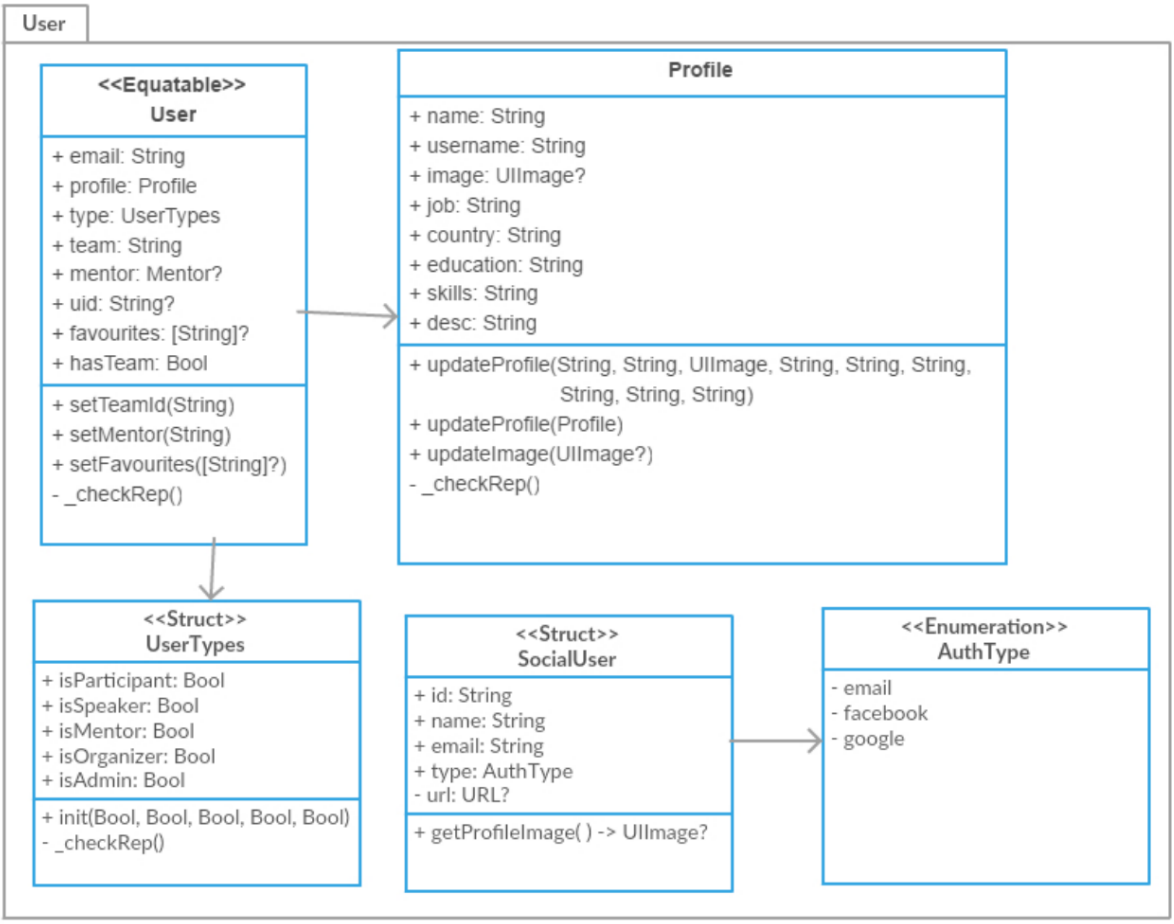
HOME MODULE



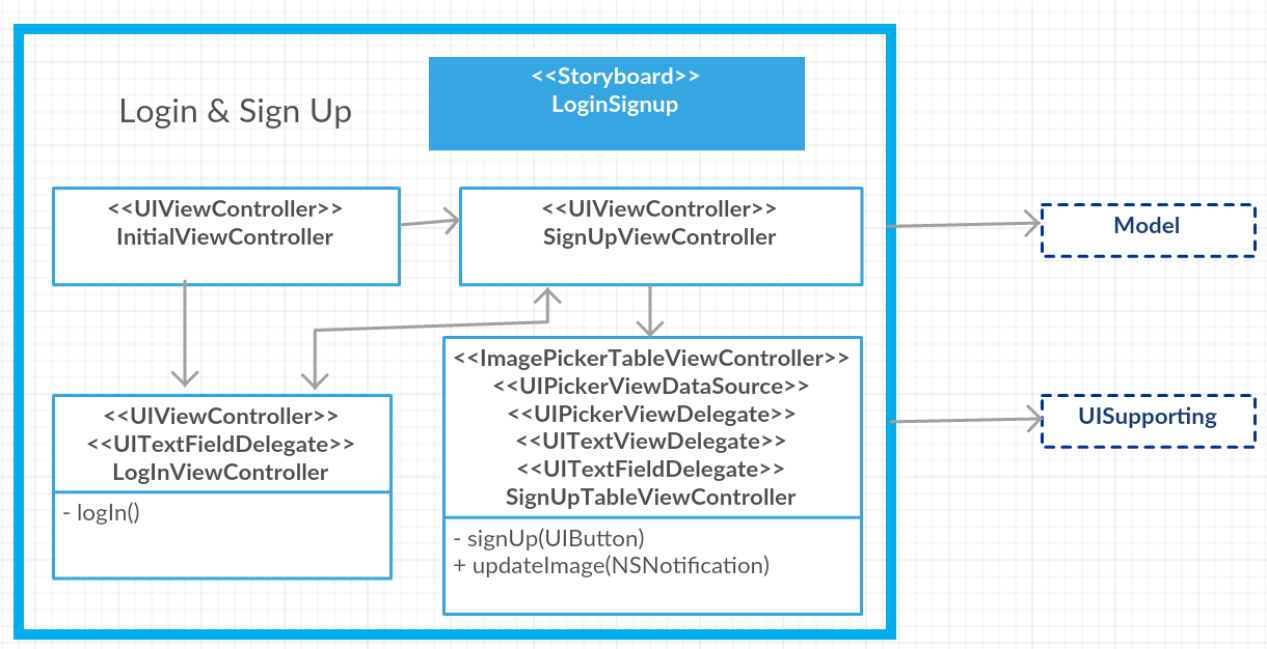
Home mainly relies on other components and amalgamates the data from the components to display in widget like container views on the HomeViewController, it mainly pulls data from Chat and Events while re-using the entire ParticipantRegistrationViewController as well.

