Slack Mac App Redesign

Detailed Design

Seong Moon
Quyen Tran
Jack Weinkselbaum
Luming Yin

3/11/18

Table of Contents

List of Figures

Terminologies

AppKit (a.k.a: Cocoa)

A UI + model layer framework for developing native macOS applications from Apple.

CoreData

A framework from Apple that both enables data persistence through a local SQLite database and relational graph to navigate within the model layer.

Keychain

A built-in secure database in macOS that is cryptographically encrypted at-rest.

UserDefaults

A mechanism that allows an app to document and retrieve user preferences with a persistent, per-app property list XML file

Electron

A JavaScript-based framework that allows developers to develop cross-platform desktop applications.

View Controller

A class that coordinates changes in model classes and corresponding view classes, as well as managing the life cycle of views it manages.

WebSockets

A protocol used by popular web applications to enable real-time client/server data exchange. Slack implements WebSockets with a custom data scheme.

Window Controller

A class that controls that life cycle of windows it manages.

1. Introduction

Last Fall, Team 7312 Accelerate decided that our goal for Junior Design was to create a better Slack app on MacOS. Slack is quickly becoming a mainstay in group communication and we felt as though their Mac app could be built better. Slack was built in 2013 to allow for business groups to communicate in a quicker and more relaxed manner versus email. There is an emphasis on IM-like communication throughout the different “channels” for each team. Our client, Kelly Ann Fitzpatrick, is part of many Slack groups as part of her work. She currently is a Brittain Fellow at the School of Literature, Media, and Communication at Georgia Tech. She is currently searching for a solution to solve her issues that come with being part of around 8 Slack groups. One of her main issues with the Slack app has been the lack of a unified timeline for her many groups. She has trouble being able to keep track of all of her groups that she’s a part of and envisioned it as a better way to keep track of her group communication. The other issue comes with Slack’s load time when it comes to launching the app. Slack has a much slower load time than other native Mac apps because it is based on Electron, which is a JavaScript container. This means that the Slack app is just a JavaScript app masking as a native Mac app and as such, has a slow load time, which causes frustration for those who use it.

To begin solving the speed issue, we built the app using Apple native Swift and Objective-C. This allowed for startup time to be much faster and allowed for a much smoother experience in general. We also implemented a customizable unified timeline to allow for the user to decide how they would like their experience to be. Users can choose what channels and teams they would like to be part of their unified timeline so only their most pertinent information could be found in the unified timeline.

After talking to Kelly Ann, we agreed that a unified timeline would solve the issue of having too many Slack groups and channels. To solve this issue, we would allow a user to select specific channels that they would want to be part of the unified timeline. That way, they can select their most important channels to be put into the unified timeline instead of everyone. This allows for users to concentrate on the unified timeline alone almost exclusively because less important channels will not make up the feed.

In this document, our app will be explained in much more detail. Section 2 will provide information about the system architecture of our app and how everything was built and flows together. Section 3 provides information on how the data for our app is stored and the design behind our data storage choices. Following in Section 4, we will show our component detailed design. Finally, in Section 5, we will go over our UI design, showing off the different screens of the app and our rationales behind our design decisions.

2. System Architecture

As Apple encourage macOS apps developed with AppKit to adopt the model-view-controller system (MVC) architecture, Team 7312's Slack client is developed with such MVC architecture. UI elements, widgets, and various screens are implemented in the interface layer, while the networking logic is implemented in the application logic layer. Underlying both of them is a data layer, which contains classes that encapsulate core data types, such as a Message, a User, or a Channel.

