swee chat

CS3217 FINAL REPORT

GROUP 10

AGNES NATASYA

CHRISTIAN JAMES WELLY

KEVIN LIM YONG SHEN

NGUYEN CHI HAI

Table of Contents

Table of Contents 1

Overview 4

Features & Specifications 5

- Authentication 5
- Modules 5
  - Permissions 5
- Chat Rooms 5
  - Private Chat Rooms 5
  - Group Chat Rooms 5
  - Forum Chat Rooms 6
- Messages 6
- Cryptography 6
- Notifications 6
- Caching 6

User Manual 7

- Logging In 7
- Home 7
- Settings 8
- Modules 8
- Chat Room 10
  - Private Chat Room and Group Chat Room 10
  - Forum Chat Room and Thread 11
  - Star Chat Room 11
- Messages 12
  - Type of Messages 12
  - Replying of messages 12
  - Editing and Deletion of Messages 12
  - Liking of Messages 13

Code Design 13

- Top Level Organisation: MVVM 13
- Event Handling 13
- Cryptography 14

Background: Public-Key Cryptography 14
Background: Authenticity 15
Signal Protocol 16
Pairwise Secure Channels 16
Group Messaging 18
Trade Offs 19
Implementation Details 20
Model-Cloud Sync 22
Message 27
Message Model 27
Message ViewModel 28
Chat Room 29
Chat Room is Model Concept and Not ViewModel Concept 29
Abstract Class and Not Class Inheritance or Protocol 30
Factory Pattern Translation between Architectural Components 31
Module Permission 32
Bitmasks implementation over Role enum 32
Authentication Module 33
Media Cache 35
How the cache works 35
Image Data 35
Video Data 36
Making MediaCache the responsibility ChatRoom 36
Have a application-wide media cache 36
Allow Classes that need access to the Cache Instantiate their own MediaCache 37
Dependency Inversion Principle 37
Notification 37
Automatic and Manual Navigation 37
Navigation Flow 39
Pagination 41
Replied Message 41
Dependency Inversion Principle (View) 43

Test Plan **43**
Strategy 43
Results 44

Reflection **45**

Evaluation 45
Lessons 45
Known Bugs & Limitations 47
Appendix **48**
Test Cases 48

Overview

Sweechat is a real-time messaging application that supports communication between instructors and students to build inclusive education communities.

Problems we want to solve:

- Difficult to navigate the plethora of different communication channels between instructors and students
    - Module announcements: Email and LumiNUS announcements
    - Forum questions: LumiNUS and Piazza
    - Real-time assistance: Chat applications, email
- Students prefer anonymity and tend not to use public channels to voice their doubts. Those who do end up frequenting forums tend to be stronger and more confident.

Our solution:

- Create an all-in-one platform that can support the communication needs of students and instructors. The application allows for private messaging, group chats, announcements, forum pages, and more.
- Develop a learning community by supporting real-time messaging for students to contribute questions and answers.

Features & Specifications

Authentication

Users can log into the SweeChat using a plethora of methods. Currently, Facebook and Google authentication are supported for the proof of concept that authentication methods can be easily extensible and flexible. Please note that to log in with Facebook, due to Facebook limitations on applications that are in the testing stage, the facebook account must be added as a tester first.

Authentication also supports cached login, thus if users do not log out, they will be automatically logged in even after the application exits and restarts.

Modules

Users can now create Modules and join modules. Upon logging in, users can create the module with a name or join a module using the module secret id.

Permissions

Modules offer different permission levels depending on whether you are a module owner or a student. Currently, users who created the module are able to create a Forum, and users who join the module (students) are unable to do so. Both module owners and students are able to create normal group chatrooms.

Chat Rooms

Once in a module, users will be able to communicate with other users that are in the module. This communication is done through chat rooms. At this point, we have three types of chat rooms: private chat, group chat, and forum posts, with an additional setting for starred chat rooms. Note that permissions can also be set for group chats so that the owner can set the permissions of other users, thus chat rooms can also double up as announcements pages for instructors.

- ### Private Chat Rooms

  Users can create private chats with other users inside the same module.

- ### Group Chat Rooms

  Users can create chat rooms with other users inside the same module. Chat Room creators can set the permissions of other users inside the chat room.

- **Forum Chat Rooms**

  Users can create forum posts and others can reply to the questions inside individual threads.

- **Starred Chat Rooms**

  Instructors can choose to star specific chat rooms on creation to highlight them to other users inside the module. This allows instructors to highlight the formal channels of communication with students.

Messages

Users can send a plethora of different messages types. Currently, we support text message, image, video, and canvas drawings. Users can also reply to messages inside a private chat room and group chat room by double tapping on them.

Cryptography

All contents of messages between users in chat rooms are encrypted from end to end. This means that not even SweeChat can read their messages.

Notifications

Users will see real-time push notifications upon receiving messages from any of their chat rooms.

Caching

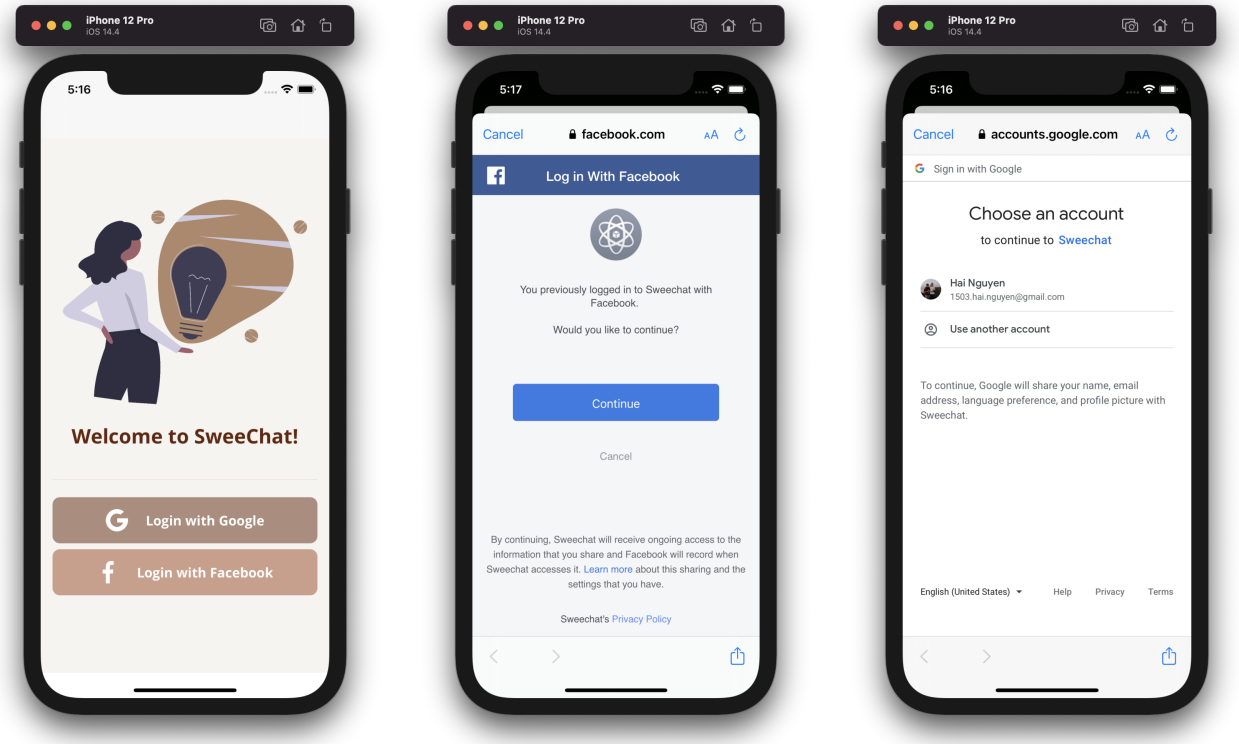
Users can open SweeChat without internet connection and still have access to messages that have been loaded before. Users will also be able to play videos or look at messages loaded before.

User Manual

Logging In

Upon entry, the user can log in via either Google or Facebook:

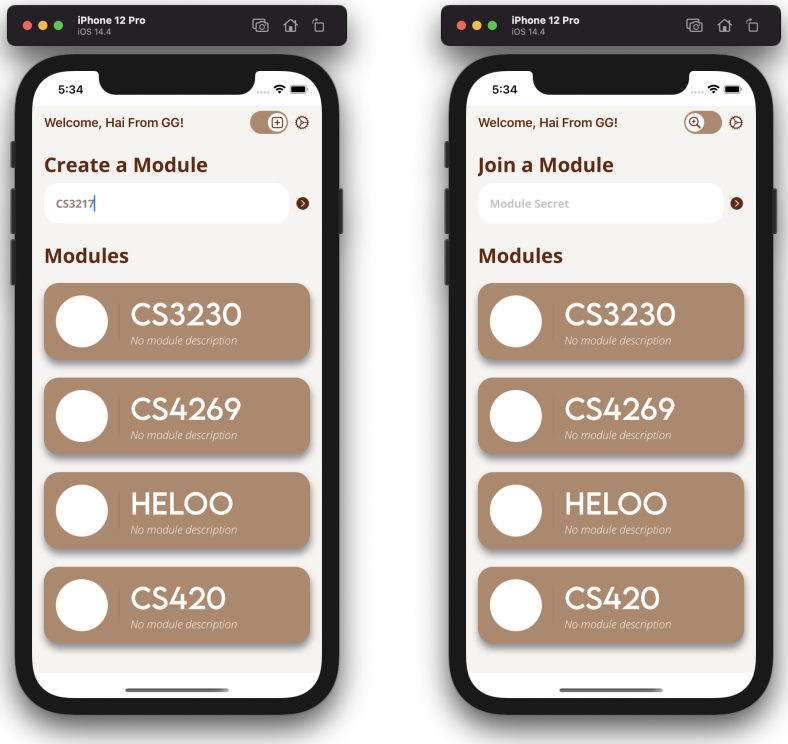
- Google: Users can sign in with their google mail account.
- Facebook: Users can sign in with their facebook account. However, in the time being due to facebook limitations, if users want to sign in with facebook, the user must be added to a list of testers.



After log in, the user will be directed to the Home page.

Home

After the login, the user will see all of the user's enrolled modules. Users can also choose to create new modules or join other modules using the module secret using the switch at the top of the screen.



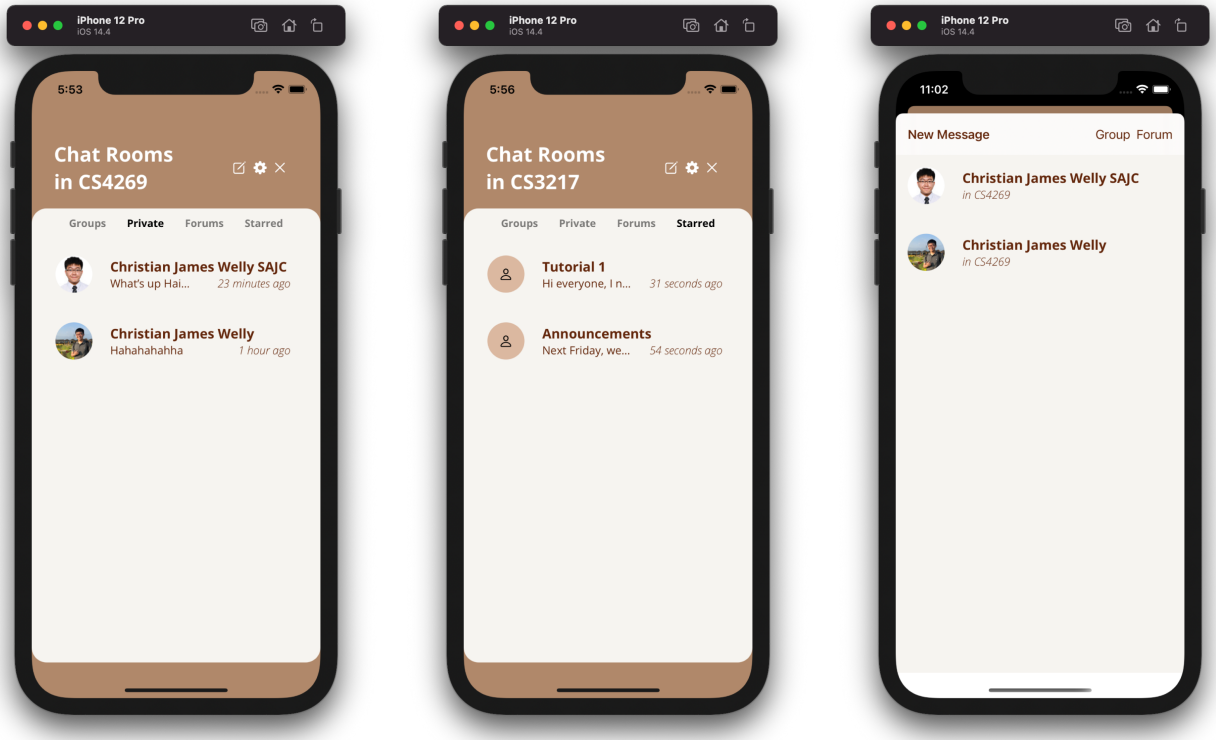
Users can tap on the module item view to enter the module, or tap on the gear icon to open settings.