PEOPLE MODULE



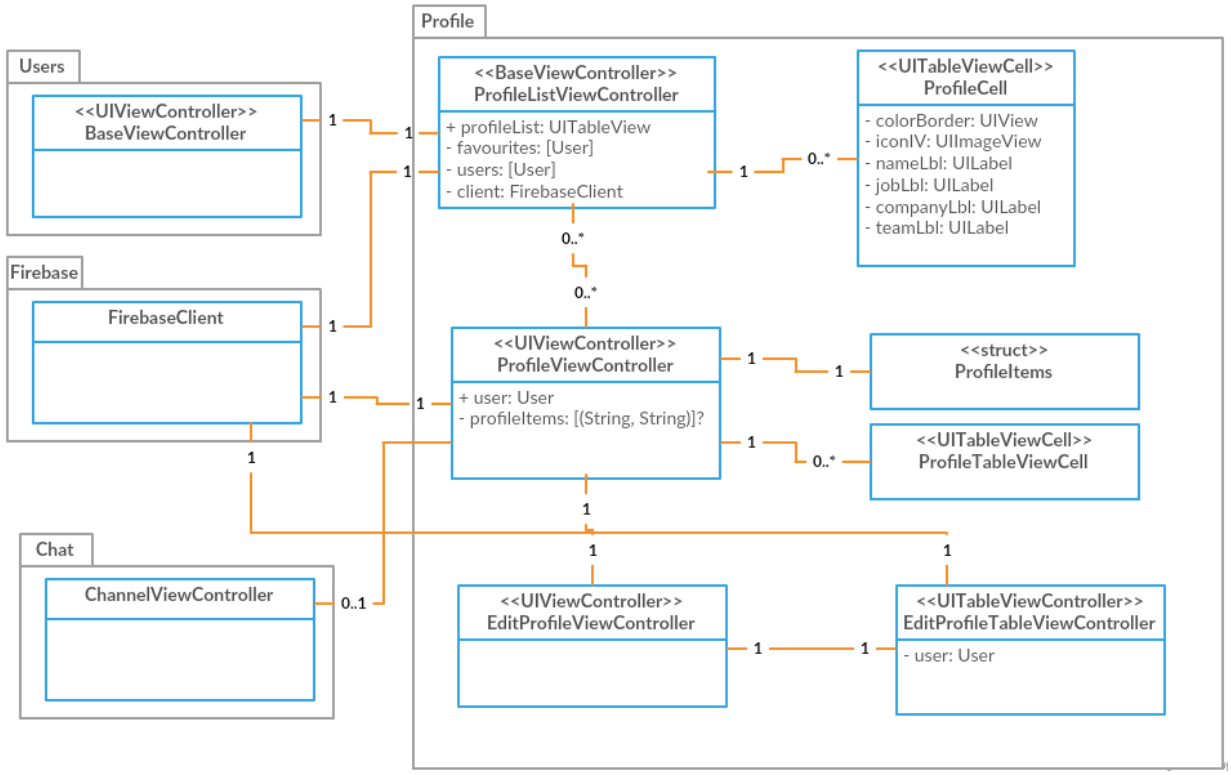

User
<<Equatable>>
User
+ email: String
+ profile: Profile
+ type: UserTypes
+ team: String
+ mentor: Mentor?
+ uid: String?
+ favourites: [String]?
+ hasTeam: Bool
+ setTeamId(String)
+ setMentor(String)
+ setFavourites([String]?)
- _checkRep()
Profile
+ name: String
+ username: String
+ image: UIImage?
+ job: String
+ country: String
+ education: String
+ skills: String
+ desc: String
+ updateProfile(String, String, UIImage, String, String, String, String, String, String)
+ updateProfile(Profile)
+ updateImage(UIImage?)
- _checkRep()
<<Struct>>
UserTypes
+ isParticipant: Bool
+ isSpeaker: Bool
+ isMentor: Bool
+ isOrganizer: Bool
+ isAdmin: Bool
+ init(Bool, Bool, Bool, Bool, Bool)
- _checkRep()
<<Struct>>
SocialUser
+ id: String
+ name: String
+ email: String
+ type: AuthType
- url: URL?
+ getProfileImage( ) -> UIImage?
<<Enumeration>>
AuthType
- email
- facebook
- google

LOGIN & SIGN UP



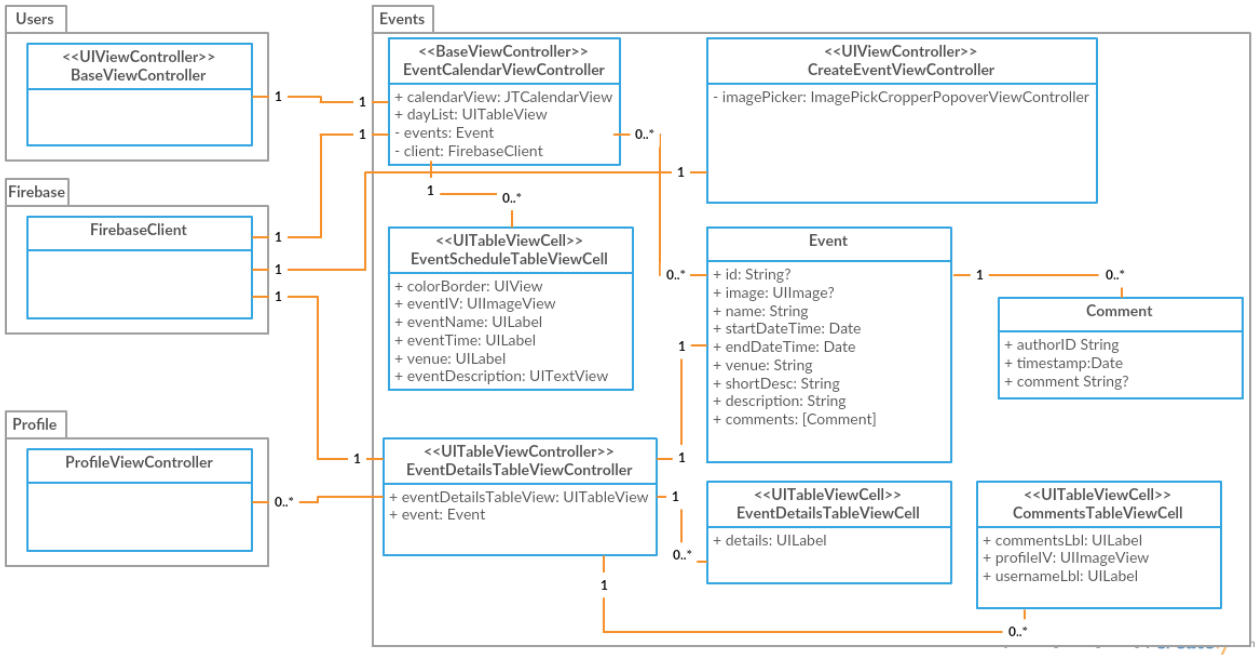
Login and Sign Up are also able to easily extend to support additional Social Media Authentications. The original system with only email and password was very easily extensible to work together with the Social Media Model classes and use them for Log In and Sign Up. For example, to integrate Social Media data into Sign Up, a SocialUser class is passed in and Sign Up would automatically read and include those variables. Log In has also been extended to facilitate the addition of additional authentication methods, if the user already has an account and is attempting to authenticate with a new account, Log In would automatically display only the existing log in methods and after logging in would automatically add the additional authentication method to the account.

PROFILE (UNDER PEOPLE MODULE)



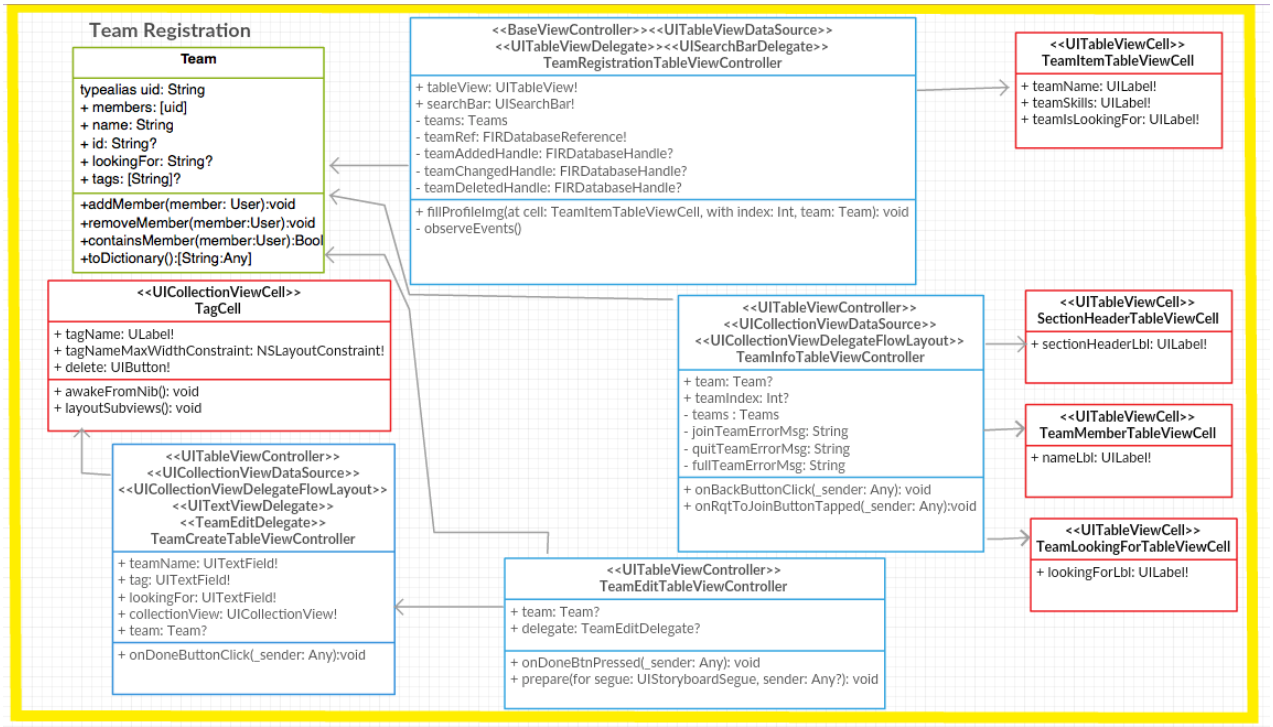
The Profile Module was initially conceived as just viewing your own profile and then being able to edit it, consisting only of the ProfileViewController and EditProfileViewController. When we decided to add the feature to allow users to search for other users and favourite them, as well as a public profile system, we found it very easy to extend from our original concept. The design of Profile View Controller was simply dependent on the user Model being given, and thus we could easily extend it to be reusable for any users. Just slightly altering certain settings depending on whether you are viewing your own profile or other people's profiles, for example viewing your own profile gives you the option to edit while viewing other people's profile gives the option to favourite or chat with the person.

EVENTS MODULE



Initially there were two ViewControllers for displaying events, one in Calendar View and one in List View, tapping on the Calendar View would bring you to the list view where you could tap on an event to see the details. We felt that this was not a very intuitive experience, and thus combined the two into a single View Controller with the calendar displayed on top, and a list of events for the selected day in the calendar listed below.