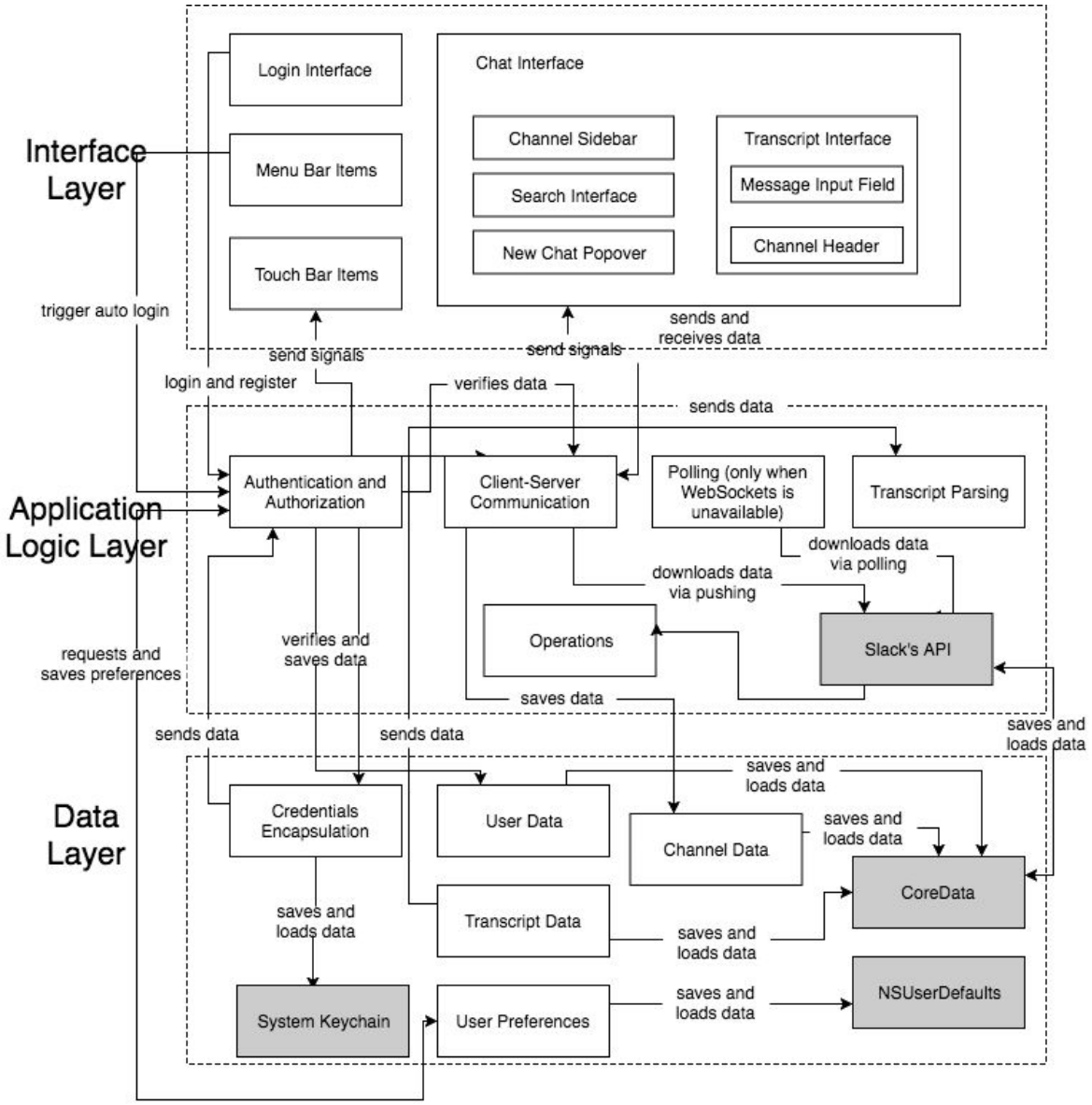

Figure 2.1: 3-Layer System Architecture Logical Diagram

Figure 2.1 illustrates the system architecture of the Team 7312 Slack app. The architecture is composed of an interface layer, an application logic layer, and a data layer. Blocks make up the composition of each major layer. Each sub-layer with a white background is implemented as an Objective-C class. Each sub-layer in grey is either directly imported, or self-implemented as a framework. Overviews of interactions between each component are highlighted in the diagram. Specific use case visualization between each block is illustrated in Figure 2.2.