Settings

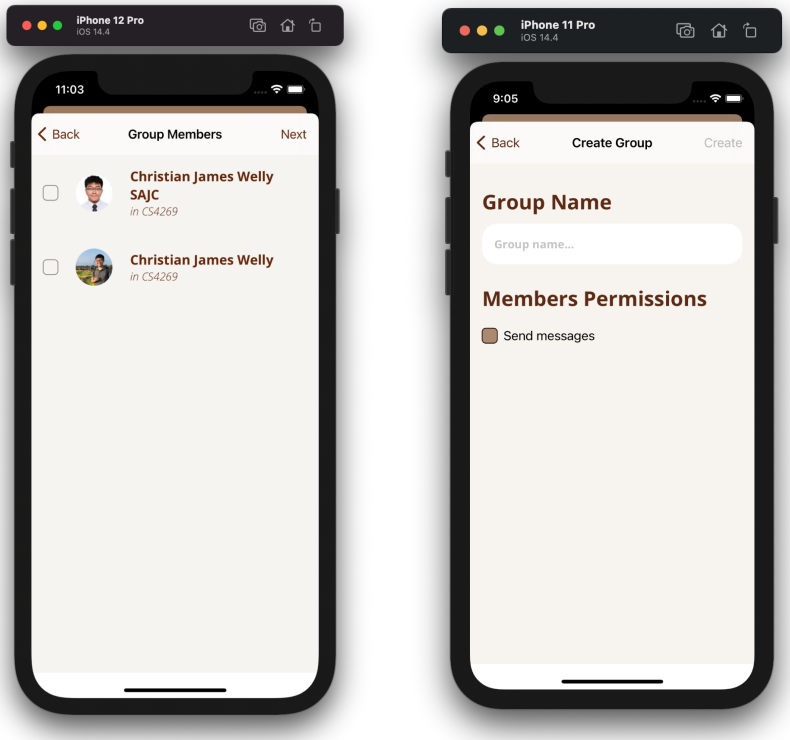
Users can tap Logout to log out of the application.

Modules

When the module page is opened, the user will see the chat rooms linked to that module. Users can choose between the type of chat rooms to categorise the types of chat rooms. Moreover, users can create new chat rooms using the paper and pen icon.



If the user chooses Group or Forum, the user will be prompted to add other members then chat room name. If the user chooses Group, the user would also be prompted to set the permission for other users.



Users can tap on the gear to see the module secret to share with other users so that they can join.

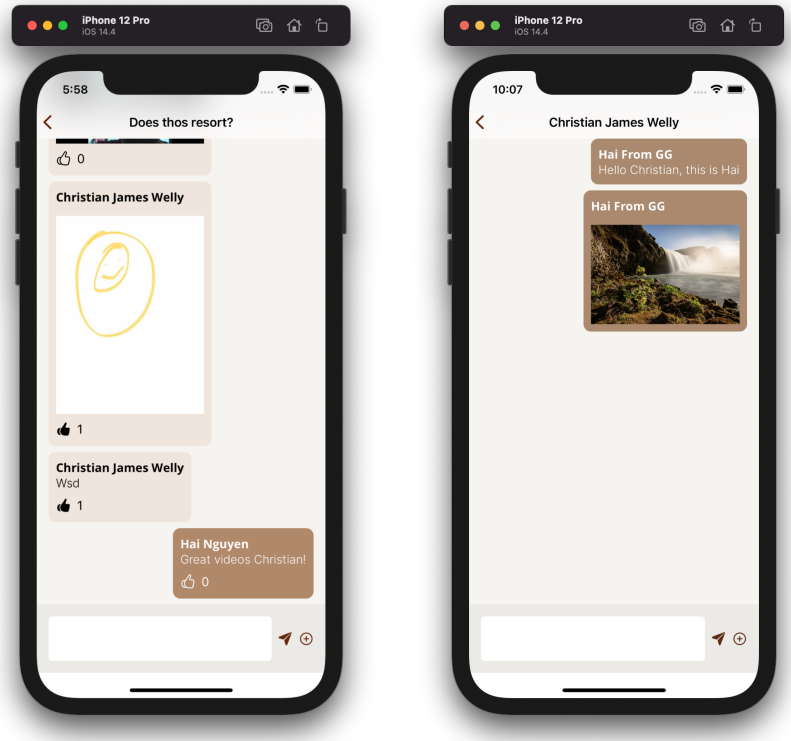
Users can tap on the cross to go back to their module list.

When the user taps on a chat room, the user will be directed to the chat room page, be it a group chat, forum, or private chats.

Chat Room

- ### Private Chat Room and Group Chat Room

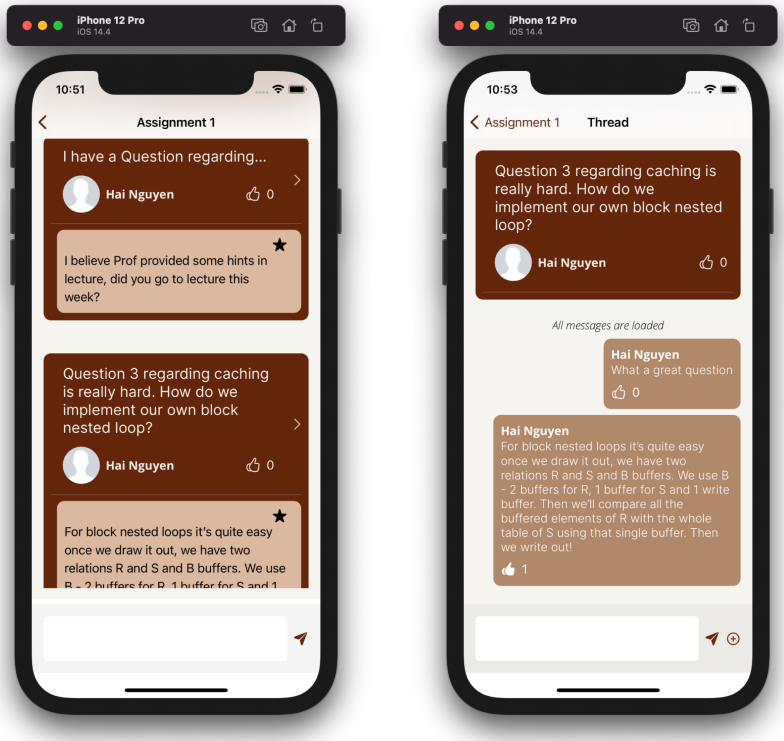
  Inside private and group chat rooms, users can send messages to and read messages from other users in that chat room.



  Note that if the owner of the group chat does not give other users send permission, other users in that chat room will not be able to chat. This is intended for instructors to create announcements channels.

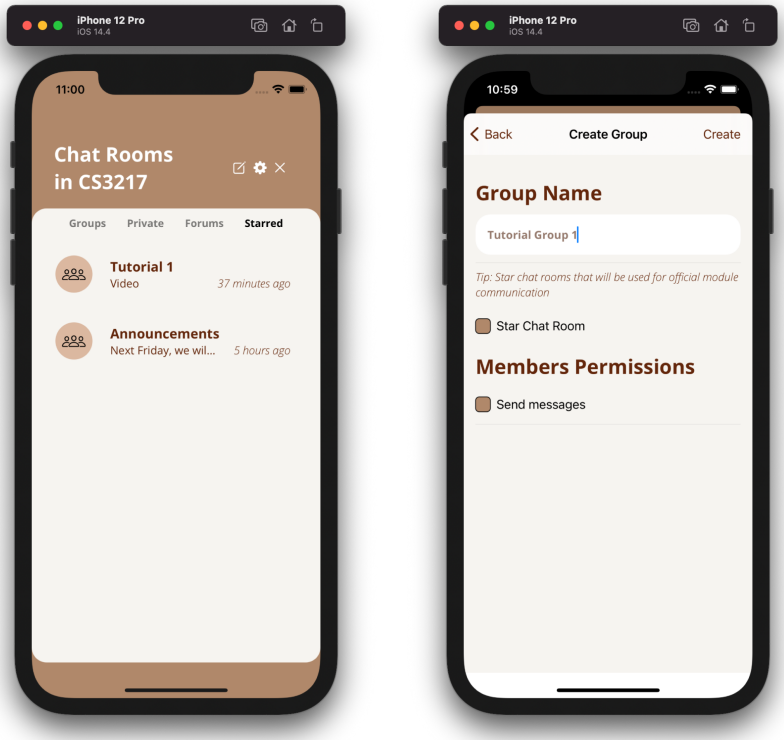
- ## Forum Chat Room and Thread

  Users in the forum chat room can create posts. Then users can double tap on the post to open up a thread and reply to that post. From the forum view, the user can also see the most popular message in the thread to quickly have a reference to the answer.



- ## Star Chat Room

  Module instructors can star Chat rooms and forums at their creation to specify channels for official modes of communication.



Messages

- ## Type of Messages
    - Users can send text messages by typing into the text field and pressing the paper airplane.
    - Users can send images and videos by tapping the plus button, choose “Image and Video” and then choose the video/image they want to send.
    - Users can send a canvas drawing by tapping the plus button, choose canvas, draw then press the send button at the top right corner.

- ## Replying of messages

  Inside group chat, private chat, and thread chat rooms, users can long press on a message to open a context menu click to reply to them.

- ## Editing and Deletion of Messages

  Inside group chat, private chat, and thread chat rooms, users can long press on a message to open the context menu and click to edit and delete messages.

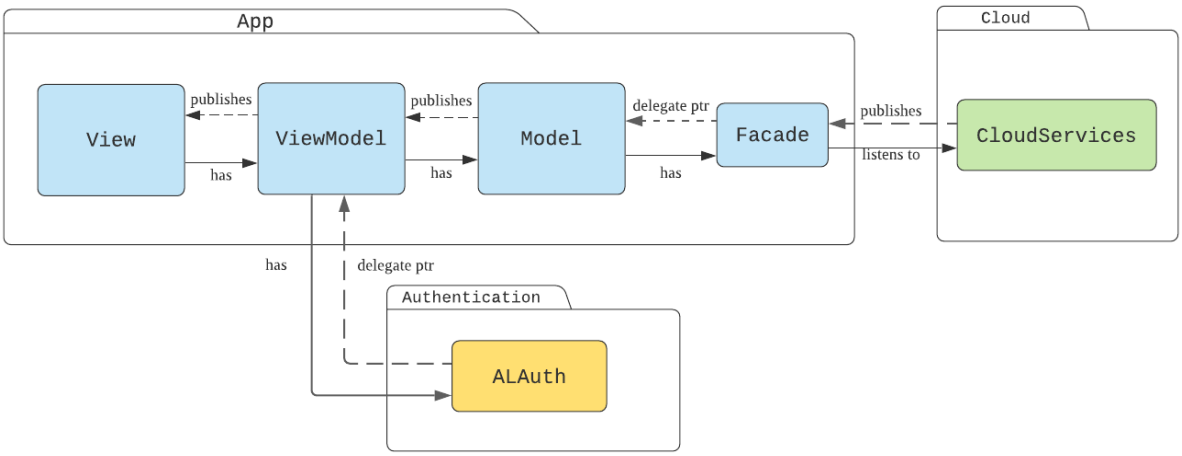
- **Liking of Messages**

  Messages, including forum posts, can be ‘liked’ and ‘unliked’ by tapping on the ‘thumbs up’ button. This is useful for forum thread messages. Messages that have the most number of likes will be previewed in the forum posts.

Code Design

Top Level Organisation: MVVM

We chose MVVM for our high level architecture.



The main benefit that MVVM brings to our architecture is the layer of abstraction between View and Model. This helps us debloat the View by a large amount as translation between from Model objects to View related objects is abstracted to the ViewModel. The View also does not need to know about the Model as it now only needs to pass the user event recorded in the View to the ViewModel. Due to this layer of abstraction, we fully decoupled the View and Model.

On top of this, we used Facade to decouple the dependency between the Model and Cloud services (more on this will be discussed later), and extracted ALAuth (Authentication Library Service) to its own module.

Event Handling

When a user opens a chat room, the user will see an input text box at the bottom of the screen. When the user types in a message, the *message content* state of the View changes accordingly. When the user click on the ‘Send’ button,

- the ViewModel will handle the sent *message content*.

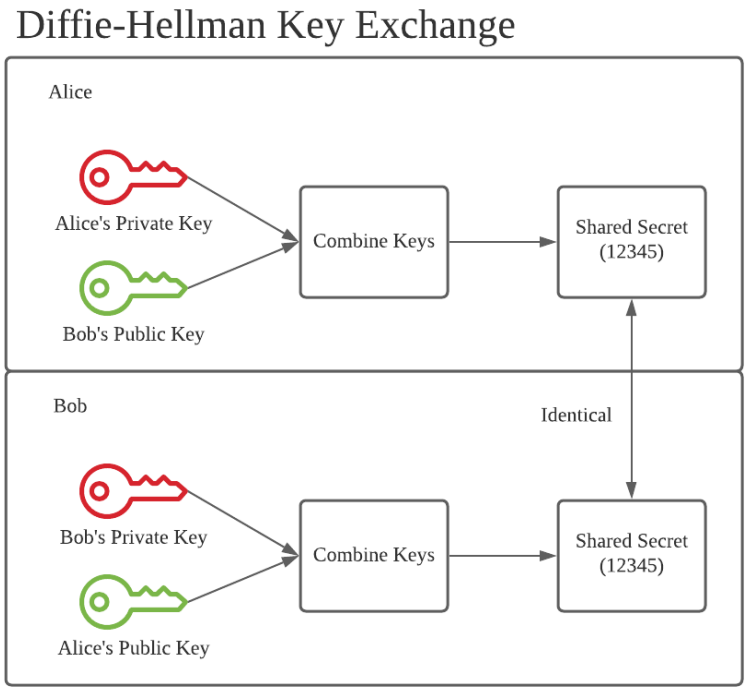
- The ViewModel will call the chat room model's method to save this message.
- The method will save the message in the Cloud database.
- The method calls the facade method that handles the adding of this entry to the database, in this case Firebase.
- A new entry will be added to the database, and this change is listened to by the facade.
- The facade will then call the respective method of the chat room Model, as chat room is the delegate of the facade.

Cryptography

Background: Public-Key Cryptography

For brevity, we assume that Alice and Bob want to communicate with each other in the following scenarios.