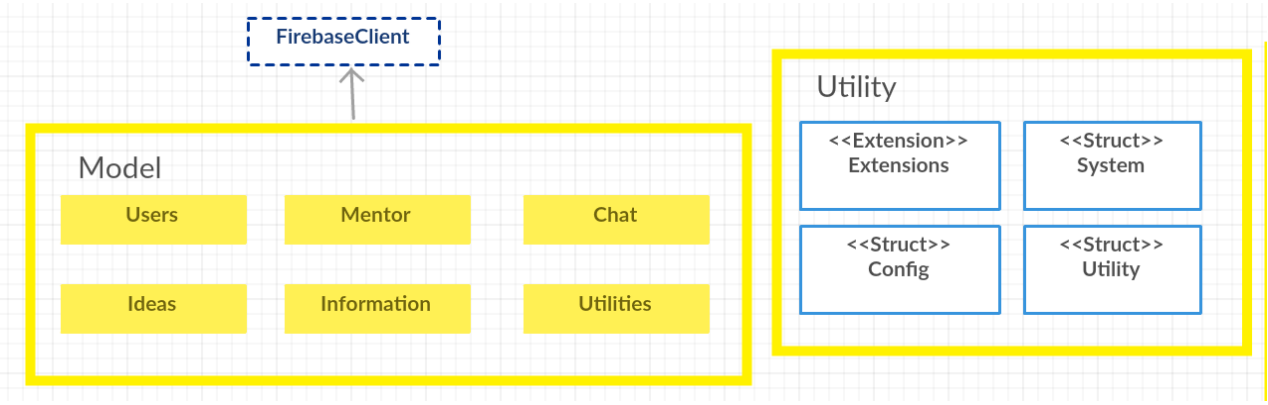
TEAMS MODULE



The main page for team registration is a list of teams handled by TeamRegistrationTableViewController, which inherits from BaseViewController to ensure consistent styling and implement UITableView datasource and delegate methods. Upon clicking the individual table cell, user will be segued to the team information page, which is handled by TeamInfoTableViewController, and implements UICollectionViewDataSource and UICollectionViewDelegateFlowLayout methods, this is because we embedded a collectionView in one of the static table cell to display skill tags. The skill tags width are adjusted based on tag length. User can click on member icon to view member profile. If user belong to this team, he will be able to tap on the `Edit` button on the top right corner to edit the team information, and chat icon to chat with team members. Once user joined a team, he will automatically join the team chat group as well. The editing of current team is handled by TeamEditViewController, and reuse TeamCreateTableViewController by embedding it inside a container view. User can create a team by clicking on the `Create` button on the upper right corner of team registration page. Note that user can only join 1 team, and maximum team size is 4.

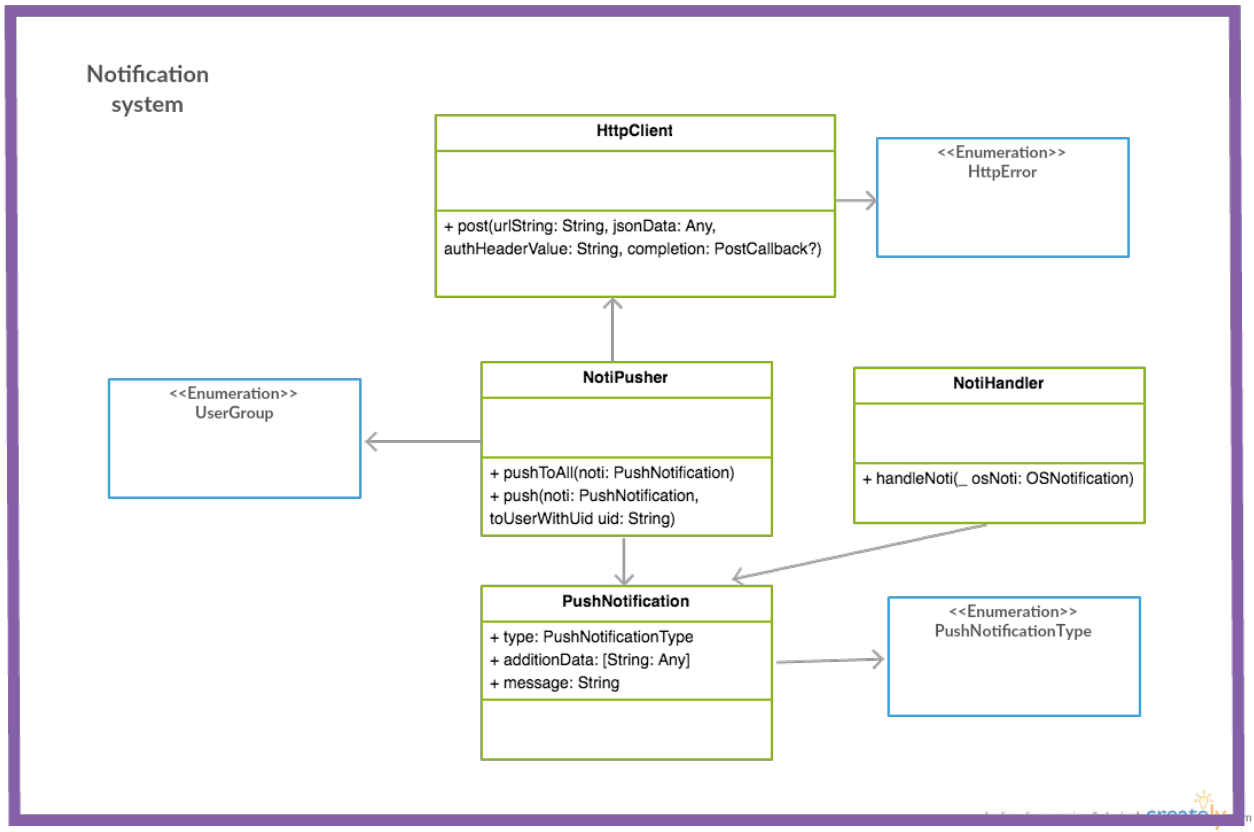
Any addition, deletion and changes to the team is synced with firebase and automatically handled by FIRDatabaseHandler class.

BACKEND MODULE

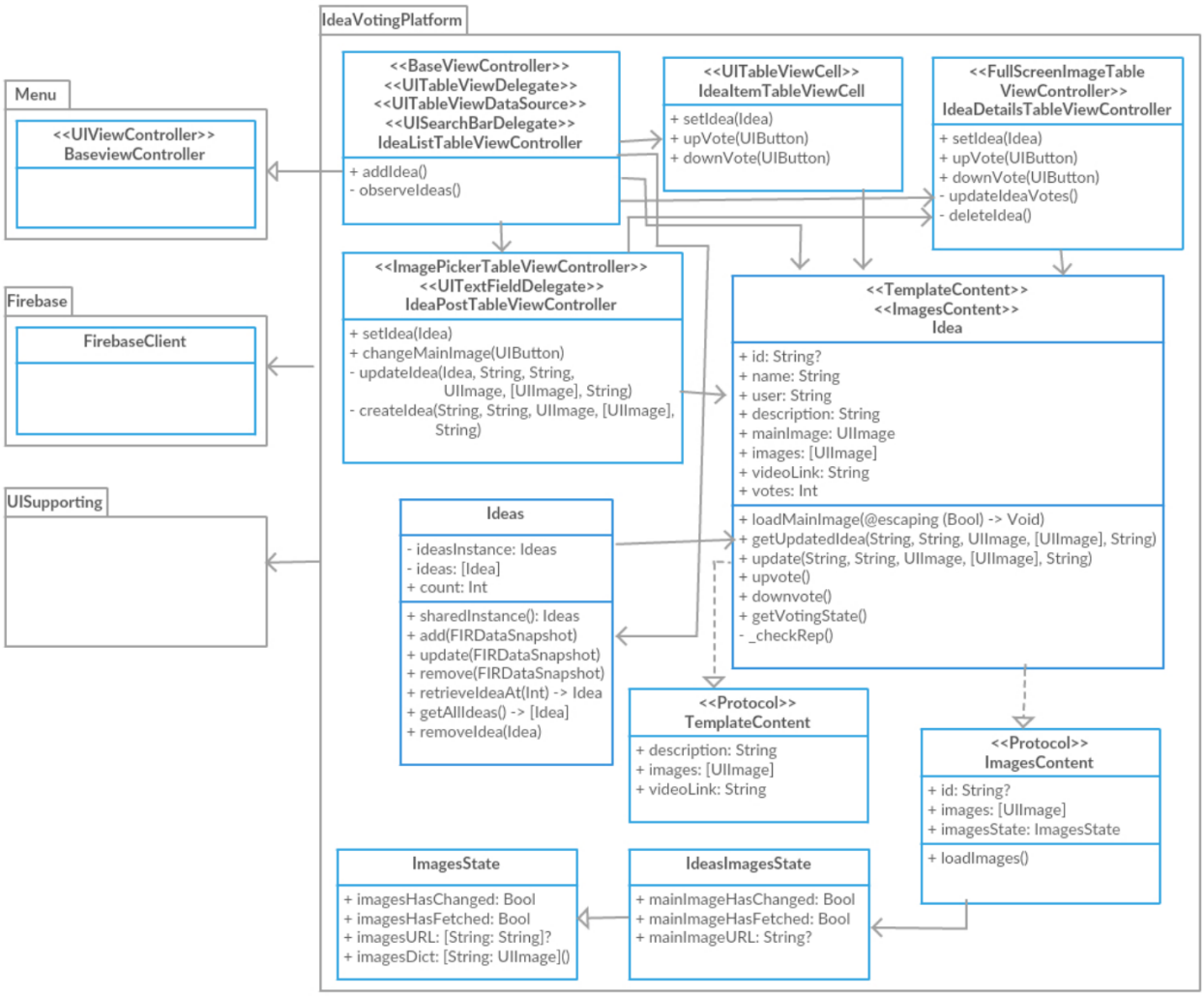


The backend primarily focuses of the Firebase Client, as the client handles most of the interactions of the App's various features with the backend storage. There are also a number of static helper Utility classes that assist in code reuse and system level constants.