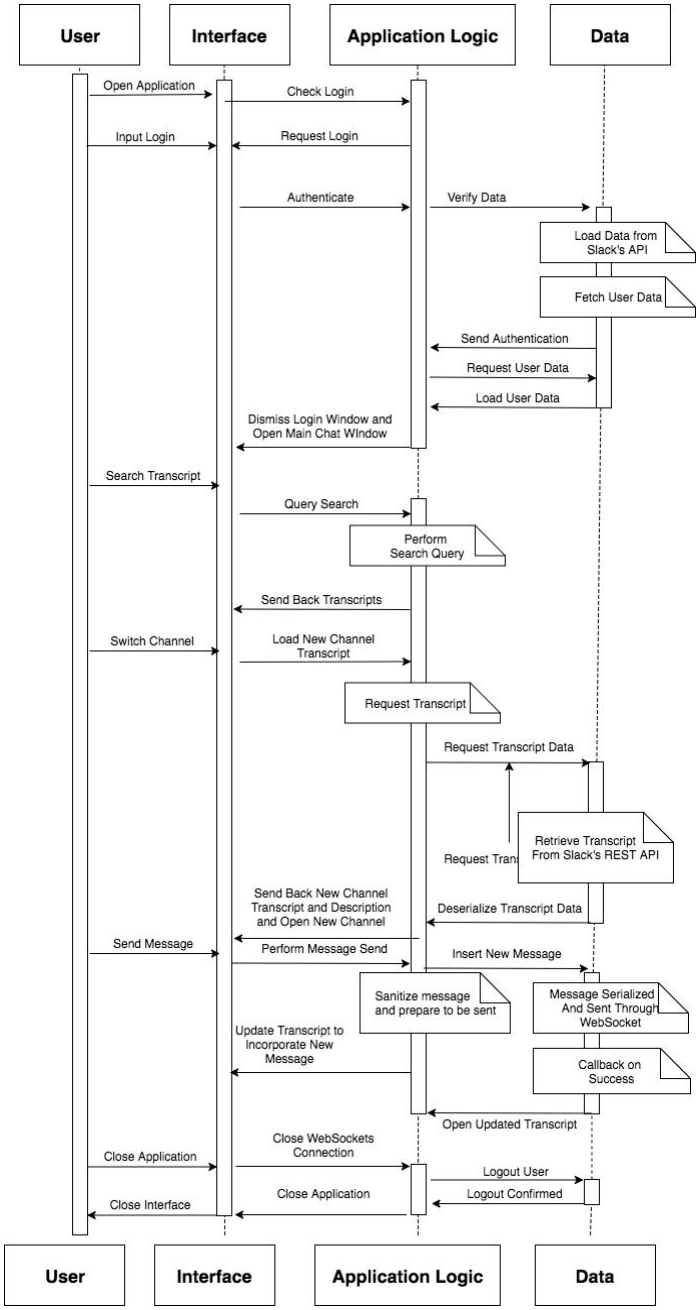

Figure 2.2: System Sequence Diagram for Sending Message

Breakdown of each layer's implementation overview and use cases follows:

Interface Layer

- The interface layer contains user-facing modules that can be interacted upon by the user. They are implemented with the basic building blocks of Cocoa views and view controllers, where the former are NSView subclasses and the latter are implemented with NSViewController subclasses.

- Rather than through programmatic means, some interface elements are created with Interface Builder, which are serialized on disk as a .xib (Apple's XML format describing a series of UI elements), then deserialized when the app launches.

- It is expected for the application logic layer to interact with the interface layer. As a result, it is important to note that while the interface layer, by itself, contains no business logic, the existence of certain components within the layer is determined by the application logic layer. For example, while Touch Bar Items are a part of the interface layer, its availability depends upon user's hardware capabilities. Similarly, while Menu Bar Items is an interface component, the availability (whether it is greyed out) and state (whether contains a checkbox) are determined.

Application Logic Layer

- The application logic layer manages certain aspects of the interface layer, while more importantly, serves as the primary mechanism to authenticate users, fetch chats, send chats, and keep the app in sync with the server.

- Authentication is done through the OAuth 2.0 protocol, where components within the application logic layer retain a temporary token that is valid within 24 hours of initial authorization to reduce round-trips needed between logic and data layer. A permanent record of user account and information are not cached in the logic layer but will be fetched on demand from the data layer.

- The logic layer also maintains one or several WebSockets connection with the Slack server. It performs standard WebSockets operations, with the addition of sending ping and parsing pong requests from the server every 5 seconds. This satisfied Slack's API requirements and ensures the connection is not cut off by the server endpoint.