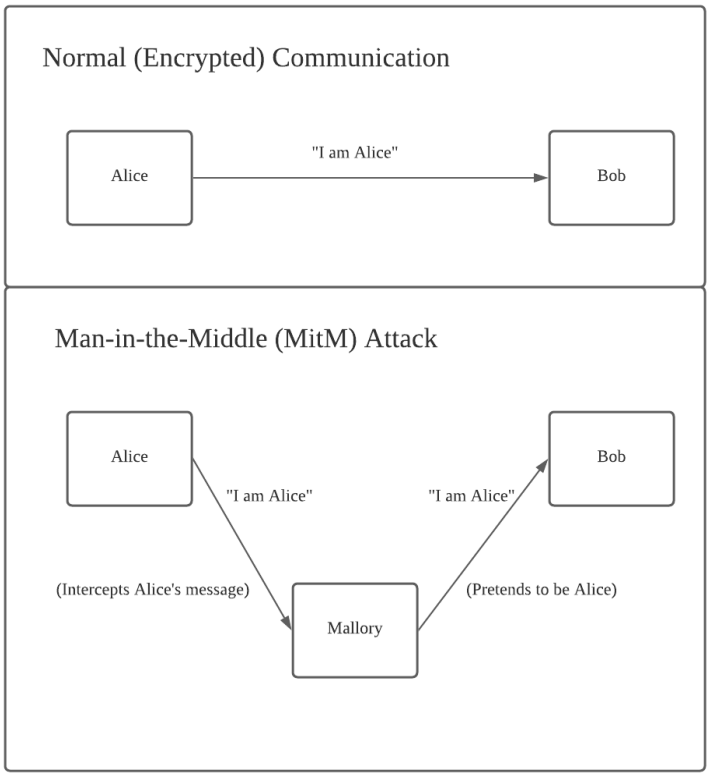
Public-key cryptography uses pairs of public and private keys to encrypt and decrypt messages. Everyone's public key is known in advance (e.g. uploaded to a server that everyone has access to). Diffie-Hellman key exchange is one way of performing public-key cryptography.



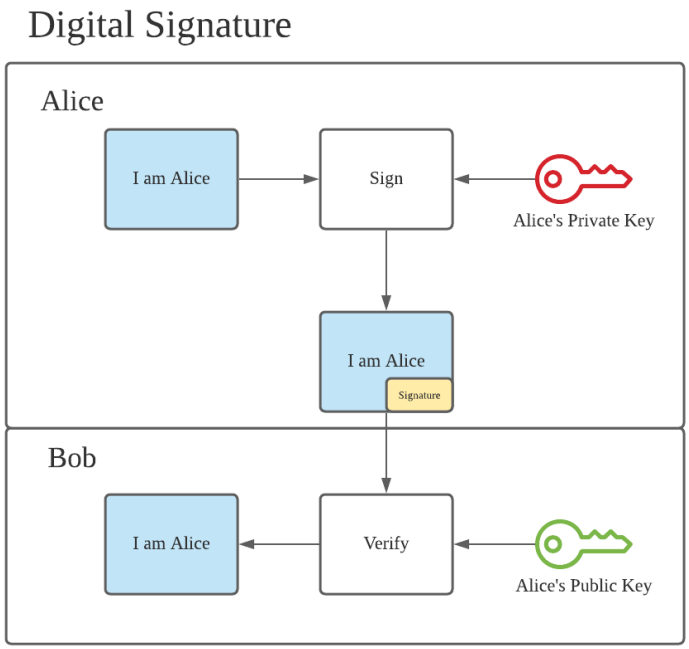
Essentially, Alice and Bob are able to derive the same shared secret from their own private key and the other person's public key. They can then use this shared secret to independently create the same master key to encrypt and decrypt communications between themselves.

Background: Authenticity

The above procedure ensures that message content is not compromised. However, users cannot be sure that the other party is really who they say they are.



Consider the above MitM scenario. Mallory intercepts Alice's message and sends a message to Bob claiming that she is Alice. Bob will perform the Diffie-Hellman key exchange with Mallory and communicate with her as though he was communicating with Alice. The problem is that Bob has no way of verifying whether the (encrypted) message is really coming from Alice. One way of verifying a sender's identity is through a digital signature.



With the use of a digital signature, Bob will be able to verify that the message really came from Alice (i.e. the authenticity of the message is preserved).

We get authenticated key-exchange by combining encryption (Diffie-Hellman key exchange) and authentication (digital signature). This is a key aspect of many modern cryptography schemes.

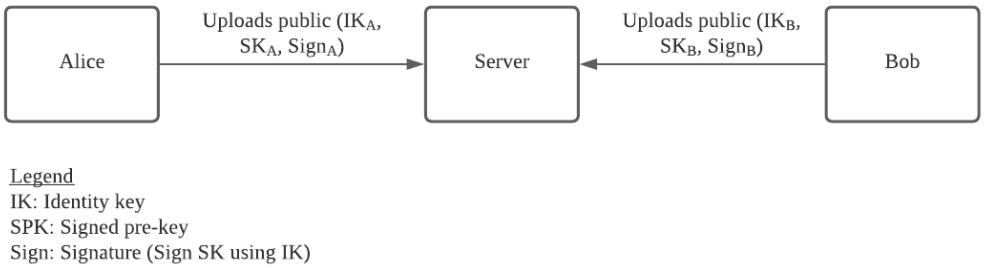
Signal Protocol

The application makes use of the Signal Protocol to encrypt all communications between users. We decided to adopt it as it is used by many popular messaging apps such as WhatsApp and Signal. All messages are encrypted client-side before being sent to the server.

Pairwise Secure Channels

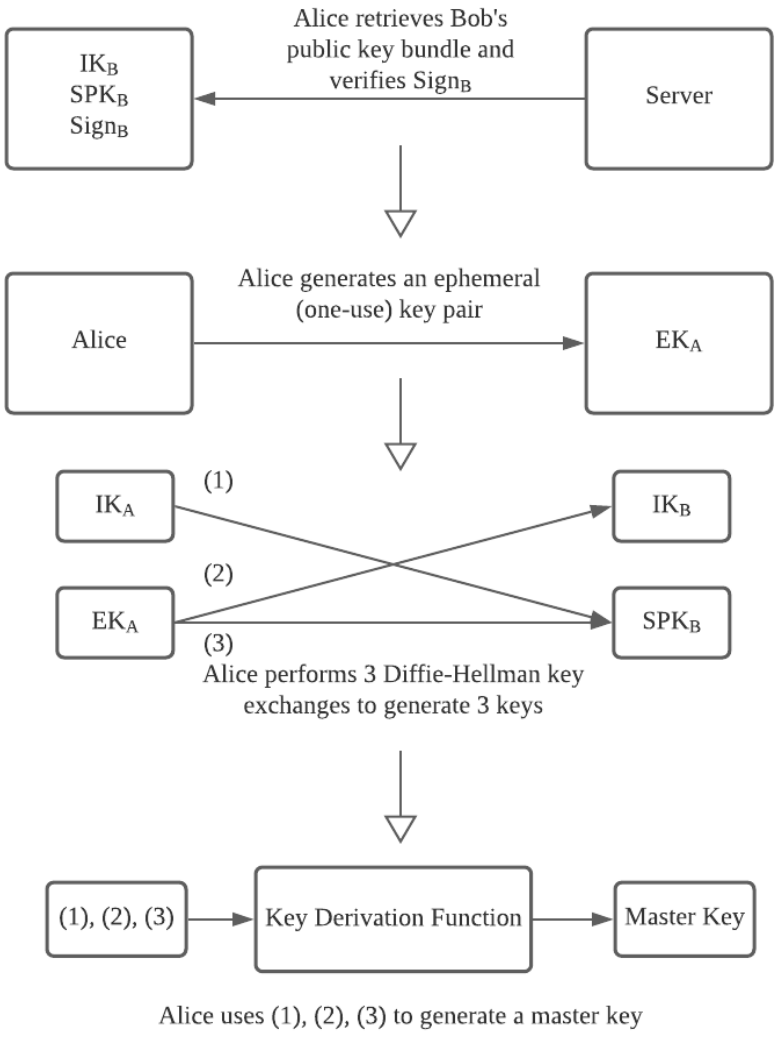
The crux of the Signal Protocol is the establishment of a secure channel between pairs of users. The following are the steps involved in setting up such a channel:

1. Every user who logs in for the first time will upload his/her public key bundle to the server. Every public key has a corresponding private key that only the user knows. All of the following key exchange operations involve the user's private keys and the other user's public keys.



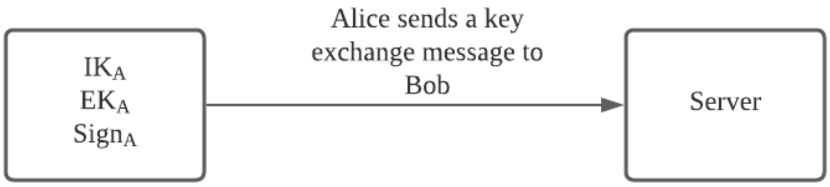
Legend
IK: Identity key
SPK: Signed pre-key
Sign: Signature (Sign SK using IK)

2. Alice wants to initiate a conversation with Bob. Alice retrieves Bob's public key bundle from the server and uses it to generate the master key via the Extended Triple Diffie-Hellman (X3DH) key agreement protocol.



Alice uses (1), (2), (3) to generate a master key

3. Alice sends a signed message to Bob containing her public identity and ephemeral keys for Bob to generate his copy of the master key.

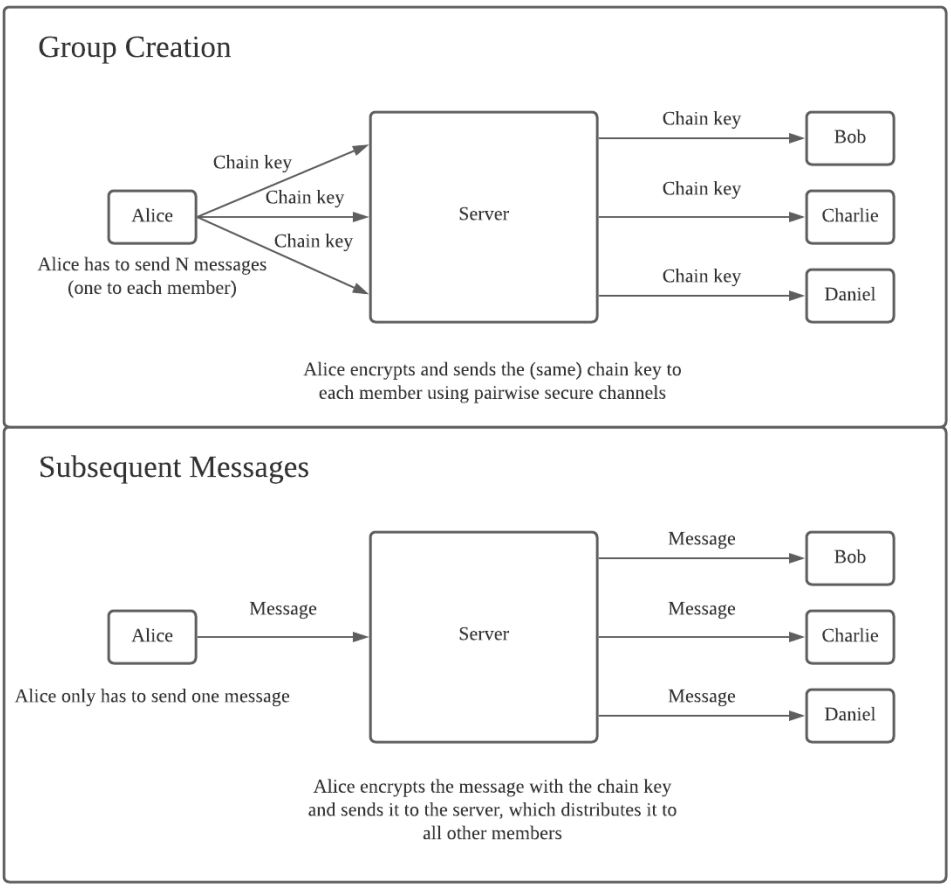


4. Bob verifies Alice's signature and uses her public identity and ephemeral keys to generate the same master key using similar steps as the X3DH algorithm from step 2 (i.e. using his private identity key IK_B and private signed pre-key SPK_B, and Alice's public identity key IK_A and public ephemeral key EK_A).

Group Messaging

The Signal Protocol was initially designed for pairwise communications. Hence, we adapted it according to the steps outlined here to apply it to group chats. For this example, we assume that Alice wants to create a group chat with Bob, Charlie, and Daniel. The key idea behind the adapted implementation is as follows:

1. Alice establishes pairwise secure channels with Bob, Charlie, and Daniel by performing a key exchange with each of them.
2. Alice creates a random key (known as the chain key) which will be used to encrypt and decrypt all future group messages.
3. Alice distributes the chain key to all group members using the pairwise secure channels.
4. All future communications are encrypted and decrypted using the chain key.



Trade Offs

This way of implementing the protocol is an optimisation over another variant, which does not involve sending out a chain key to all members. Instead, sending messages uses pairwise channels between all members. This means that every time someone wants to send a message, he/she has to send it individually to all other members using their pairwise secure channels. The tradeoffs are summarised as follows:

Tradeoff / Implementation	**Chain Key (Current Approach)**	**Pairwise Group Messages**
Number of messages when creating group	O(N)	$O(N^2)$
Number of actual messages each time a member wants to send out a message	O(1)	O(N)
Forward secrecy	Possible	Possible

Post compromise security ("self-healing" property)	Not possible because of the group setting	Possible

Forward secrecy means that an adversary is unable to read past messages if he/she were to obtain the message key. Hash ratcheting is the most common technique to achieve this. A one-way hash function is used to generate (hash) a new key every time a message is sent. Users keep the number of passes through the hash function in sync by incrementing it upon send/receipt.