NOTIFICATION MODULE

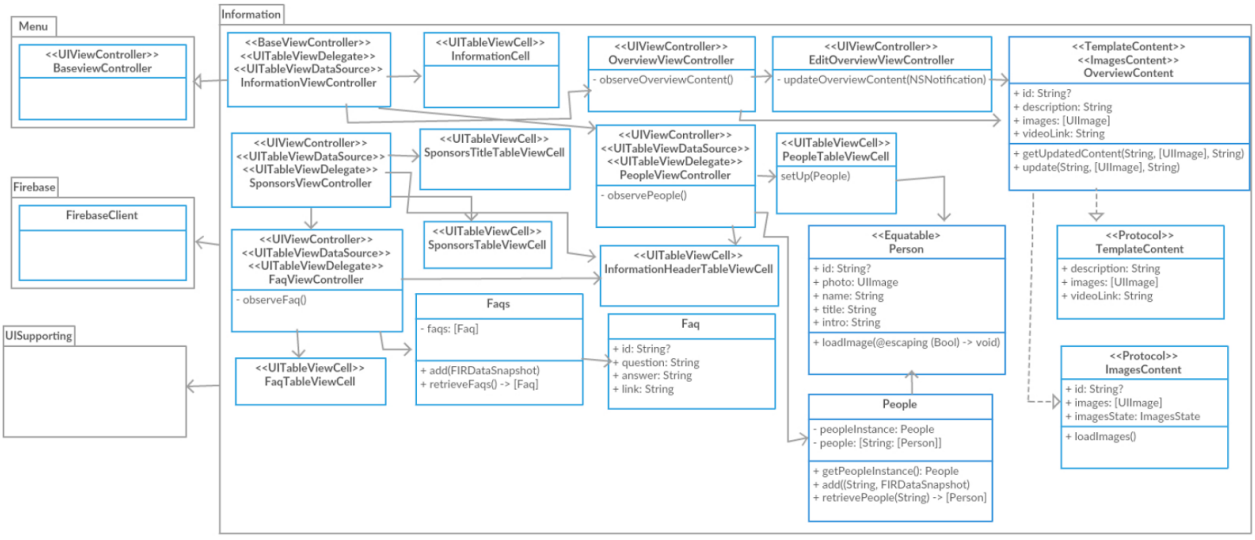


IDEAS MODULE



We realized that displaying and editing ideas and overview content are very similar. Both Idea and OverviewContent contain a description, some images and a videoLink. Therefore, to improve extensibility and reusability, we extract out those similar parts.

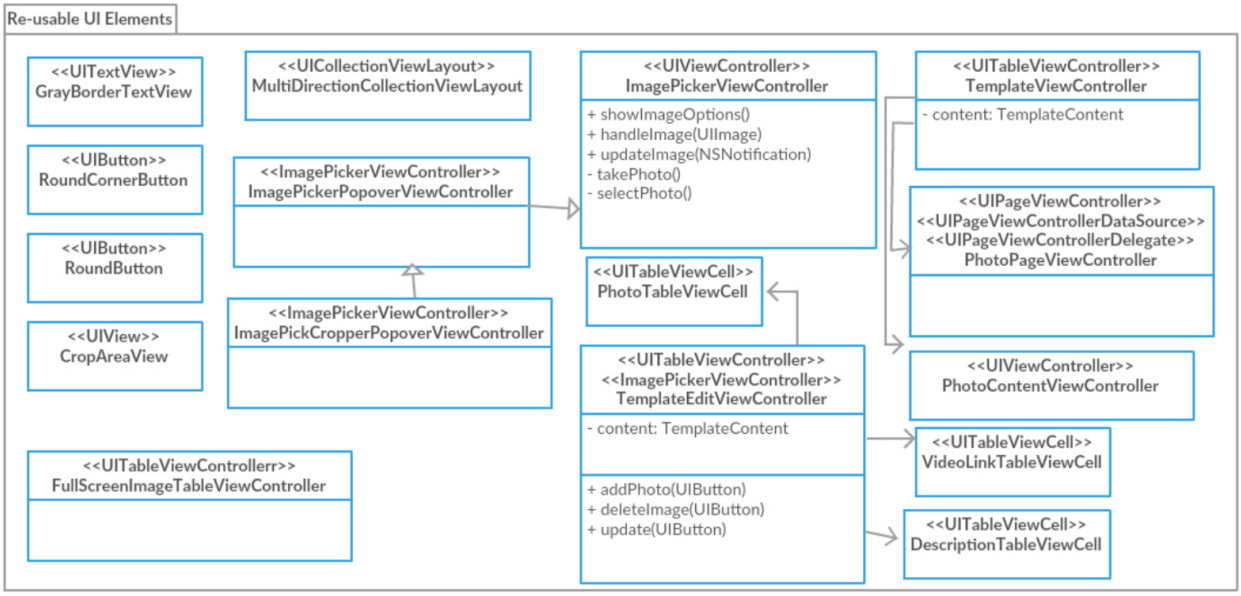
INFORMATION MODULE



In the aspect of view and view controller, we write a TemplateViewController, a view controller which can display textual description, images and embedded video, and TemplateEditViewController, a view controller which support users to edit description, images and video link. Therefore, when we implement features for idea and overview, we can simply embed corresponding view controllers in a container view, and don't need to repeat again.

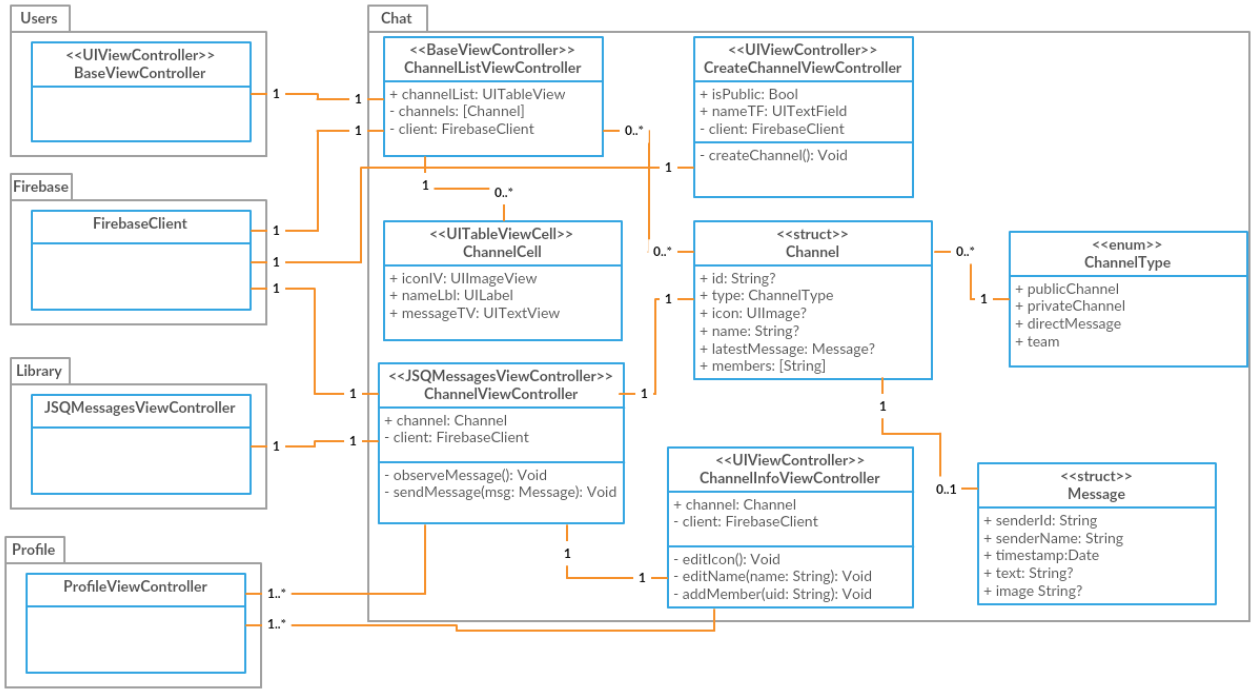
In the aspect of model, we introduce protocols ImagesContent and TemplateContent, which will be used by TemplateViewController and TemplateEditViewController to load as well as upload images, and display content.