- It may be intuitive to think of all communication capabilities of the logic layer as a separate component within the application logic layer, but that would be a drastic simplification of the actual structure. User-facing communication capabilities are intervened with protocol-level connection capabilities, where communication requests, such as outgoing messages, incoming messages, or user status update (transition from

online to offline), are all handled in coordination within components within the logic layer. A majority of these coordinations are internally implemented with delegate or block callbacks. A minor subset of these coordinations that are sparsely encountered are implemented as the broadcast and receival of NSNotification through NSNotificationCenter. This balances coding efficiency, maintainability and runtime performance.

Data Layer

- With CoreData and the relational object graph it provides, user data, transcript data, and channel data are all represented with auto-generated NSManagedObject subclasses. The CoreData framework enables fast runtime query of all data components.

- It is important to note that while the app is multithreaded, all data operations and exchange happen on the main thread. This indicates that upon saving database changes to disk, there will be an inconceivably short period of lock-up on the main thread. We made this architectural decision to sacrifice an insignificant amount of performance in order to improve resilience against problematic data merges or locks when exchanging data between different managed contexts on multiple threads.

- For non-sensitive preference data, they are stored and retrieved with NSUSerDefaults, which are in turn stored under ~/Library/Preferences/com.team7312.slack.plist. We note that while our code path for NSUserDefaults implementation does not vary across macOS releases, Apple made significant under-the-hood changes to the way NSUserDefaults is implemented. Under macOS releases equal to or newer than High Sierra (10.13), the property list is cached in memory by system daemons. Therefore, manually editing the preference file, or writing it with an API other than NSUserDefaults such as NSFileManager or the UNIX file API may put the app in an inconsistent and unrecoverable state.

- Additionally, sensitive elements are persisted through the macOS system keychain and do not go through the CoreData stack. This ensures that even if the user's local filesystem has been compromised, as long as there is no memory photography (a.k.a: memory dump) taken, an attacker cannot extract user credentials from the app. We do not block memory photography as it is blocked by default on macOS with the built-in System Integrity Protection mechanism.