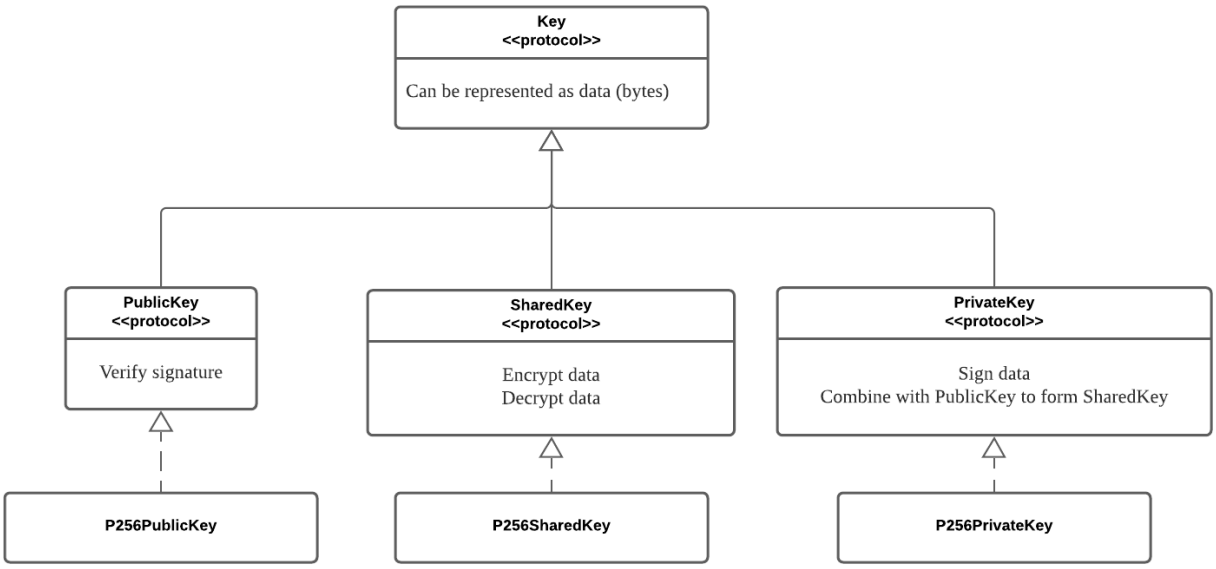
Post-compromise security means that an adversary is unable to read future messages if he/she were to obtain the message key. The most common technique to achieve this is called the double ratchet algorithm. This is only possible to do in the pairwise context and not with our current approach.

We ultimately decided to go with the chain key approach in anticipation of a potentially high number of users when considering our target audience. The app's use in an educational setting means that there could be chatgroups with dozens or even hundreds of users. Hence, we value optimising speed a bit more than improving the security of the algorithm, especially since it is extremely difficult to obtain the message key in the first place.

Interestingly enough, we did find a research paper here which discusses the use of Asynchronous Ratcheting Trees (ART). The use of these data structures allows for post-compromise security to be achieved in the group setting while still having the desired speed advantages and forward secrecy. However, we decided that it would take too long to implement the algorithm while ensuring a minimum level of correctness. In contrast, the Signal Protocol is very well-known and it was easier for us to find related resources when implementing it ourselves.

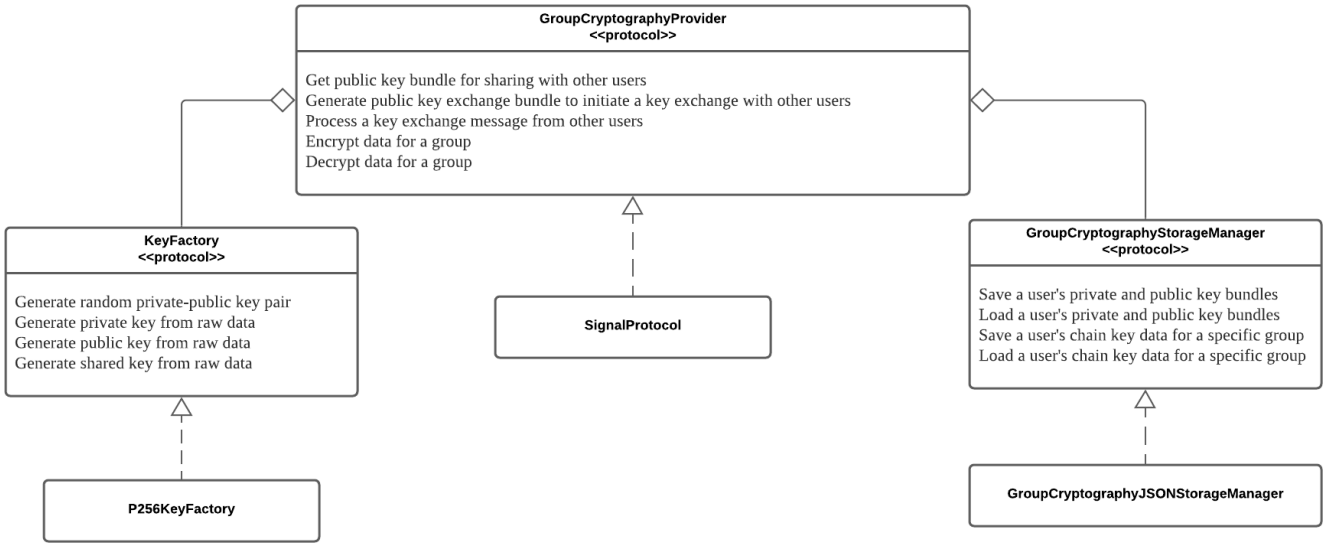
Implementation Details

This section will go over some implementation details from low to high levels of abstraction.



Our keys are represented with protocols to allow for different implementations. Public keys can verify signatures. Private keys can sign data and combine with a public key to form a shared (symmetric) key. Shared keys are used to encrypt and decrypt data.

Our implementation uses NIST Curve P-256 for elliptic curve cryptography. We used the Diffie-Hellman key-exchange algorithm to generate shared keys and the elliptic curve digital signature algorithm (ECDSA) to generate signatures. The ChaCha20-Poly1305 stream cipher is used for encryption and decryption.



The GroupCryptographyProvider is an abstraction over the cryptography library that exposes methods related to group-based cryptography protocols. As mentioned before, our implementation is based on the

well-known Signal Protocol. It contains a KeyFactory for producing keys and a GroupCryptographyStorageManager for client-side storage. Our concrete implementations provide NIST P-256 backed keys and use JSON encoding for storage, respectively.

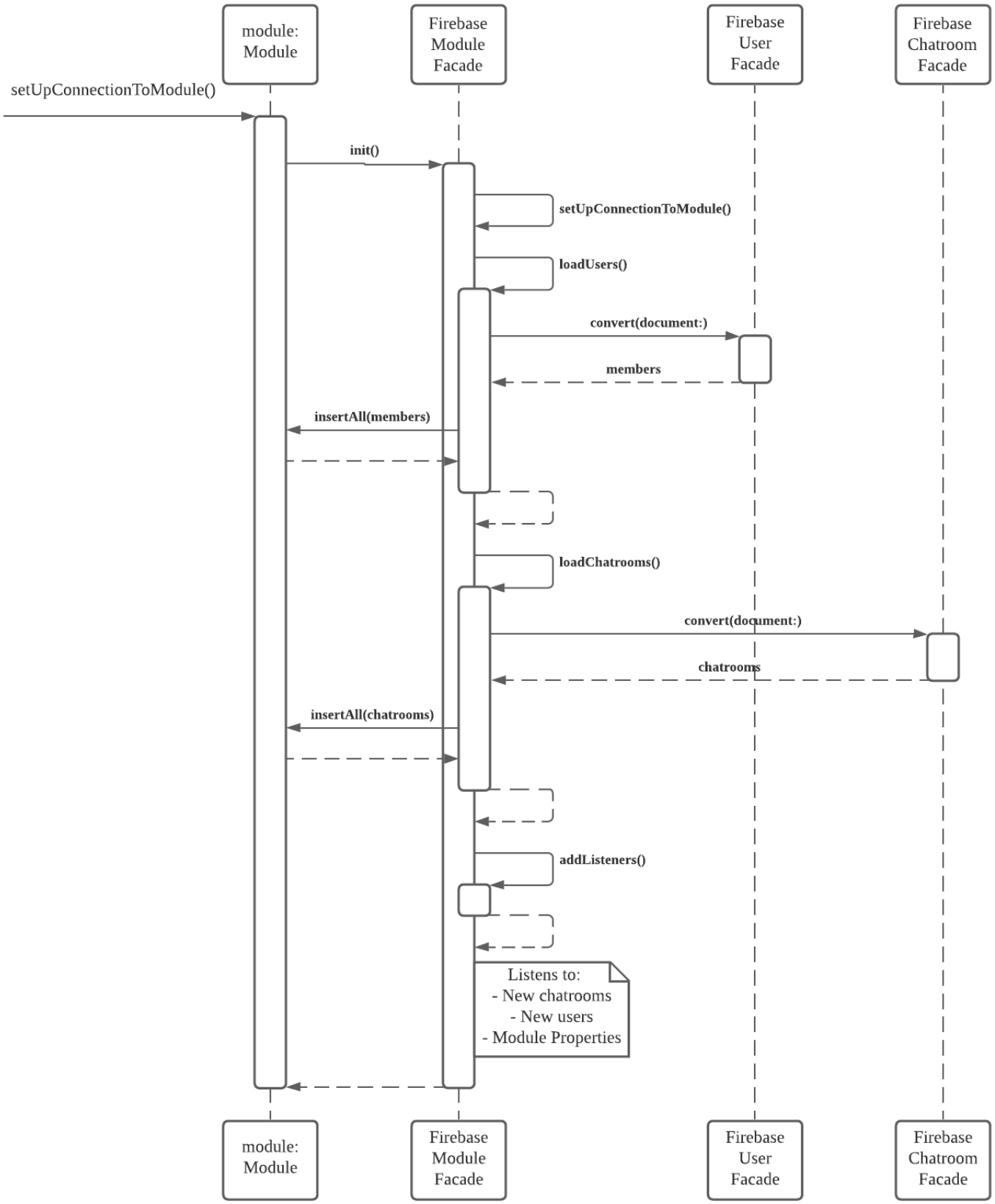
Model-Cloud Sync

Due to the nature of the messaging function, our model has to stay in sync with the cloud service. To ensure that there is no coupling between our choice of cloud service and our model, the connection from our model to the cloud service is done through our Facade. The following shows the design considerations:

1. Facade vs Adapter
   Instead of our current Facade implementation, we also considered having an adapter between the model and our cloud service to reduce the coupling. However, we chose to not use the adapter even though the adapter would solve the problem of mismatch data types between our Model and our Cloud service, it cannot abstract away the setting up of connection between the Model and the Cloud service, especially when this connection will vary based on the cloud service we use. For example Firebase listens to snapshots but AWS might need to open a socket connection. By having a facade, we can abstract the full cloud service from the model and only need to define functions that will be called by the facade in the Model.
2. Communication of Facade and Model
   Our Model has a reference to the corresponding Facade. The Facade is a protocol that has a delegate as a property, and this delegate will be set to be the Model itself. Thus, the Model must also implement the corresponding Delegate protocols. It is clear that the Model can call the methods of the Facade since it maintains a reference to it. The Facade, on the other hand, will have to call the methods of the Model via the delegate, since the Model implements it. An alternative consideration is to pass in closures from the Model to the Facade for the Facade to call. We choose to implement the delegate as it improves the extensibility of the code compared to closures. Suppose we are adding new methods to the delegates, then all classes implementing the delegates must implement the new method before the code can even compile. This is less fragile compared to adding new closures as properties, as the Model is not enforced to set these closures which might be needed by the Facades.
3. Additional layer of abstraction between the Facade implementation and Model
   Knowing that the cloud service can change and there may be a possibility of connecting our application to more than 1 type of cloud service, we abstracted the actual Facade implementation ie. FirebaseChatRoomFacade behind a protocol ChatRoomFacade where the Model only has connection . This is to ensure that our code is open to extension but closed to modifications following Open-Closed Principle.

Understanding how the facade is used in our design for cloud syncing, we discuss how we set up the connection to our cloud services, and how we sync the model to our storage in the cloud.

The following sequence diagram shows how we initiate the connections pertaining to a module:


module: Module
Firebase Module Facade
Firebase User Facade
Firebase Chatroom Facade
setUpConnectionToModule()
init()
setUpConnectionToModule()
loadUsers()
convert(document:)
members
insertAll(members)
loadChatrooms()
convert(document:)
chatrooms
insertAll(chatrooms)
addListeners()
Listens to:
- New chatrooms
- New users
- Module Properties
module: Module
Firebase Module Facade
Firebase User Facade
Firebase Chatroom Facade