RE-USABLE UI ELEMENTS



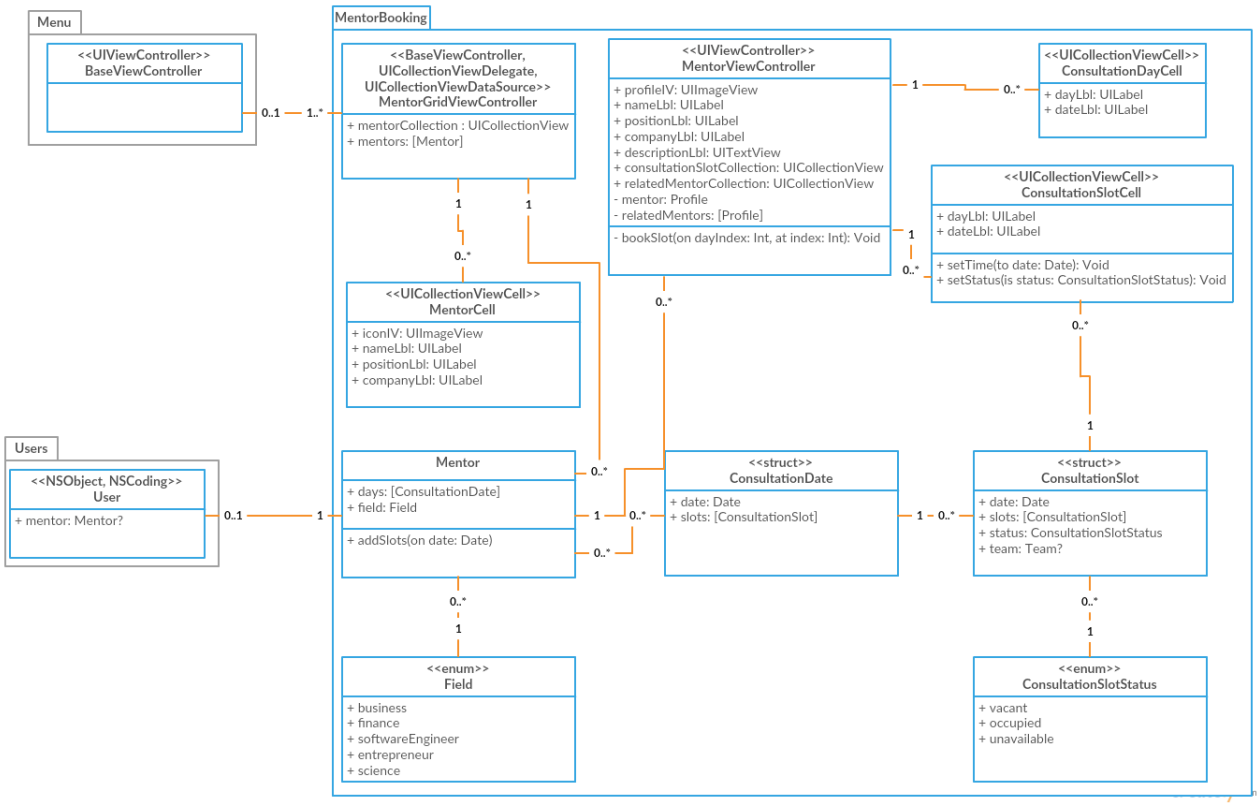
Classes in Re-usable UI elements are used in multiple components of the App. For example, ImagePickPopoverViewController, a view controller that support users to select an image or take a photo, is used in Chat and TemplateEditViewController. Its subclass, ImagePickCropperPopoverViewConroller, a view controller that support users to crop a selected image, is used in Signup, Profile, Idea, Event, and Chat. By using those re-usable UI elements, we improve code reusability, extensibility and follow DRY principle.

CHAT MODULE



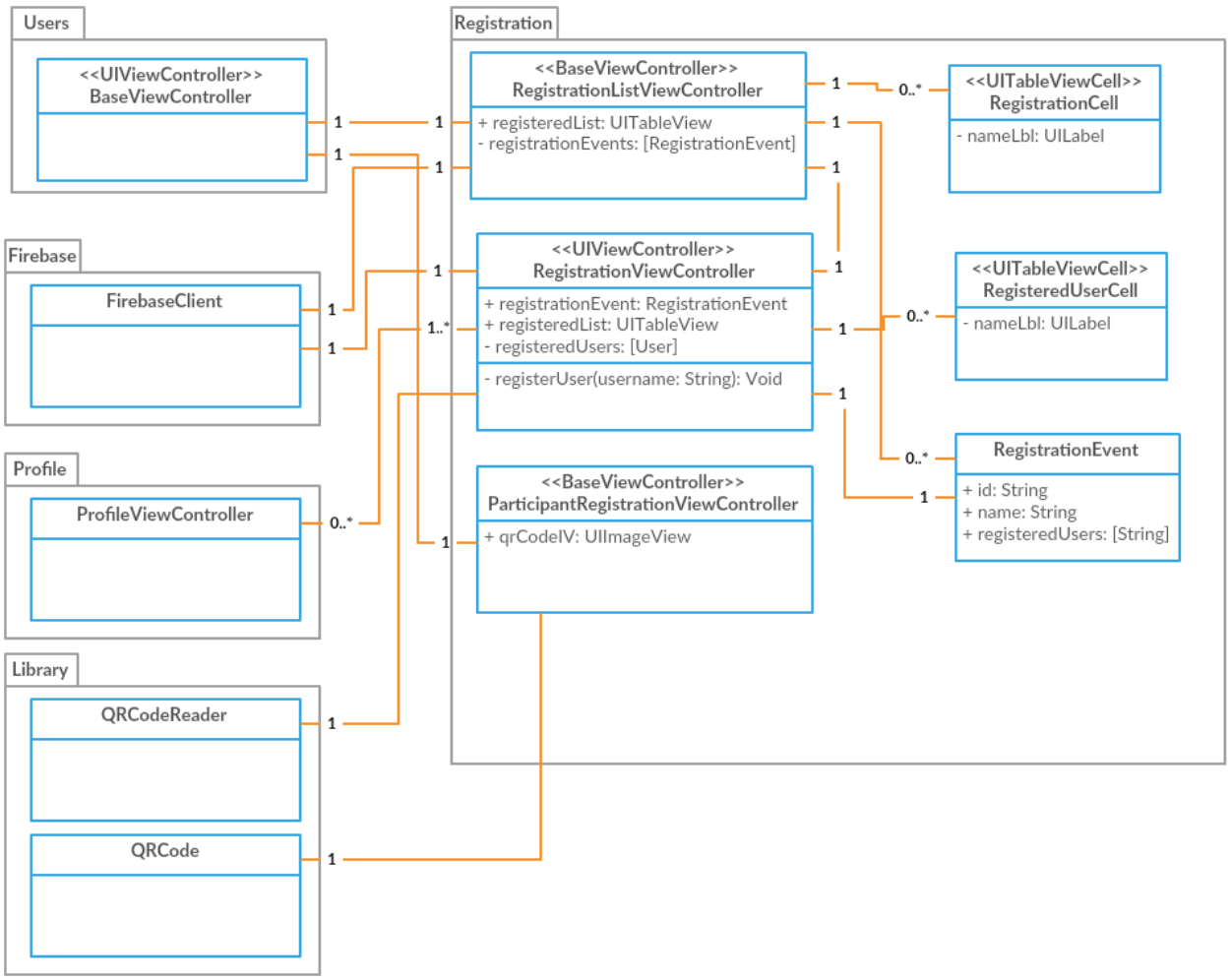
The use of a ChannelType enum was not originally intended, originally we simply had only a members array and we would “infer” the type from there. For example, an empty members array would mean a public channel, an array of 2 would mean a direct message and an array of 3 or more would mean a private group. Eventually, we decided that this was too cumbersome, and susceptible to bugs and abstracted out an explicit specification of ChannelType. This has also allowed us to easily extend the handling of different types of channels, for example we added the Team channel that functions like a private channel but disable any group info editing features. This could only be easily implemented because we abstracted the ChannelType allowing for easy extensibility.

MENTOR BOOKING MODULE



The things to be noted here is the use of ConsultationDate, ConsultationSlot and ConsultationSlotStatus. The initial idea was for Mentor to simply hold an array of Consultation Slots and each Consultation Slot would simply have a Bool of whether it was booked or not. Issues then arose of ordering the Slots, since the slots were not organised by dates, it would be hard to then display them by days since you would need to manually check if there were slots on a day, or count the number of slots in a day for the UICollectionView. We then considered used a dictionary of [Date: [ConsultationSlot]], while able to organise the Consultation Slots into days, the dictionary did not provide the ability to sort by date since dictionaries are unordered collections. Eventually we settled on creating a ConsultationDate struct that would have the date as a property that it could be sorted by, and slots as an array within it. By doing so, we also leave room for extensibility should we want to put in additional functionality specific to Consultation Dates. We also decided to abstract the status of Consultation Slots from a Boolean to an Enum, because we wanted to have the flexibility to extend it with multiple different types of statuses if needed in future. In particular we thought that it would be simpler if all mentors had the same slots on a day, but if they were unavailable to mark a slot as unavailable rather than simply not display it.