3. Data Storage Design

For Database Use

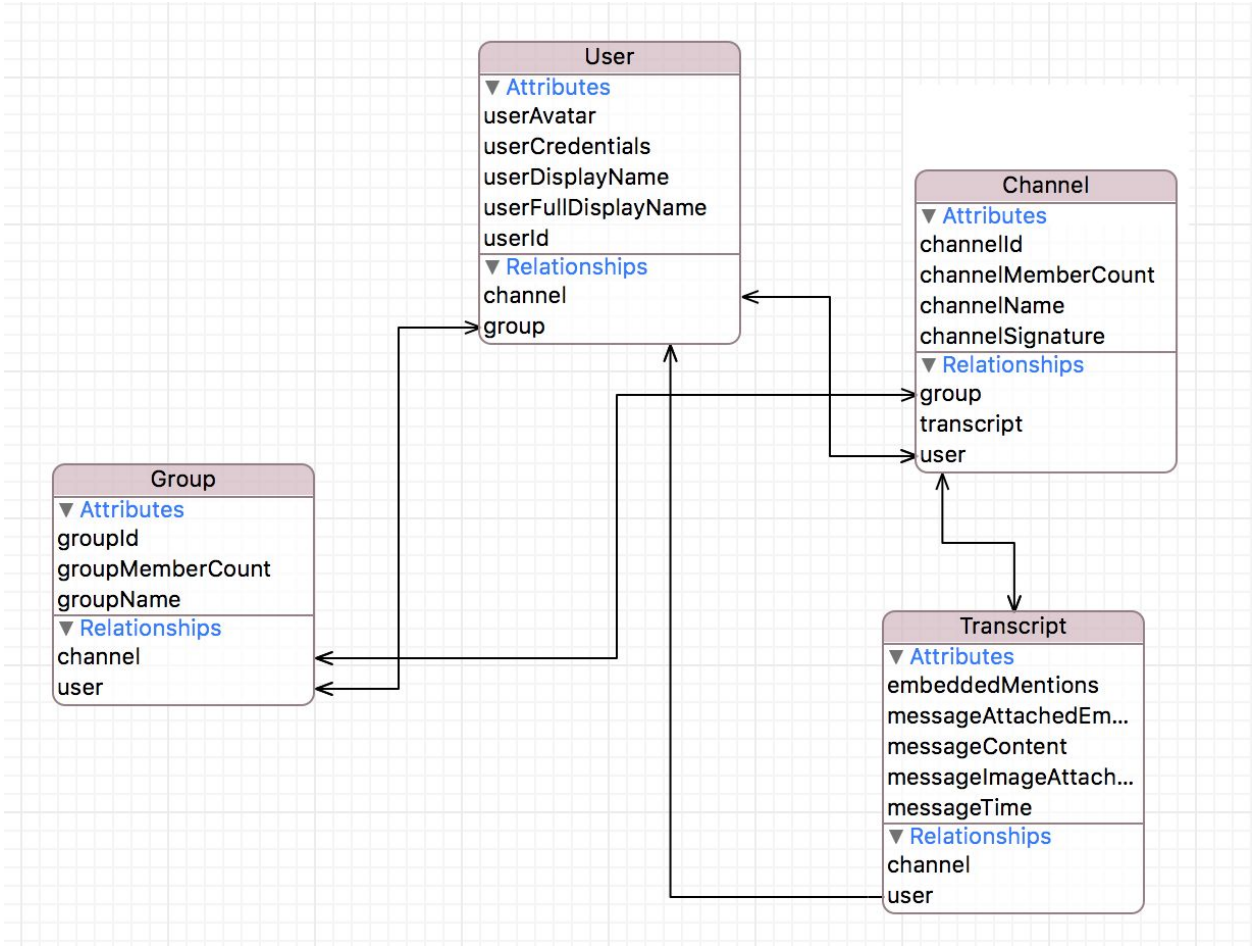

Figure 3.1: ER Diagram/relational Graph of Database for Team 7312's Slack App

Figure 3.1 represents an ER diagram that satisfies the need for persistence and quick data lookups for our third-party Slack client. The diagram highlights entities, attributes and the relationships between various User, Group, Channel, and Transcript entity. CoreData, NSUserDefaults and Keychain are used for data persistence.

Specifically, each significant entity contains the following data:

1. **User**: Contains information of any user, whether it is the currently logged-in and authenticated user or another user (individual) that is a part of the same team as the currently logged in user. Some properties, such as userCredentials, is both fetched from keychain at runtime and is optional.

2. **Group**. Contains its own attributes such as its ID, member count, and name, as well as bi-directional relationships with every channel within the group and users that are registered as a part of the group.

3. **Channel**. Similar to Group, it contains its own attributes, including ID, member count, and name. In addition to these, it also includes bi-directional references to which group it is a part of, the entire message transcript it contains, as well as all users that have joined the channel. Private direct messages that are one-to-one are internally represented as a channel with only two users, which is consistent with the JSON data Slack's backend API returns.

4. **Transcript**. Represents one particular chat transcript entry. Contains data needed to properly display a message transcript snippet, such as its time, content, image attachment, emoji reactions and embedded '@'-based mentions. Contains a one-way relationship to the user it is from, as well a bidirectional relationship of the channel it is a part of.

For File Use

Main Database: The main database is stored in SQLite format.

Small Attachments: Images and attachments that are smaller than 100KB are stored as a binary blob within the database.

Large Attachments: Images and attachments larger than 100KB are stored under the application's sandboxed container directory, /Users/Library/Containers/com.7312.slack/Library/Caches/Attachments, as cache files, while the path to the actual file is stored in the SQLite database instead.

Preferences: Some preferences, such as whether auto-login is enabled, or the previous window size, are stored in a Property List (.plist) format under /Users/Library/Preferences/com.7312.slack.plist and are fetched at runtime through NSUserDefaults.

Credentials and Tokens: They are stored as a part of macOS's built-in keychain. The keychain's particular directory location is opaque to the application, while the access is transparently handled by calling system APIs.

For Server-Client Data Exchange

Most types of local chat data are deserialized into JSON before they are uploaded to the server using WebSockets or REST API protocols. Remote, server-side chat data can be

deserialized into a full, new NSManagedObject subclass instance. Data segments can also be inserted as (or replaces) a certain property of an existing NSManagedObject subclass instance when downloaded from the server.

Images and attachments are uploaded and downloaded as-is, without any parsing, serialization or deserialization. They are delivered in compressed JPEG or PNG format without a file extension. For downloaded image and attachments, we delegate the responsibility of determining image file types to NSImageView classes, and simply refer to the downloaded files with their SHA and MD5 checksum.