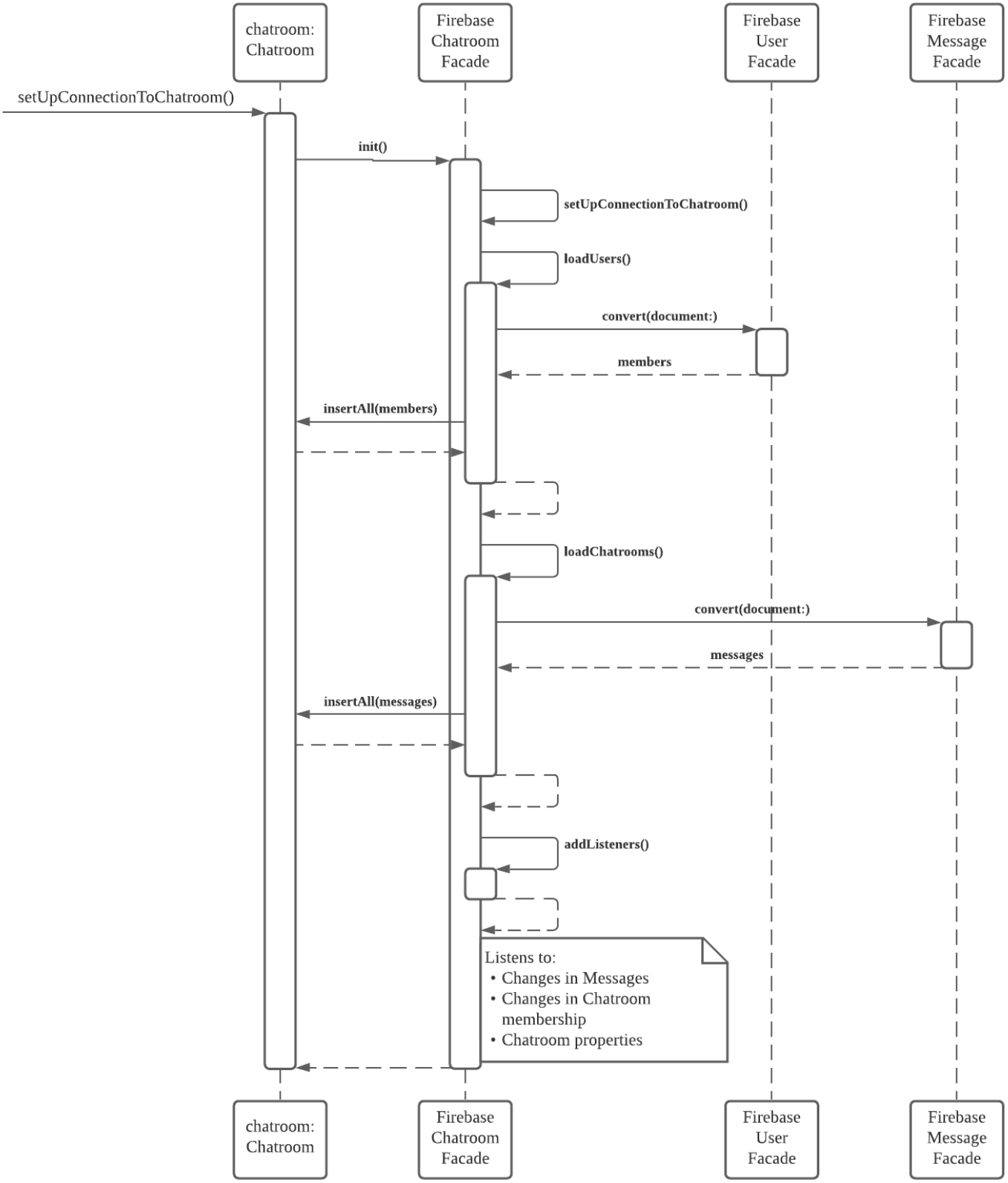
As seen above, we will call the instance method of Module: `setUpConnectionToModule()` to begin the set up. We will also request 2 things from the cloud while doing the set-up: the users who are registered in the module and the chatroom models associated with this module.

Finally, we will call `addListeners()`, which will allow the module to listen to new additions of chat rooms and users (related to the module), and also the module properties such as the module name.

We have chosen to load the chatrooms early in the module in order to allow the following:

1. When a user opens the module, the list of chat rooms has been loaded. This allows a user flow of opening a module, closing it, opening the a different one, and repeat without compromising the responsiveness of the application
2. Having all our modules listen to new additions of chat rooms and users will open the possibility of having in-app notifications while not having the module on screen.

We also have a similar sequence diagram that sets up the connection of the Chatroom:


chatroom: Chatroom
Firebase Chatroom Facade
Firebase User Facade
Firebase Message Facade
setUpConnectionToChatroom()
init()
setUpConnectionToChatroom()
loadUsers()
convert(document:)
members
insertAll(members)
loadChatrooms()
convert(document:)
messages
insertAll(messages)
addListeners()
Listens to:
• Changes in Messages
• Changes in Chatroom membership
• Chatroom properties
chatroom: Chatroom
Firebase Chatroom Facade
Firebase User Facade
Firebase Message Facade

This setting up of the connection is called right before the chatroom is inserted into the list of chatrooms of a module. It retrieves from the cloud the list of users that belongs to this chatroom, and the list of messages that is inside the chatroom. At the end of the retrieval of data, the chatroom will add listeners that listens to updates in messages, updates in chatroom membership (whether a new member is invited or someone has left), and other properties of the chatroom such as the name of the group chat.

The connection of the chatroom has been set up as early as when it is inserted to the module to support the following:

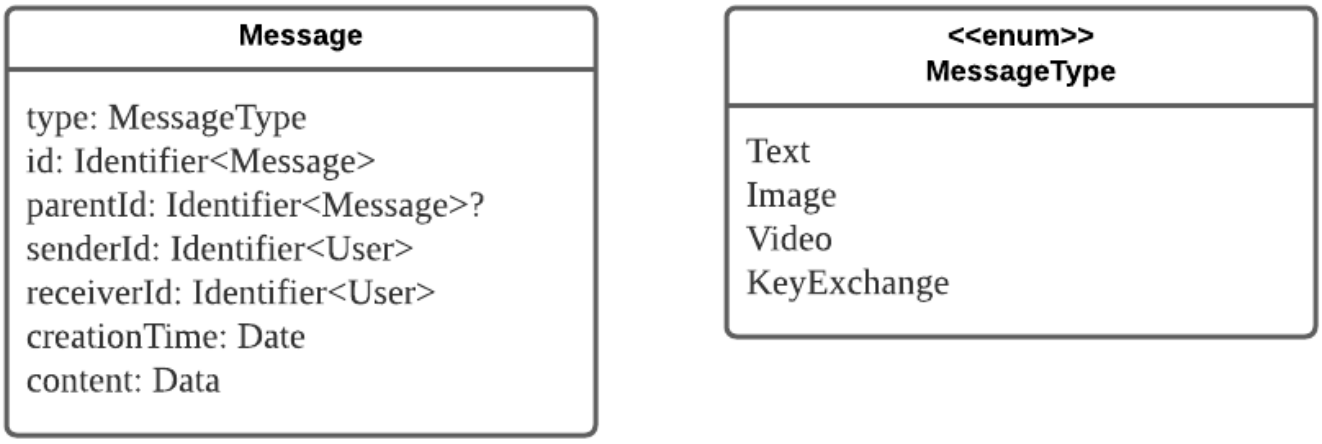
1. In-app notifications of new messages being added, as long as the module connection has been set up.
2. Being able to preview the latest message from the list of chat rooms without even opening any of the chat rooms.

The implementation of the listening is done through the Firebase's API of `addSnapshotListener` which takes in a closure. On changes to a Firebase document snapshot, this closure will be invoked and in the closure we call methods to update our model local state accordingly. This allows our models to be synced with the information in the cloud at all times.

Message

Message Model

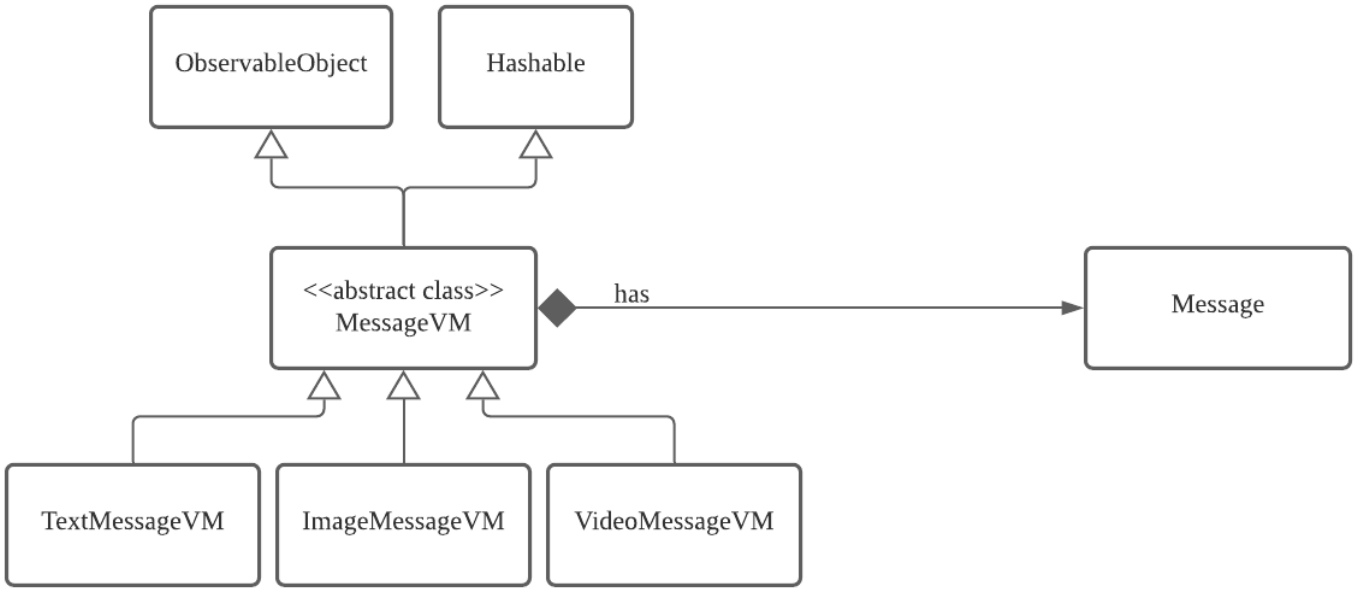
The following diagram shows the Message model and the properties it has:



The Message model is a simple dumb container over any content we want the message to be. The `content` of type Data here can be thought of as a Byte array that represents data. The property `type` will be used to interpret this byte array that can be used by the View to display the desired data appropriately

Message ViewModel

The Message ViewModel is structured with inheritance with the MessageViewModelVM acting as an “abstract class”



This inheritance structure is done as the View Model contains the presentation logic that starts translating information from the Model for the View. As the Message is simply a dumb container wrapping over the content, we choose to use inheritance with different properties for the specific child view models in order to allow flexibility of having more properties.

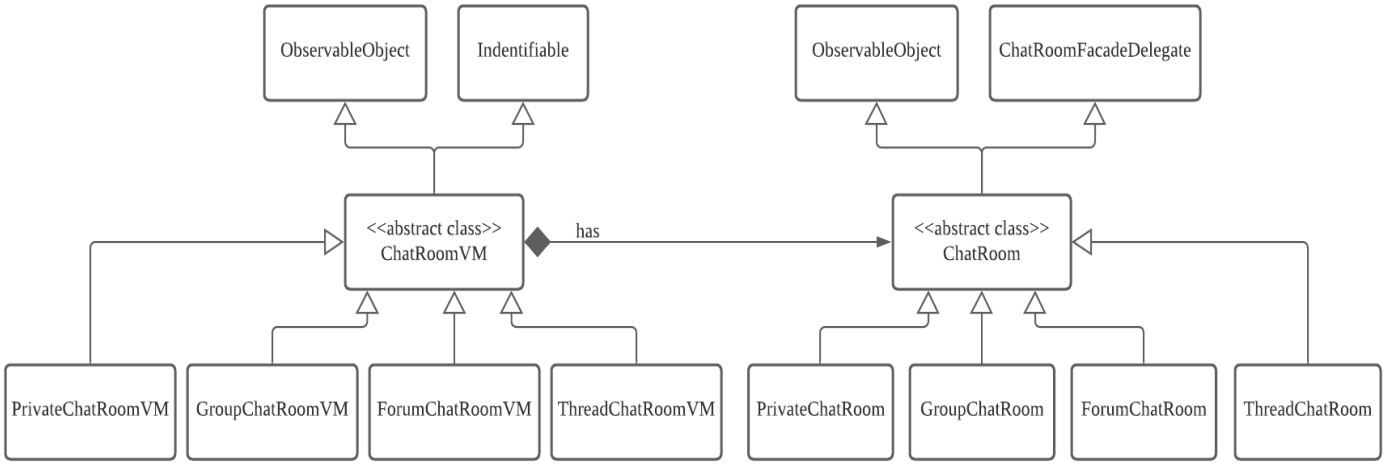
For example, a TextMessageVM might only have the property `text` of type String, and a VideoMessageVM might have the property `url` of type URL which it helps to give to the View for it to fetch a video. VideoMessageVM in the future could also support additional properties such as links to thumbnails, without affecting the logic of other view models.

One might then ask why we choose to keep our Message without inheritance, as it almost seems like the Message can follow the same structure as the View Models. This is because we would like the Message to be as generic as possible and not specific to the application. The properties of the current Message is such that it is very bare bones and can be used for almost any other application which requires messaging. On the other hand, the MessageViewModels are the classes which have the presentation logic specific to the application, so it makes sense to start separating them into their own classes with their own specific functionalities.

Chat Room

In this part we will discuss the considerations that we used to decide on our current implementation of the ChatRoom.

In our implementation, in the Model, ChatRoom is an “abstract class” (note that abstract class is put in quotation marks because Swift does not have abstract classes, thus it’s actually a class that is used like an abstract class) that is implemented by 3 classes PrivateChatRoom, GroupChatRoom, ForumChatRoom, and ThreadChatRoom. Then similarly, in the ViewModel, ChatRoomVM is also an abstract class implemented by GroupChatRoomVM, PrivateChatRoomVM, ForumChatRoomVM, and ThreadChatRoomVM.



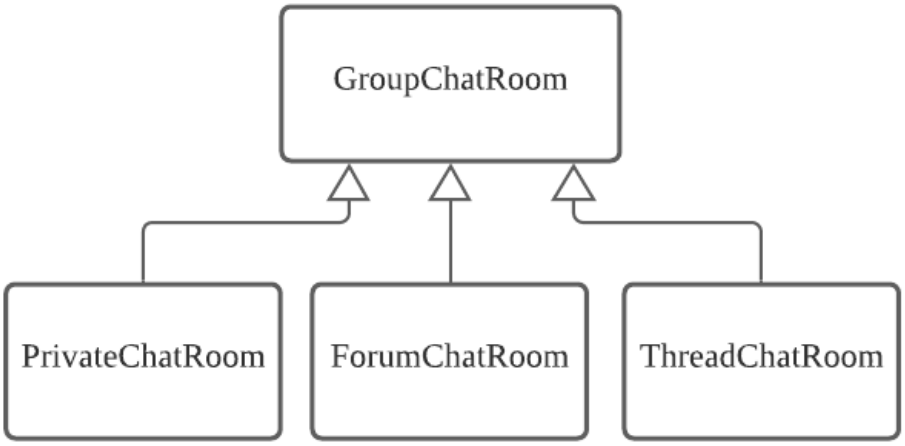
Then, to create corresponding ChatRoomVMs from the ChatRoom type, we used Factory pattern (we will discuss the usage of the factory pattern in its own part). Now we will discuss the alternatives to model different types of chat rooms that our group have considered, and thus conclude why we are using our current implementation.