REGISTRATION MODULE



The Registration Module has two initial View Controllers, RegistraitonListViewController is only for Organisers while ParticipantRegistrationCiewController is for all other users. RegistrationViewController uses the QRCodeReader Library to scan QR Codes and ParticipantRegistrationViewController uses QRCode to display QR Codes.

Reflection

EVALUATION

Successes of development is that we were able to achieve most of the features that we set out to do, especially despite falling behind time during the last few weeks, we were able to catch up during the final sprint and also ensure that our App was properly refined and in good working condition. Overall, the team is satisfied with the App and feels that it has achieved appropriate levels of refinement and is relatively bug free.

Integration of the different components was done relatively well, while initially they were worked upon separately they have come together as a cohesive package with cross-linking across features for things like Profile and Chat. This is also testament to the reusable design that we adopted in creating our View Controllers to ensure that they are very easily reusable simply by passing in the correct object. I feel like the way we designed the View Controllers is to be cohesive and as independent from one another as possible, so as to easily facilitate reusability. The ease of reuse for ProfileViewController and ChannelViewController, are testament to that. Any outside feature can simply call a segue to it, pass the appropriate User and Channel class to display and it works automatically. The proper use of the Navigation Controller through multiple storyboards has also let us easily manage navigation without the need to manually handle each action.

Failures of development may be the reliance on Firebase and the network, due to the asynchronous nature of calls there may be certain unexpected behaviour due to network congestion and then a late callback. While we worked to minimise any and all of such bugs we came across, we cannot be fully sure then none exists anymore.

The largest mistake would be initially designing for a local database without considering the implications of a networked one from the backend. Due to our lack of expertise in backend, we assumed that we could simply replace the local code with that of the backend ones while re-using the same interface. This was not so for Firebase, and thus we ended up redoing a great chunk of the code, letting the time and effort of doing local persistent storage going to waste. Hence our project's logic code was almost rewritten from scratch around 2 weeks before StePs.

LESSONS

I think scheduling was our biggest mistake and lesson learnt, in the initial stages we did not put in much thought into scheduling. Haphazardly putting together a rough schedule based on what we thought was easier and more difficult, doing the easier ones first. This turned out to work against us since we had not considered that our work for other modules would ramp up towards the end as well. Leading us to fall behind schedule particularly in the last few weeks.

As mentioned in the evaluation, the concept of working locally then transitioning to a backend really backfired against us. Causing us to do almost double the work having the entire initial local portion scrapped and then having to rework the App from the ground up to work with Firebase. This cost us a great deal of time, particularly towards the end and we also had to go through the learning curve or understanding and working with Firebase. To do things differently a second time round, we would definitely have started by setting up the Backend in the first week and then directly working off the backend instead of doing what we did. This would have saved us a great deal of time, and the learning curve for the backend would have been when we had more time at the start of the project.

Regards to known bugs and limitations, primarily the bugs typically come from the handling of Firebase callbacks. Since sometimes the callbacks do not work as intended. Particularly for images and the updating of data, certain bugs that are more common are typically the ones involving the updating of an image and when going back the image is not updated until you refresh the page by jumping in and out. This seems to be because the firebase callback for value changed is not being called and certain instances, though we have not been able to pinpoint when and why. For the table in IdeaPostTableViewController, there are issues with scrolling as well. When the description TextView is focused, the table will scroll to the bottom.

In terms of Limitations, I believe that our App does achieve most of the specifications with exception to a few areas. The first is Push Notifications, we have not been able to get a proper notification system up and running as well as implementing a notification system that we had originally intended. The second is Team Administration, by right the owner of the team should be able to review members and approve them before they join a team. The current procedure simply involves anyone being able to join and quit a team on a whim. We also intended to have private teams, that can only be joined on invite, but eventually it was not implemented.

Appendix

FORMATS

Most of the formats are either implied or handled by our libraries, for example for images they should all have a “gs://“ at the front since they are loaded into the Firebase Database. However this is all handled automatically by the API, and we have performed the appropriate defensive checks to ensure they are correct. In addition, since we are using Firebase, a NoSQL Database, we do not need to have any sort of specific format for our backend calls, we just need to be award of the hierarchy of the database. Any models being fed to the Firebase database, should also be in dictionaries of either numbers or Strings since Firebase cannot deserialise classes natively. Most user inputs should be non-empty, and we have put in checks to ensure that it is the case.

MODULE SPECIFICATIONS

CHAT SYSTEM

Channel

Channel is a class used to represent a Channel in the Chat System

Specifications:

- id: Channel ID as given by Firebase
- type: The Type of Channel that is it using the ChannelType enum
- icon: An Optional UIImage to represent the Icon for the chat
- name: The Name of a Channel, not present if it is a direct message
- latestMessage: The Latest Message in the Channel
- members: An Array of Member's UIDs (String)

Representation Invariant:

- If the Channel is not a Direct Message, it should have a Name

ChannelType

ChannelType is an class used to represent the type of channel

Specifications:

- publicChannel: Open to All Users (Only Admin can create)
- privateChannel: Open only to Members
- directMessage: Only between two people
- team: Same as Private Channel except Channel Info cannot be edited

Message

Message is a class used to represent a Message in the Chat System

Specifications:

- senderId: User ID of the Sender
- senderName: Username of the Sender
- timestamp: The Date that the message was sent
- text: The text of the message
- image: The URL of the image for the message

Representation Invariant:

- The message should have either text or image, both cannot be null together

EVENTS

Event

Event is a class used to represent a Events in the Events System

Specifications:

- id: ID of the Event as given by Firebase
- image: Image Icon for the Event
- name: Name of the Event
- startDateTime: Date for the start of the event
- endDateTime: Date for the end of the event
- shortDesc: A Short Description to display on the List/Calendar View
- description: A Full Description to display in the Details View
- venue: Vanue for the Event
- comments: An Array of comments for that event

Comment

Comment is a class used to represent a Comment in an Event

Specifications:

- authorID: User ID of the User who posted the comment
- timestamp: Date that the timestamp was posted
- text: Contents of the Comment

IDEA SYSTEM

Ideas

Idea is a class that represents an Idea in the Idea System

Specifications:
- votes: A dynamic variable to tally what is the idea's voting score
- id: The ID of the Idea as given by Firebase
- name: Name of the Idea
- user: User ID of the Idea Creator
- description: Description of the Idea
- mainImage: Main Image Icon of the Idea
- images: Image Array of supplementary images as details
- videoLink: Link to an embedded video
- imagesState: An enum representing the status of the loading of images in Ideas
- upvotes: A Set containing the User IDs of people who upvoted
- downvotes: A Set containing the User IDs of people who downvoted

Representation Invariant:
- There should not be any users who both upvoted and downvoted

ImagesContent

ImagesContent is a protocol that represents a content which has multiple images.
It has some default implementation to fetch images.
Specifications:
- id: The ID of the content, to be used for notification
- images: Image Array of supplementary images as details
- imagesState: Store whether images have been fetched or changed

TemplateContent

TemplateContent is a protocol that represents a template content, which will be shown in TemplateViewController.
Specifications:
- description: Description of the Content
- images: Image Array of supplementary images as details
- videoLink: Link to an embedded video

ImagesState

ImagesState is a class to indicate whether images have been fetched or changed.
It helps to achieve a better performance for loading and uploading images.
Specifications:
- imagesHasChanged: Whether images have changed

- imagesHasFetched: Whether images have fetched
- imagesURL: A link to images
- imagesDict: Fetched images

Ideas

Ideas is a collection of all ideas object which represents a local cache of all ideas.
It will also communicate with Firebase client, and update the model
correspondingly.
It follows singleton pattern.
Specifications:
-ideas: A mutable [Idea] object which represents all idea

INFORMATION SYSTEM

OverviewContent

OverviewContent is a class that represents an OverviewContent in the Information System
Specifications:
- id: The ID of the content, which is to comfrom protocol ImagesContent
- description: Description of the Idea
- images: Image Array of supplementary images as details
- videoLink: Link to an embedded video
- imagesState: Store whether images have been fetched or changed

Representation Invariant:
- The description cannot be empty.