4. Component Detailed Design

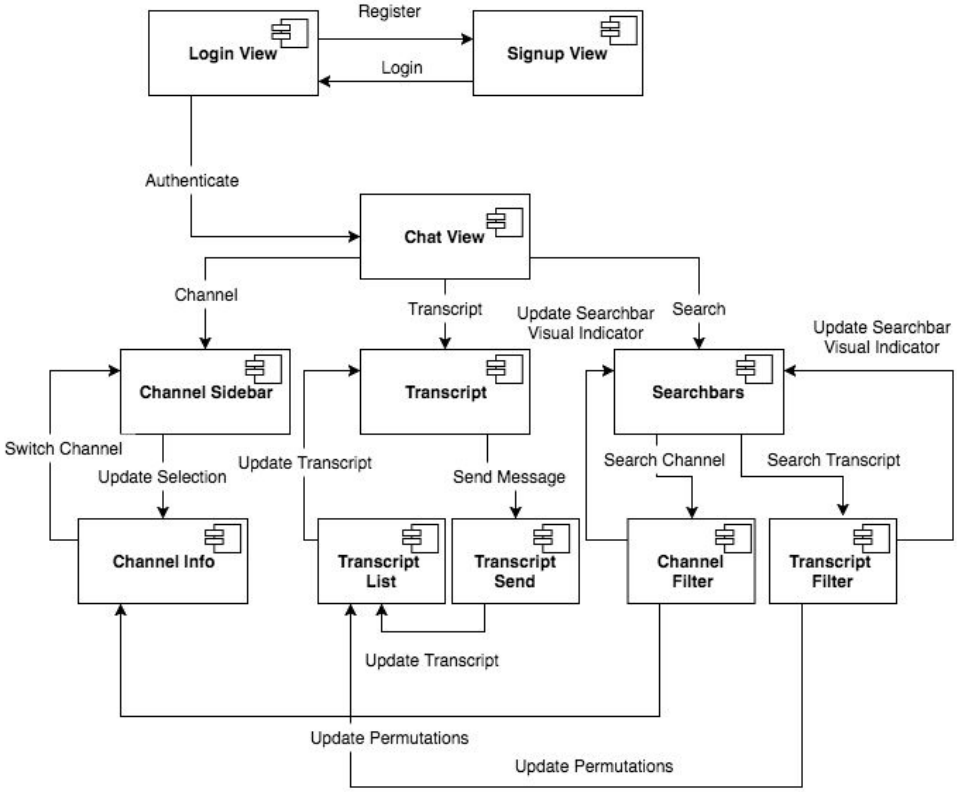

Figure 4.1: Static Components Diagram

Figure 4.1 shows the app's system components statically. Within the diagram, user interaction-able components are represented with square boxes, while the transition condition between various views and screens are labelled on arrows that connect each component or next to such arrows.

The application first displays a login view. When the user clicks the register button, the user is taken to a signup view. After a successful registration, the user is brought back to the login screen, which contains three text boxes - one for team name, one for username and another one for password. After clicking on the login button, the application authenticates with its model layer and calls upon Slack's API. After a successful sign in, the user is taken to the chat view.

Upon displaying the chat view, the window size determines whether all channel sidebar, transcript and search bars are displayed. Users can also manually trigger them by expanding and contracting them by clicking on relevant buttons. If the user clicks on an alternative channel in the channel sidebar, the channel selection is updated, new channel information is queried and the user is switched to the new channel. When the user sends a message, the local

transcript and remote transcript are updated. Additionally, transcripts may also self-update during a specified polling period or upon receiving a push from the origin WebSockets server.

When the user types in keywords in either one of the two search bars of the app, channel, and transcript filters are triggered, then the model layer generates an NSPredicate query, and a fast, local query (if possible) is performed. When no results are found, we pull down more past transcripts from the remote Slack API endpoint and perform the local search again. After the search completes, the visual indicator of the search bar is updated.

The static diagram is limited by its inability to illustrate the program flow in a dynamic context. We attempt to illustrate the dynamic transition between views and various states with the following dynamic component diagram, Figure 4.2.