Chat Room is Model Concept and Not ViewModel Concept

Let’s consider how Message is implemented. In the Model, Message the message type is stored as a variable, the separation of the different types of message is only done in the ViewModel (ImageMessageViewModel, VideoMessageViewModel...). So why can different Message types be a ViewModel concept but ChatRooms cannot? This is because the different types of Message only vary in the content which is all stored in Data which has no other implications on other information stored in the message. On the other hand, type of ChatRoom would determine other variables, such as the name and chat room image (Private Chat Room would be the name and image of the other user) or permissions (members in Private Chat Room and Forum Chat Room must have write permission). This hinted to our group that different Chat Rooms have different

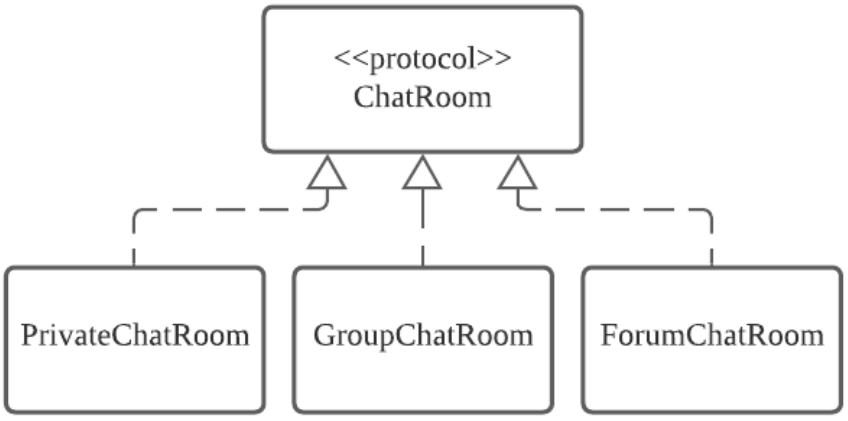
variables and representation invariants that they must ensure. This led our group to decide that the different types of ChatRoom must be represented in the Model.

Abstract Class and Not Class Inheritance or Protocol



Alternative representation with inheritance from GroupChatRoom

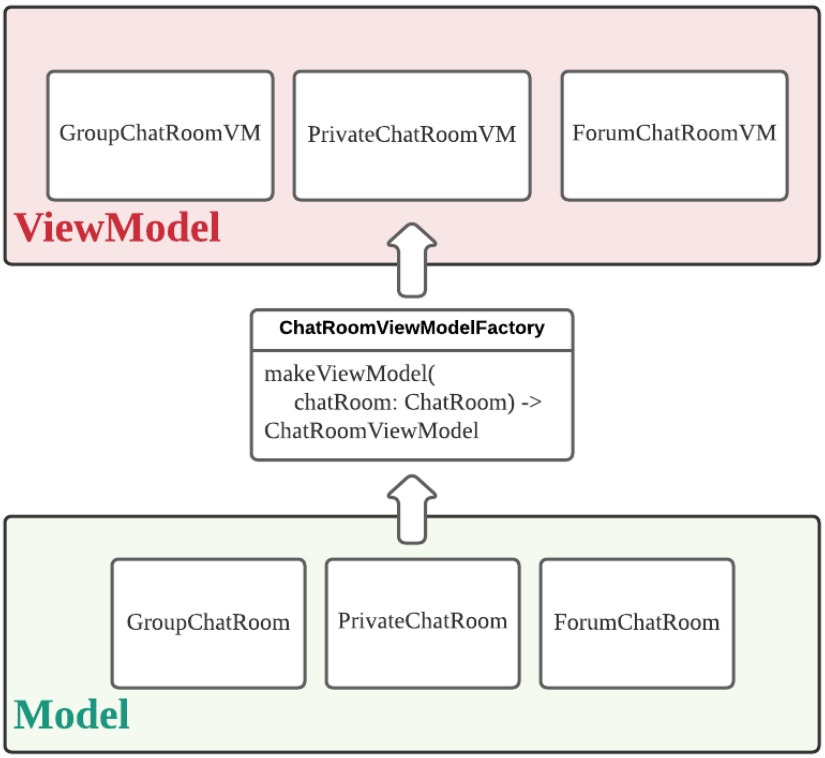
This structure was the first structure that we considered. Despite being simple, it breaks Liskov's Substitutability Principle as the sub classes are more restrictive than the parent as PrivateChatRoom and ForumChatRoom determines other variables of the GroupChatRoom such as the name, permissions, picture and other representation invariants.



Alternative representation using Protocol

We also considered having ChatRoom as a protocol, on a higher level, this design is identical to our current design. The reason we avoided Protocol for ChatRoom is simple because protocols in swift are extremely restricting. To ensure DRY, we must ensure that ChatRooms must inherit from ObservableObject and ChatRoomFacadeDelegate and have wrapper properties which is not possible with protocols., thus it must be a class and not protocol.

Factory Pattern Translation between Architectural Components



ChatRoomViewModelFactory takes in a ChatRoom (of 1 of the 3 types) and returns the corresponding ChatRoomViewModel

Above is an example of how we use this architectural pattern to translate from lower architectural components of higher architectural components. This choice was made in place of the Visitor pattern.

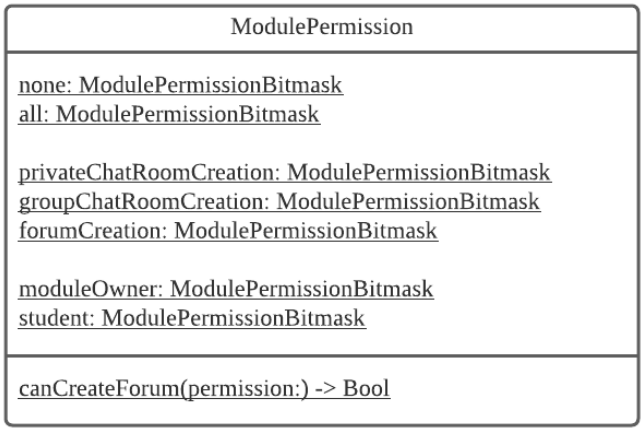
In the Visitor pattern, the Visitor will return the data of the type we desire. To do so, the Visitor should implement a `visit` method that takes in the specific instance of the classes that implements the Visitable, and the Visitable will implement an `accept` that takes in a Visitor.

The Visitable is what we are translating from and in our case that would be the low-level component. Since it has to accept the Visitor, which is the high-level component, it implies that low-level component will have to depend on the high-level component, which is undesirable in the MVVM architecture.

With this choice of using a factory, the most notable trade off is that we need to ensure that our switch cases are exhaustive when switching on class types (as there will be no compile time error). However, that is a cheap price to pay as compared to the Visitor pattern as it would expose our higher level component which is the Model to ViewModel.

Module Permission

The module permissions are implemented as a bitmask on what behaviours are allowed in the module as represented in this struct:



The `moduleOwner` and `student` permission is a logical conjunction of the individual bitmasks, creating a combination of what the module owner and student can do in the module.

The ViewModel will be calling the methods in ModulePermission, to check whether the current user in the module has the permission to perform certain operations, before passing that information down to the View.

Bitmasks implementation over Role enum

An alternative we considered is to use a role enum. Instead of using ModulePermissionBitmask, define a Role enum, whose cases can be either `moduleOwner` or `student`. The method that checks whether a particular Role can perform a certain operation will then be defined in terms of a switch statement that returns a boolean literal depending on the different cases.

The disadvantage of using this Role enum is that once we have many operations, and we want to add a new role, every function has to be changed to accommodate this new Role enum. For example, if we have `canCreateForum`, `canCreatePrivateCR`, `canCreateGroupCR`, and add new role `Instructor`, all the three functions have to be changed in order to support the new role.

On the other hand, with bitmasks, one will only need to define the role `Instructor` as a logical conjunction of the individual permission bitmasks.

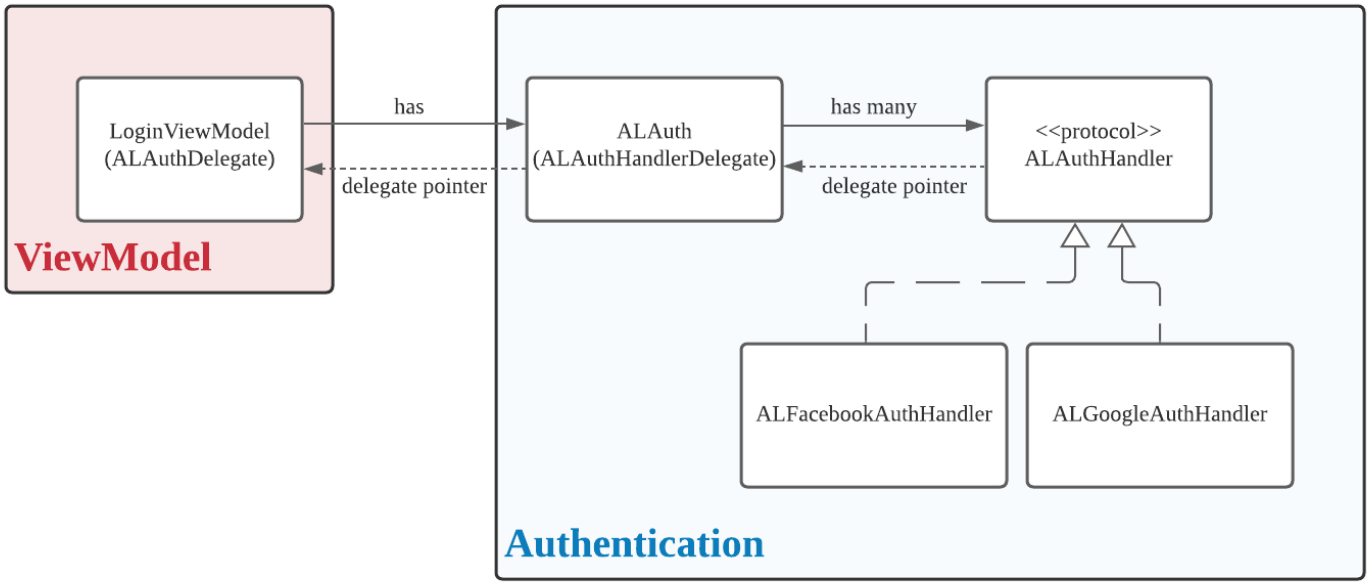
Another advantage of using the logical conjunction approach is that the permissions are localised -- one can clearly see what a particular role is allowed to do in a single line (or a few more). With a Role enum, the information of what the role can do is contained in the functions, and one will have to go through many lines of function in order to determine all the permissions of that role.

Thus, we have chosen the bitmask implementation.

Authentication Module

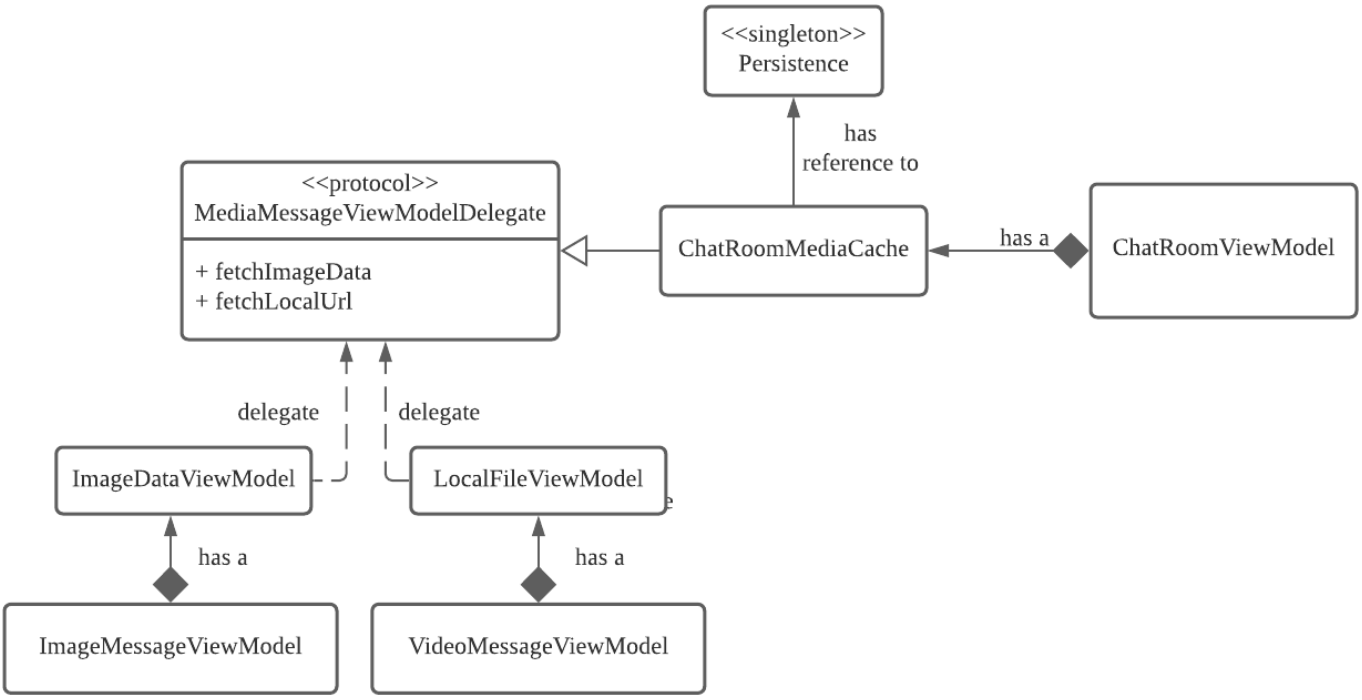
Authentication (Authentication Library ALAuth) is supported by Firebase authentication and is implemented as a separate module from the rest of the SweeChat. It is meant to be reusable in other applications. Currently, it's main dependency is on Firebase authentication to supply other handlers such as Google and Facebook sign in.