People

People is a collection of all person object which represents a local cache of all person in Information System. It follows singleton pattern.
Specifications:
- people: A mutable [String: [Person]] object which represents all people

Person

Person is a class that represents a person in the Information System
Specifications:
- id: The ID of the Person as given by Firebase
- name: Name of the Person
- photo: Photo of the Person
- title: Title of the Person

- intro: Intro of the Person
Representation Invariant:
- The name of the Person cannnot be empty

Faqs
Faqs is a collection of all Faq object which represents a local cache of all faq
in Information System.
Specifications:
- faqs: A mutable [faq] object which represents all faq

Faq
Faq is a class that represents a faq in the Information System
Specifications:
- id: The ID of the Faq as given by Firebase
- question: The question of the faq
- answer: The answer of the faq
- Link: The link of the faq
Representation Invariant:
- The question and answer of the faq cannot be empty

MENTOR SYSTEM

Mentor
Mentor is a class used to hold mentor specific variables in User
Specifications:
- days: An Array of Consultation Dates that contain Consultation Slots
- field: Field an enum of work fields, user to group related mentors together by field

ConsultationSlot
ConsultationSlot is a struct used to hold details about a Consultation Slot
for a Mentor within ConsultationDate

Specifications:
- startDateTime: The Date when the slot starts
- status: An enum of ConsultationSlotStatus detailing the status of the slot
- team: If booked, it is the Team ID of the team that booked it
Representation Invariant:

- If the slot is booked, it should have a team ID

ConsultationDate

ConsultationDate is a struct within Mentor used to hold information about ConsultationSlots for a day

Specifications:

- date: Date whose ConsultationSlots are contained
- slots: An array of ConsultationSlots for the day

ConsultationSlotStatus

ConsultationSlotStatus is an enum that represents the status of a ConsultationSlot

Specifications:

- vacant: Slot is vacant and can be booked
- booked: Slot has been booked by a team
- unavailable: Slot has been marked by the Mentor as unavailable and cannot be booked

Field

Field is an enum that represents the field of a Mentor

REGISTRATION SYSTEM

RegistrationEvent

RegistrationEvent is a class that represents a Registration Event in the Registration System

Specifications:

- id: ID of the Registration Event as assigned by Firebase
- name: Name of the Registration Event
- registeredUsers: Array of user IDs who have been registered for that event

SOCIAL MEDIA

FBUser

FBRequest is a struct that helps with the request and deserialization of data from the Facebook API

Specifications:

- Directly accesses the Facebook Access Token that is valid only if user logged in
- Response contains a SocialUser with the deserialized data.

SocialUser

SocialUser is a struct that helps with the request and deserialization of data from Social Media, both Facebook and Google

Specifications:
- id: ID for the respective Social Media account
- name: Name on the Social Media Account
- email: Email from the Social Media Account
- type: AuthType representing who is the Authentication Provider (FB, Google etc.)

AuthType

AuthType is an enum representing the different possible Authentication Providers.
The raw Value string is what Firebase identifies Auth Providers as.

TEAM SYSTEM

Team

Team represents a team within the Team system

Specifications:
- members: A String Array of the UIDs of Members
- name: The Name of the Team
- lookingFor: The type of people the team is looking for
- isPrivate: Whether the team is a private team
- id: ID of the Team as assigned by Firebase
- tags: String Array of Tags of the existing Skill Set of the Team

USER SYSTEM

User

User contains the essential information about the user account, such as email, type, team as well as optional values depending on the user type like mentor

Specifications:
- email: The email of the User
- profile: The Profile of the User
- type: The UserType
- team: The User's Team
- mentor: Mentor Specific Variables like ConsultationSlot
- uid: ID for the User as assigned by Firebase
- favourites: An array of user IDs that have been favourited by the user

Representation Invariant:
- Email must be valid
- Non-Participants must not have teams
- If not a mentor, the Mentor variable should be nil

Profile

Profile contains all the description and details about the User for his Profile Page
Specifications:
- username: The username of the User
- name: The Name of the User
- image: The Profile Image for the User
- job: The User's Job/Position
- company: The company where the user is currently working
- country: The country the user is from
- education: The user's education background
- skills: The user's skills
- desc: A short writeup about the user

UserTypes

UserTypes represents the type of the user, and supports multiple user types
Specifications:
- isParticipant
- isSpeaker
- isMentor
- isOrganizer
- isAdmin

NOTIFICATION SYSTEM

PushNotification

PushNotification store information necessary to send and received push notification.
The content of additionData is agreed between NotiPusher and NotiHandler.
Fields:
- type: represent the type of the notification
- additionData: additionData necessary to handle the notification.
- message: a string that will be displayed by the notification by iOS.

Invariant:

- The type of the notification decide the type, content and format of additionData
  + type announcement has and empty additionData
  + type message has additionData has only 1 key-value pair (Config.channelId: id)

TEST CASES

- Black box:
  - **Initial View:**
    - If a user has logged in, after opening the app, the user should see the homepage.
    - If a user hasn't logged in, after opening the app, the user should be prompted to sign up / log in.
    - Click sign up button: Navigate to sign up page.
    - Click log in button: Navigate to log in page.
    - Click Facebook button: Prompt for Facebook Login
    - Click Google button: Prompt for Google Login
  - **Sign Up:**
    - The user should can see a sign up form, where the user can upload profile image and fill in information about full name, username, email, password, country, job, company, education, skills and description.
    - The user can tap the back button to go back to the previous page, or tap the login button to navigate to the login page.
    - The user can tap on the profile image to change profile image. The user should can select a photo from album or take a photo.
      - If user choose to take a photo but the device doesn't have a camera, the action should fail and the user will be prompted.
      - Otherwise, after the user choose a photo, the user will be asked to crop the image to circle and the profile image will be updated.
    - Full name, username, email, password, country, job and company should only support single-line input. After the user presses "Next" in the keyboard, next item in the form should be focused.
    - Skills and description should support multi-line input.
    - User's input in the "password" row should be secured text.
    - When a user select the "country" row, the user should can see a picker that shows a list of country. The user should can choose a country by tapping an item in the picker or tapping the "Done" button.

- When a user tap around, the keyboard should be hidden.
- The keyboard should have a toolbar to allow jumping to next or previous text field and a Done button.
- The sign up button should be enabled only after the user has filled in full name, username, email, password, country, job, company, education and skills (i.e. only profile image and description are optional).
- After tapping the sign up button:
  - If the email address is invalid, sign up should fail and the user will be prompted.
  - If the password is invalid, sign up should fail and the user will be prompted.
  - If the email address has been taken, sign up should fail and the user will be prompted.
  - If there's a problem signing up (such as network connection problem), sign up should fail and the user will be prompted.
  - Otherwise, the user will log in and should can see the home page.Log in:
- A user should can see a login form, where the user need to enter email and password.
- The user can tap the back button to go back to the previous page, or tap the sign up button to navigate to the signup page.
- User's input in the "password" row should be secured text.
- When a user tap around, the keyboard should be hidden.
- The login button should be enabled only after the user has entered email and password.
- After tapping the login button:
  - If the email is invalid, login should fail and user will be prompted.
  - if the email does not match any account, login should fail and user will be prompted.
  - If the password and email does not match, login should fail and user will be prompted.
  - If there's a problem logging in (such as network connection problem), log in should fail and the user will be prompted.
  - Otherwise, login should success and the user can see the home page.
- **Profile:**
  - A user should can see the profile page after tapping the area around profile image in menu.

- The profile page should show information about the user: profile image, name, team if the user is a participant, username, country, job, company, education, skills and description.
- The user can tap the back button to go back to the previous page.
- The user can tap on the profile image to view the image full screen.
- The user should can edit profile. After tapping the edit button, the user should can see a form that is similar to the signup form, except that:
  - The form is filled in with the current information about the user.
  - There's no email and password entries.
  - The user cannot navigate to login page.
  - After tapping the done button, the user will go back to profile page.
- The user can change password by tapping the "Change Password" row.
- The user will be prompted to enter the current password and new password.
- If the current password is not correct, change password should fail and the user will be prompted.
- If the new password is invalid, change password should fail and the user will be prompted.
- If there's a problem changing password (such as network connection problem), change password should fail and the user will be prompted.
- Otherwise, change password should success and the user will be notified.

- **Event System:**
  - A user should see the "Schedule" option when tapping on the menu button. Upon tapping on the "Schedule" button, he/she should be presented with a table view listing all the events for the day.
  - Days with events will be highlighted in gray dot, upon clicking on the days with events, he/she will be directed to the event schedule page, which highlights the details of the events.
  - Clicking on non-highlighted dates will not direct the user to the event schedule page.
  - A user should be able to tap on the > and < button to scroll to the event listing before or after the day.
  - Upon tapping on a particular event listed, the user will be presented with the details page of the event, including the event name, date time, venue, detailed description.
  - On the event details page, user can comment below the event as well. Once he/she presses "add" button, he/she will see the comment together with the username and

profile picture, similar to any commenting system currently existing, such as wechat, Facebook, etc.

- Any user logged into the system should be able to view all other users' comments and replies for the particular event.