We decided to make Authentication it's own module because we saw that it's something that could be entirely decoupled from the rest of the application (Note that logic of storing Users and Authentication are entirely different). Not only does this help us when testing as the modules can be tested separately, thus reducing the possibilities of bugs, it also allows us to work on Authentication in parallel with the development of the rest of the application.



The application connects to the ALAuth via the LoginViewModel. The ALAuth exposes an array of AuthHandlers to the ViewModel, which then in turn exposes the information of the various auth handlers to the view. Moreover, to follow Dependency Inversion Principle, ALAuth does not depend on specific ALAuthHandlers but through an ALAuthHandler protocol, moreover specific ALAuthHandlers do not depend on ALAuth but through a ALAuthHandlerDelegate. This is to ensure that our Auth Library is loosely coupled so that new AuthHandlers can be easily added in the future.

Media Cache



In our current implementation, every ChatRoom holds a ChatRoomMediaCache that will be responsible for getting data for images and local file url for videos. ImageDataViewModel and LocalFileViewModel will access the ChatRoomMediaCache through the protocol MediaMessageViewModelDelegate to reduce dependency.

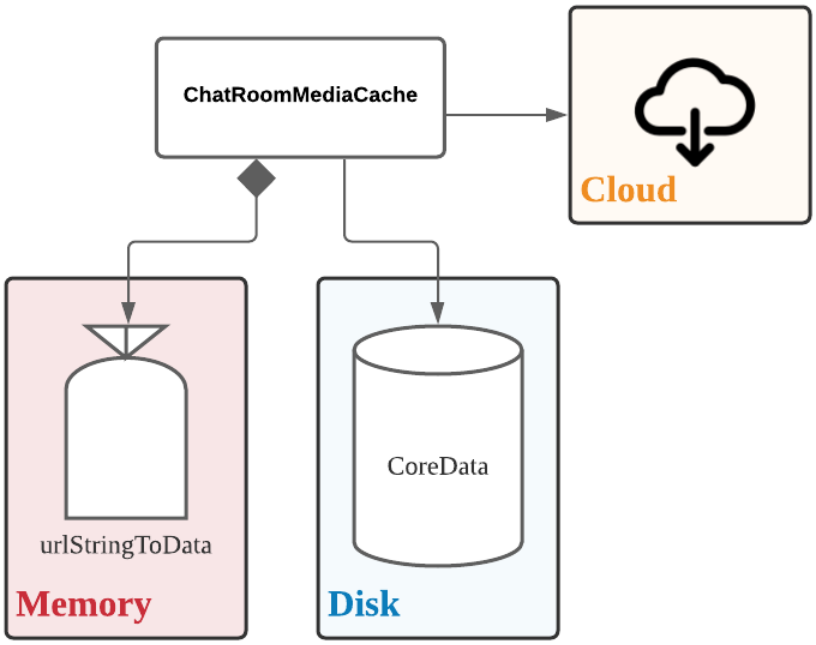
How the cache works

In short, the cache ensures that once media data is downloaded from the web, the application never has to query the web again. This allows users to have offline access to media and cuts down on data usage.

Image Data

ChatRoomMediaCache implements caching of image data using a three level approach. On instantiation, the MediaCache will load small data chunks into the urlStringToData dictionary to hold some of the data in memory. When The ChatRoomMediaCache is queried for data using the Url as key, media cache will:

- look in the dictionary to see if it has been cached in memory, if the data exists in the dictionary it will return
- else ChatRoomMediaCache query CoreData store to see if it has been downloaded. If the data exists in core data, it will return that,
- Else it will use the actual Url to query for the data then store it in disk so that the application will not need to query for the data again.



Video Data

Since Video data is much larger compared to image data, there is no in memory caching for Video data. ChatRoomMediaCache translates the online Url into a local file Url. If the local file Url exists, it will return the local file url, else it will download the data into the local file url before returning the local file url.

Making MediaCache the responsibility ChatRoom

There were a few alternatives to the current design that gives the ChatRoom the responsibility of holding the MediaCache.

Have a application-wide media cache

Instead of having a media cache in every ChatRoom, it was possible to have an application wide media cache. However, since ImageDataViewModel and LocalFileViewModel would need to have reference to this application-wide media cache, this would mean that either:

1. Instantiate the MediaCache in the HomeViewModel and pass it all the way down to MessageViewModel
2. Or, instantiate a MediaCache that is a singleton and allow the entire application to freely access it.

We did not go with option 1 because that would increase the amount of boilerplate code to pass down the MediaCache implementation all the way from HomeViewModel to ImageDataViewModel and LocalFileViewModel, through ModuleViewModel, ChatRoomViewModel. That would also increase the amount of dependency on MediaCache. We also decided against option 2 because despite being much more

convenient, we wanted to ensure that our MediaCache is not accessed in other classes that are not supposed to have access to it.

Allow Classes that need access to the Cache Instantiate their own MediaCache

This would be the simplest of all solutions as given the Url, the class that needs access to the data can just query its own local cache to see if the data has been downloaded before. However, this would disallow the use of early loading from disk. Given n items in a module, this solution would need to query CoreData n times, as compared to our current solution that would only need to query CoreData once. This significant decrease in time complexity made us choose our current implementation instead.

Dependency Inversion Principle

Note that the LocalFileViewModel and ImageDataViewModel do not depend on the ChatRoomViewModel itself, they only depend on the protocol provided by MediaMessageViewModelDelegate. This ensures that even though an ImageDataViewModel and LocalFileViewModel has reference to the ChatRoomViewModel, there is no strong dependency through the use of the MediaMessageViewModelDelegate.

Notification

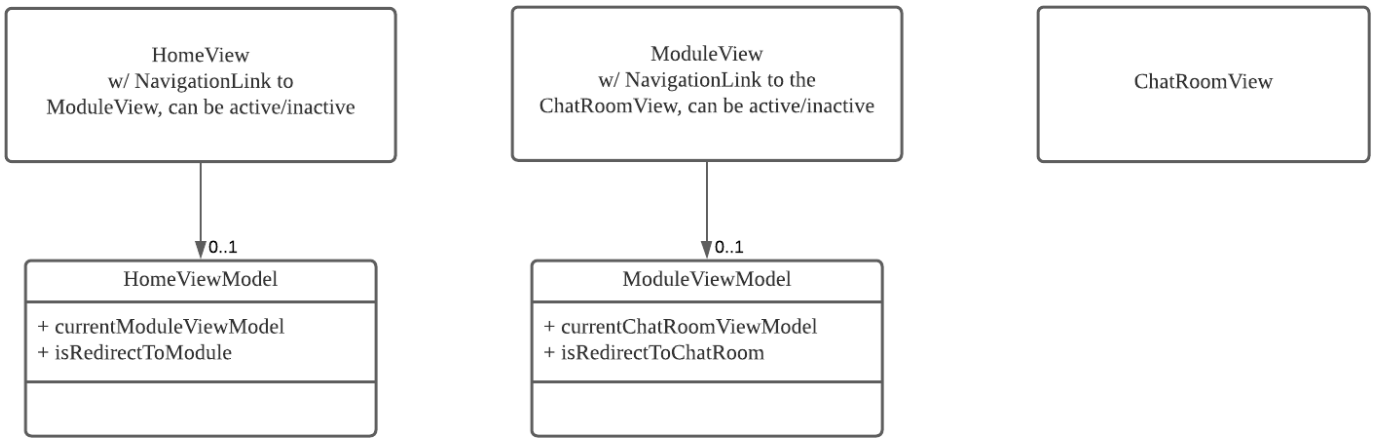
We implemented a notification feature for the users to receive notification when they receive messages from their chat rooms. Upon clicking on a notification, the app will be opened, and the user will be redirected to the chat room where the message came from. We relied on NavigationLink from SwiftUI to handle this redirection.

Automatic and Manual Navigation

In response to an opening of a notification, we have an automatic navigation from any view that the user was from, to the chat room view that the message of the notification originated from. To handle this, we redirect the user from any view the user was from to Home view. Then, from Home view, it will redirect to the module view of and then to the chat room view, from where the notification of the message came from. The redirection to home view is to prevent double stacking of the view on top of each other in a notification response.

In addition to this, manual navigation is still there for the users to navigate between all other views. This means that automatic navigation from notification response needs to cooperate with the existing manual navigation. To handle this,

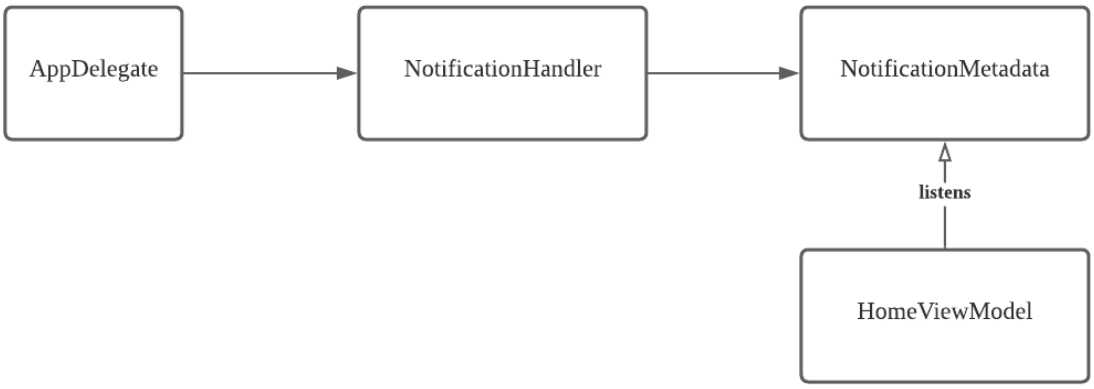
- HomeViewModel
    - Only has at most 1 ModuleViewModel, currentModuleViewModel
        - In a notification response, it will eventually be set to the ModuleViewModel that the message of the notification came from
        - In a normal navigation, it will be set to the module chosen by the user.
    - Has a boolean flag on whether the ModuleViewModel is loaded, isRedirectToModule, indicates whether the HomeView should be redirected to ModuleView
        - In a notification response, it will be set to true when the ModuleViewModel is completely loaded from the Firebase backend
        - In a normal navigation, it will be set to true when the user clicks on the module
- ModuleViewModel
    - Only has at most 1 ChatRoomViewModel, currentChatRoomViewModel
        - In a notification response, it will eventually be set to the chat room that the message of the notification came from
        - In a normal navigation, it will be set to the chat room chosen by the user.
    - Has a boolean flag on whether the ChatRoomViewModel is loaded, isRedirectToChatRoom, indicates whether the HomeView should be redirected to ChatRoomView
        - In a notification response, it will be set to true when the ChatRoomViewModel is completely loaded from the Firebase backend
        - In a normal navigation, it will be set to true when the user clicks on the chat room



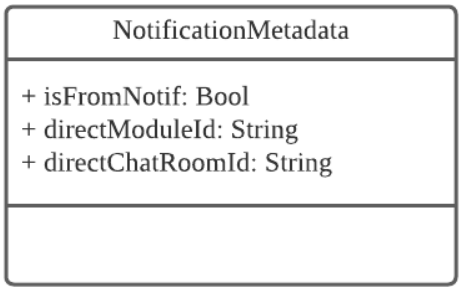
Navigation Flow

There is a notification handler which will handle push notification being clicked by the user. The NotificationHandler will set the property of NotificationMetadata to the ones specified by the push notification payload.

Here is the class diagram of classes related to changes of notification.



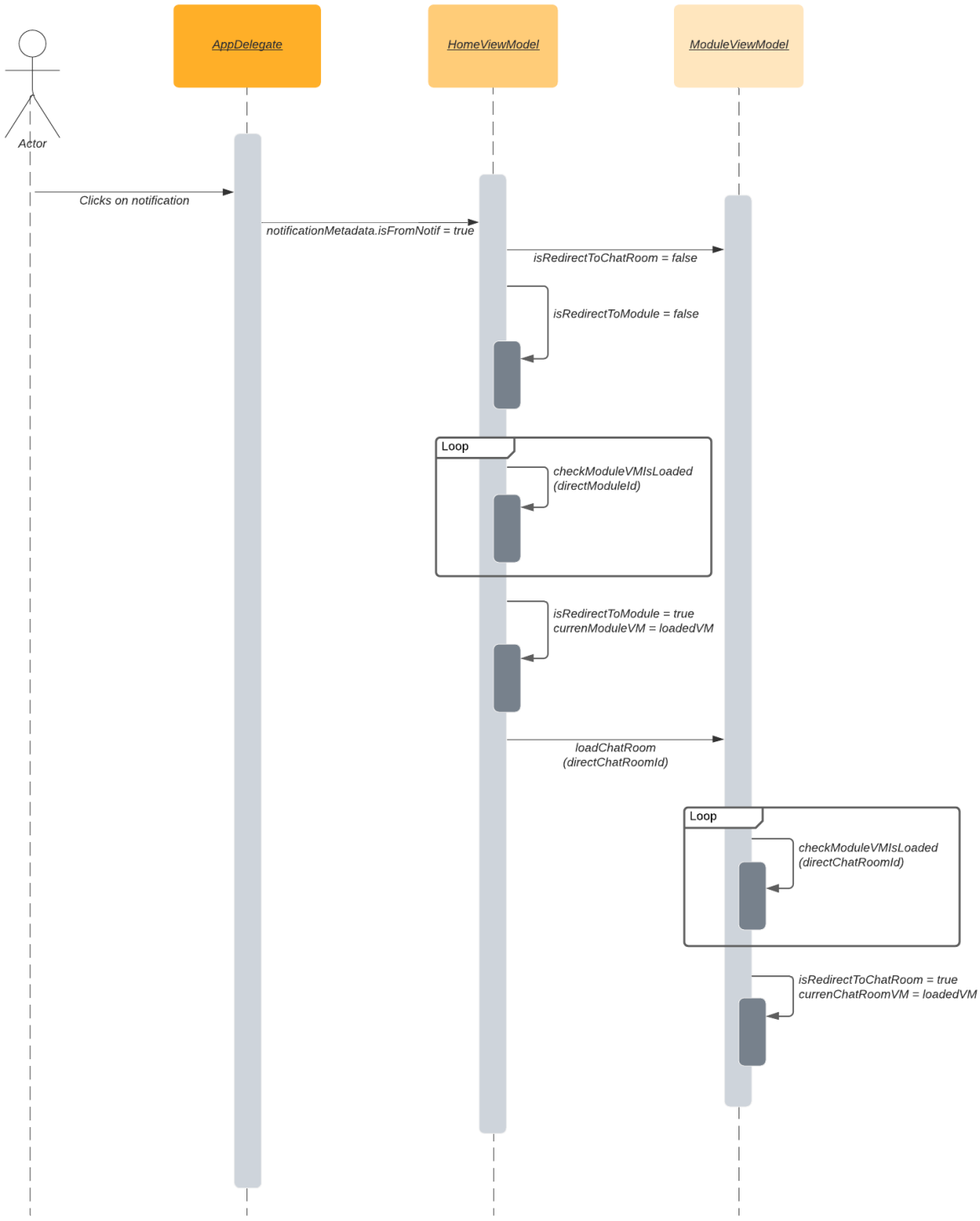
NotificationMetadata stores information related to the payload of the push notification from Firebase Cloud Messaging (FCM).



When the user clicks on the push notification,

- NotificationHandler will set notificationMetadata.isFromNotif = true.
- HomeViewModel listens to the change in NotificationMetadata properties. When notificationMetadata.isFromNotif is changed to true,
    - HomeViewModel will ask ModuleViewModel to set its isRedirectToChatRoom to false, causing any ChatRoomView stack to be closed, if there is any
    - HomeViewModel will set itsRedirectToModule to false, causing any ModuleView stack to be closed, if there is any. This will cause the view to be redirected to HomeView.

- HomeViewModel will wait until the intended ModuleViewModel is loaded from Firebase, set its currentModuleViewModel to it and set isRedirectToModule to true. This will redirect HomeView to ModuleView
- HomeViewModel will ask the ModuleViewModel to wait until the intended ChatRoomViewModel is loaded from Firebase, set its currentChatRoomViewModel to it and set isRedirectToChatRoom to true. This will then redirect ModuleView to ChatRoomView.



When the user navigates normally through the application,

- notificationMetadata.isFromNotif is false, thus there will be no automatic navigation happening.
- If a user clicks on a module, the corresponding properties of HomeViewModel are set as described in the previous section
- If a user clicks on a chat room, the corresponding properties of NoduleViewModel are set as described in the previous section.

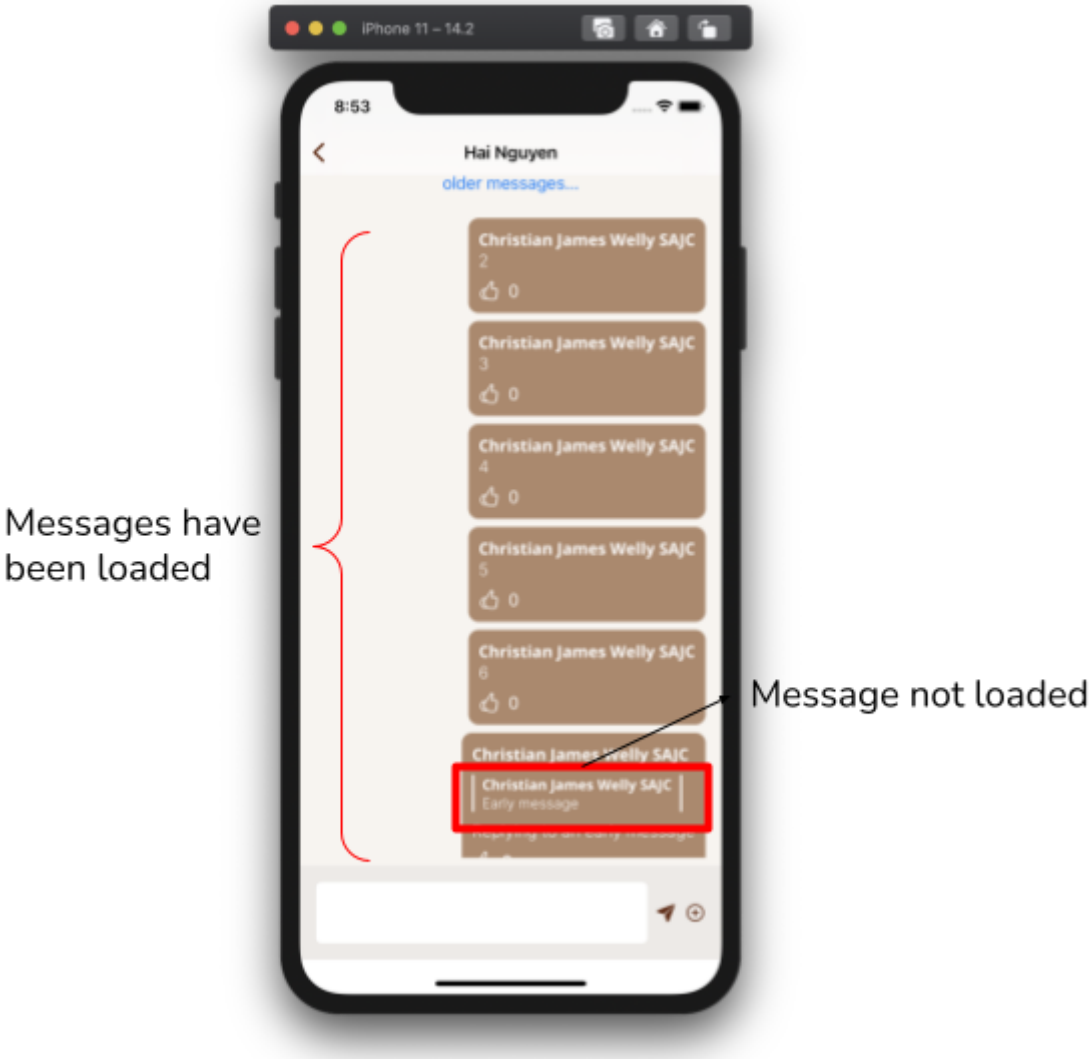
Pagination

As users, we are often interested only in the last few messages in the chatroom. It is thus wasteful to load every single message in the chatroom, even worse if the Chat Room has a huge amount of messages, e.g. 10000. To overcome this, we have implemented pagination that does not request for every message in the database for the queries.

We define a block size, which in our implementation is 10, to be the number of messages requested at once. When we open the chatroom, we will first request only for the last 10 messages, and as we attempt to scroll to earlier messages, we will request for the earlier 10 messages.

Replied Message

A particular problem that can arise is that among the already-loaded messages, there could be messages that replies to a message that has not been loaded (because it was sent very early in the chat history for example). The following is an illustration:



In order to handle this, we maintain 2 data structures of messages. The first one is for the messages which have been loaded, and the other one is what we call “earlyLoadedMessages”. Early Loaded Messages are to maintain the messages that are not in the chatroom, but is one of the messages being replied to among the messages already loaded.

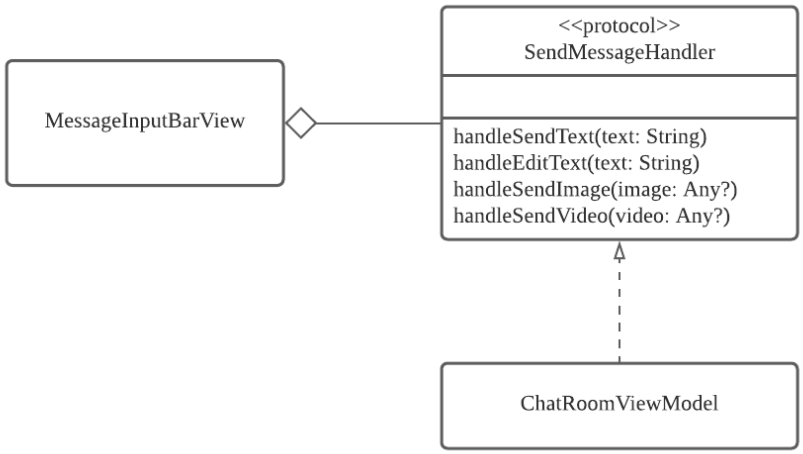
When we load the earlier messages, and we are attempting to load a message that is in our earlyLoadedMessages, we will remove it from earlyLoadedMessages, and insert it into the main data structure.

When we click on the replied message, what we will do is to load all the messages until that particular message has been loaded.

Now an invariant in the ChatRoom that we must hold is that there is no message that is both in earlyLoadedMessages and the normal messages.

Dependency Inversion Principle (View)

Where possible, we have implemented our Views to follow the Dependency Inversion Principle. The following diagram is an example:



Our MessageInputBarView (which includes the TextInput and the send button) depends on the protocol SendMessageHandler. In our current implementation, ChatRoomViewModel conforms to this protocol, and this is because when we are about to send messages, we are asking the ChatRoomViewModel to handle it by creating the necessary messages to be added to the chatroom.

DIP in this case allows our MessageInputBarView to be reused for other components, in the event that we would like to send messages in another context other than chatroom.

Test Plan

Strategy

Our team uses a mixture of white and black-box testing. The former allows us to detect incorrect or missing functionality, while the latter acts as a metric of completeness with respect to our test selection criterion. We decided that a combination of the two would be appropriate for designing a comprehensive test suite for the application.

We chose to use modified condition/decision coverage (MC/DC) adequacy as the coverage criterion for our test cases. We considered other criteria like line coverage, path coverage, and branch coverage. Line coverage aims to ensure that every line of code is tested but is too general as complete line coverage may not imply testing all branches in a program. Path coverage leads to an exponential increase in the number of possible test cases and is unfeasible for anything besides the simplest of software. Branch coverage is more comprehensive than line coverage but could still miss potential edge cases. Hence, we chose MC/DC

adequacy as a compromise between path coverage and branch coverage. It enables us to test important combinations of basic conditions without the exponential blowup in test suite size that comes with path coverage. An important combination means that each basic condition can be shown to independently affect the outcome of each decision. Each basic condition *C* requires two test cases: the values of all evaluated conditions except *C* are the same, and the compound condition as a whole evaluates to true for one and false for the other.

Given the MVVM architecture of our application, our plan is to have a mixture of unit tests, integration tests, and UI tests. We use unit tests to test individual modules within our Model and ViewModel domains. Stubs/mocks are used in conjunction with dependency injection to test software components in isolation. Thereafter, we have integration tests to test interconnected components. These include components at a different architecture layer (e.g. ChatRoomViewModel and the ChatRoom model representation) and those which are at different abstraction levels (e.g. ChatRoom and FirebaseChatRoomFacade). UI tests are used for testing interactions between the View and the rest of the system, i.e. does each user interaction produce the expected on-screen change. The use of stubs implies a bottom-up approach towards testing. We felt that smaller, individual modules should first be tested before testing interactions between them. Contrast this with top-down testing where driver classes have to be written to test high-level components of the system first. We felt more confident about fixing the bugs detected by a bottom-up approach at the lowest possible level of abstraction instead of ones based at a higher level with many more inter-component interactions.

We intend to test code involving third party service providers by using the testing SDKs provided by them (e.g. Facebook's FBSDKCoreKit). These SDKs allow us to mock the functionality provided by third party services to test how well they have been integrated with our code.

We used the XCTestCase class that Swift provides for all of our implemented tests.

Results

We have implemented and documented unit as well as integration tests for the Model and ViewModel code which do not involve the use of third party APIs. As stated above, we do have a strategy for testing these sections of code, but decided to forego it for Sprint 2 to focus on finishing our deliverables due to the time required to set up the testing SDKs. Similarly, we also did not implement UI tests in code but followed the written plan rigorously on our own devices/simulators. We placed more emphasis on implementing tests on smaller modules and for invariants in the system.

Reflection

Evaluation

According to our initial proposal, we targeted to complete the following during sprint 3:

- Liking, edit, and removal of Messages
- Module permissions
- Star Chat rooms
- Notifications
- Local Media cache
- Refactor Threads
- Most liked message in forum posts
- Pagination
- Polish UI-UX
- Refactor code

We think that our group has finished the requirements that were in our proposal and discussions with Prof. We achieved this by working together intensively and having regular meetings to work on the implementation together. This helps us to achieve higher productivity.

Lessons

We learned about different and important design patterns that are needed in a messaging app to ensure features are extensible, which are also common in different types of applications as well. We explored a lot about cryptography and how to integrate it into our application. Having to implement it ourselves really forced us to understand the lower level details of the algorithms and ensure correctness.

We also learned that working together on a Swift project has other challenges that we might not encounter in other platforms. For example, during the initial step of our implementation, we need to manually resolve conflicts in `pbxproj` file which causes us a lot of trouble. This was caused by us initially branching our code and then importing the different Podfile dependencies in our different branches.

Finally, we realise the importance of planning and having a common understanding of the design. There were many hours which we spent to decide on the architecture of our design and more time spent ensuring that we all are on the same page before writing the code. We might also come with a different understanding of various concepts, such as the meaning of ‘State’ and the role of ViewModel. It is important that we clarify the terms we used so as to reduce miscommunication.

We learnt the importance of keeping a single source of truth and how to keep our model in the front end, in sync with the data in the cloud. This allowed us to explore the use of facades and queries to keep the 2 components in sync while still keeping them loosely coupled.

Known Bugs & Limitations

- UI Limitations

  - Message reply
    Currently, the user cannot click outside the two vertical lines to scroll and go to the replied message.
  - Media
    Currently, the user cannot click zoom/display the media in the application.
- Cryptography Limitations
  Currently, users are unable to sign in to different devices using the same account because of the way we do the Public Key Cryptography exchange and local storage of the key to keep it safe.

Appendix

Test Cases

Link to the test cases:
https://docs.google.com/document/d/1g18J6kr7NKdZE49F3b73CwhHWUXze5uWiRuiwuJs_jM/edit?usp=sharing