- **Team Registration:**
  - User should be able to tap on the "Teams" button listed in the menu page, and be directed to the teams page.
  - User can see all the currently public teams on the teams page.
  - User can tap on the team to learn more about the team, including their members and description, and the type of person that the team is looking for, and skill set possessed by the team.
  - User can view the `Edit` button and `chat` icon if they belong to the team, clicking on Edit button will direct them to the event editing page, while clicking on chat icon will direct them to chat page.
  - User can also tap on "Request to join" button below each team description if the team is not yet full. A request will be sent to the team leader, who will decide whether the person can join the team. User's profile in the slide menu page will also reflect the correct team number.
  - User can quit team at their own discretion. If no member is present in the team, then the team will be automatically deleted, and should not appear again in the team registration page.
  - User can tap on the search field, and search the team info based on the keywords.
  - User can create their own team by tapping on the "Create" button at the top of the page. If the user created a team, then he automatically becomes the team leader.
  - If the user taps on the "Create" button, then he will be presented with a team creation page, in which the user can enter the team name, the skills possessed by the team and the type of person that the team is looking for.
  - When user is creating or editing the team, several checks are put in place to ensure that team name cannot be empty, skill tags cannot be empty, skill tags cannot have duplicated content, duplicate check is not case sensitive, the skill that the team is looking for also cannot be empty.
  - If the user does not belong to the team, he/she will be prompted with an error message upon clicking `Done` button.

- Tap on the team member icon in the TeamInfo page will direct the user to the member profile. If the member is the current user, he/she should be able to edit his profile.
- The user can also choose to make his/her team public or private. If the team is private, then it will not be listed under the teams page.
- If the user already has a team, then he/she cannot send request to join other teams, nor to create his/her own team.

- **Mentor Booking:**
  - User should be able to view a list of all mentors
  - User can view the details of a mentor by tapping on a Mentor Cell
    - Expected Behaviour:
      - App brings the user to the Details Page of the Mentor
      - Details page should have the correct information about the mentor
      - All Consultation Slots for the Mentor are shown
        - Red indicates occupied, Green indicates vacant and grey indicates unavailable
      - Related Mentors are shown at the bottom
  - User can return to the list of mentors by pressing the bar button from the Mentor Details Page
  - User can view the mentors profile by tapping on the description
    - Expected Behaviour:
      - Brings user to the Mentor's Profile Page
  - User can chat with the mentor by tapping on the chat button at the top right
    - Expected Behaviour:
      - Starts a direct message chat with the mentor
  - User should be able to see all Consultation Dates by scrolling vertically on the Consultation Slots section
    - Expected Behaviour:
      - Scroll area for the TableView allows the user to scroll up and down, stopping in the correct positions and bouncing back when the user exceeds bounds
  - User should be able to see all Consultation Slots on a Date by scrolling horizontally on the Consultation Date
    - Expected Behaviour:
      - CollectionView should display cells horizontally and only allow for horizontal scrolling

        - All slots should be viewable by scrolling horizontally
    - User should be able to interact with Consultation Slots by tapping on it
        - If he is a participant
            - Expected Behaviour:
                - Tapping on a red/grey slot has no effect
                - Tapping on a green slot prompts the user to book the slow
        - If he is a Mentor
            - Expected Behaviour:
                - Tapping on a red slot shows which team booked it with the option to jump thought that Team's Profile
                - Tapping on a green/grey slot prompts the user if he wants to change the status of the cell
        - If he is any other role
            - Expected Behaviour:
                - Nothing happens
    - User should be able to see a related mentor by tapping on the related mentor cell
        - Expected Behaviour:
            - The Related Mentor's Detail Page will appear
- **Chat System:**
    - User should be able to view a list of all available chats by scrolling up and down
        - Expected Behaviour:
            - All available chats to the user are displayed
    - User should be able to create a new chat by tapping on the new chat icon
        - Expected Behaviour:
            - If not an organiser
                - Two options would be presented a Direct Message or Private Channel
            - If an organiser
                - Three options would be presented, the previous two plus a Public Channel
    - User can tap on creating a Direct Message, and enter a username
        - Expected Behaviour:
            - If username is valid, enters a direct message with the user
            - If username is not valid, shows an error and prompts the user to enter the username again
    - User can tap on creating a Private Channel
        - User tries to tap Done

    - Expected Behaviour:
        - Error because no name entered
    - User enters a Name
        - Expected Behaviour:
            - Error because only one member
    - User adds another member
        - Expected Behaviour:
            - Successfully create a private channel
- User should be able to invite any user to his chat through the invite field
    - Expected Behaviour:
        - Autocomplete suggest usernames when the user types usernames into the field
        - Selecting a valid username adds the user into the chat
        - Typing an invalid username shows an error message
- User should be able to tap on a chat in the available chat and view it
    - Expected Behaviour:
        - Chat screen appears
        - Most recent messages are at the bottom of the screen
- User should be able to scroll and see past messages in the Chat Screen
    - Expected Behaviour:
        - More messages load as the user scrolls upwards
- User taps on the Camera Icon
    - Expected Behaviour:
        - Prompt to select a photo from camera roll or take a photo appears
        - Selecting a photo then displays it in the textfield area
- User taps on the Send button
    - Expected Behaviour:
        - Message sends what is in the textfield if it is not empty
- User taps on the chat icon in the top right
    - Expected Behaviour:
        - If direct message, jumps to that user's profile
        - Else shows Channel Info Page
- Channel Info Page Appears
    - Expected Behaviour:
        - If Public Channel, only icon and name are visible, nothing available to edit
        - If Team, member list is also visible but no way to edit

        - If Private Channel, all editing functions are present
    - User taps on the Chat Icon, Selects Prompt to Select photo and selects a photo
        - Expected Behaviour:
            - Chat Icon Updates
    - User taps on the Name, Enters a new name in the Popup
        - Expected Behaviour:
            - Name Updates
    - User adds a new member to the chat via username
        - Expected Behaviour:
            - User is added to the chat, and can send messages
- **Information:**
    - A user should can see the information page after tapping “Information” in the menu.
    - The user can click on “Overview”, “Speakers”, “Judges”, “Sponsors”, “Organizing Team” and “Frequently Asked Questions” to get information about the event.
    - In Overview, the user should can see description, images and a video. The user should can tap any image to view image full screen.
    - In Speakers, Judges and Organizing Team, the user should can see the name, title, description and photos of those corresponding people. The user should can tap any photo to view photo full screen.
    - In Sponsors, the user should can see the categories and logos of sponsors. A user should can go to the sponsor’s homepage by clicking the logo.
    - An organizer can edit Overview (only support edit Overview since Dr Leong Wai Kay mentioned that the functionality is not important) . The organizer should can edit description, add, delete and rearrange images and update video link.
- **Idea Voting:**
    - A user should can see the idea voting platform page after tapping “Idea Voting” in the menu.
    - The user should can see a list of ideas.
    - The user should can search for a idea.
    - The user should can vote for the idea. The user can undo vote by clicking the same button used to vote. To change a vote from up to down, click the down arrow, and vice versa.
    - The user should can click on a row and will be directed to the details page of the idea. Inside details page, the user should can see the name of the idea, the name of the idea owner, description, images and video of the idea.

    - The user should can tap any image to view image full screen.
    - The user should can tap the idea owner's name to view profile.
  - A participant should can create an idea. The participant should can add idea name, description, images and video link.
    - The user can create another new idea after the current idea is uploaded to the backend
  - The idea owner should can edit and delete idea.
    - The idea owner can only edit an idea after the idea's images are loaded.
  - The idea list page should be updated when there's any change.
  - The idea votes in idea details page should be updated when there's any change.
  - The user should can click on a row and will be directed to the details page of the idea. Inside details page, the user should can see the name, team, description, images and video of the idea. The user should can comment on the idea.
  - An participant who belongs to a team should can create an idea. The participant should can add idea name, description, images and video link.
- **Log Out:**
  - A user can log out from the menu
  - After tapping Logout the user should be presented with a prompt before actually logging out
- **Registration:**
  - If the user is not an Organizer, the Registration page should only show a QR Code
  - If the user is an organiser
    - He can create a Registration Event by tapping on the Plus Icon and entering a name
      - Expected Behaviour:
        - A Registration Event would be created
      - He can enter the Registration Event
      - In the Registration Event, he can add a user using his username in the textfield and click add
        - Expected Behaviour:
          - If the username is valid, it would be added into the table
          - If the username is invalid, an error prompt would appear
      - He can tap on the QR Code button at the top right
        - Expected Behaviour:
          - The App should transition to QR Code Scanning Mode
      - He can scan a user's QR Code

        - Expected Behaviour:
            - If the user has not been registered, he would be added into the table
        - If he has an error appears
- **Glass box:**
    - Utility:
        - Test email validation (Utility.isValidEmail): Using partition tests
            - valid email should return true
            - invalid email (invalid @, invalid local part, invalid domain) should return false
        - Test password validation (Utility.isValidPassword): Using partition tests
            - valid password (length >= 6) should return true
            - invalid password (length < 6) should return false
    - Team:
        - Test add member: use partition tests
            - able to add member if member id is not nil
            - not able to add member if member id is nil
        - Test remove member: use partition tests
            - able to remove member if the member is inside the team with a valid id
            - not able to remove member if member either do not reside in the team, or do not possess a valid id
        - Test contains member: use partition tests
            - return true if the member is indeed inside the team
            - return false if the member does not belong to the team