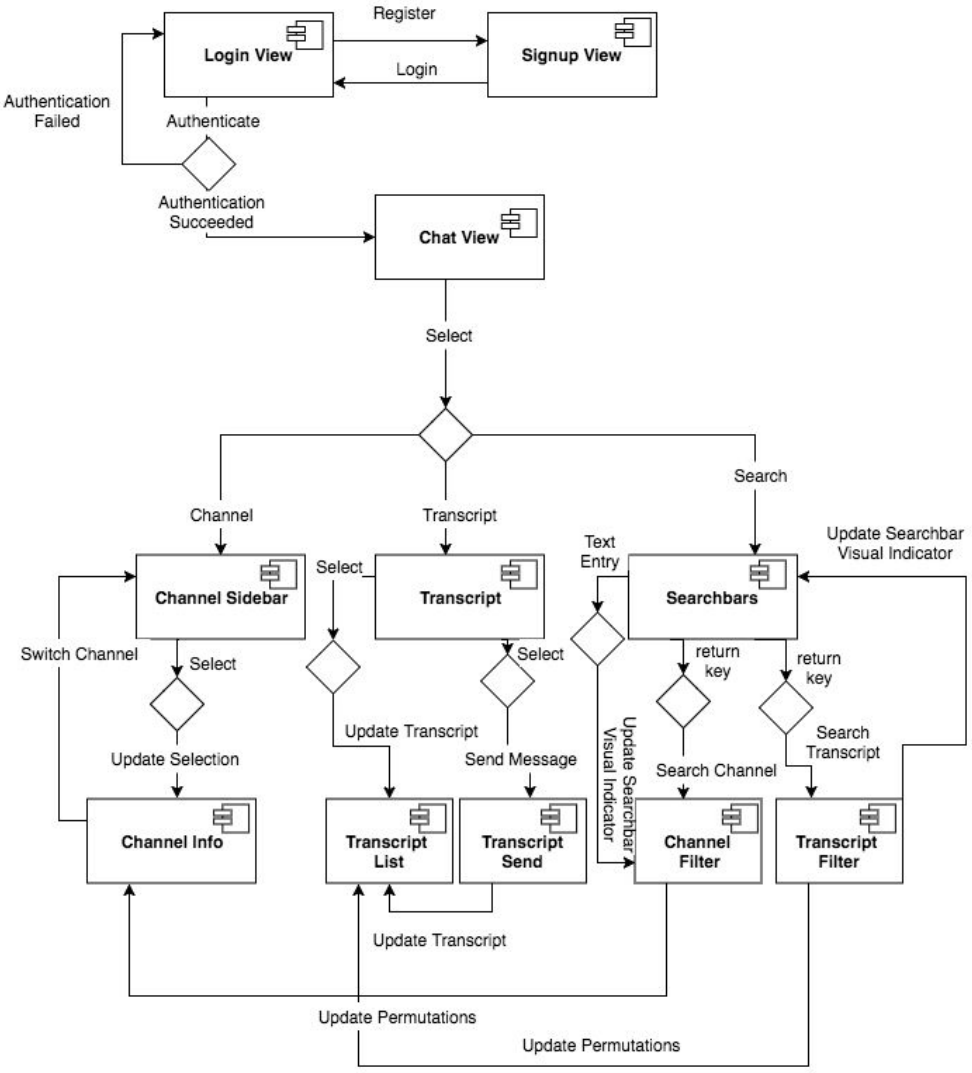

Figure 4.2: Dynamic Components Message Interaction Diagram

Figure 4.2, a dynamic component diagram, represents state transitions that has the program control flow in mind. While it is largely similar to the static diagram where boxes continue to represent application components, there are important differences, such as the introduction of diamonds, which symbolizes conditional code branches.

When the user enters a wrong or unrecognizable set of login information, an alternative code path becomes triggered and upon authentication failure, the user is brought back to the login view. Upon entering the main chat view, if the window is in compact size, the user's selection determines the following state transition, whether it is to display channel view, or transcript view or search view.

Within the channel sidebar, selecting an alternative channel triggers the channel selection to be updated. In the transcript view, a selection on one of the message bubble rows trigger the transcript list to be updated and display possible actions, such as to add an emoji reaction, while selecting the "Send Message" button or pressing the return key triggers the conditional code path of sending the user's draft message.

Additionally, when using the search bar, channel filters occur automatically whether text entry has changed, or return key has been pressed (since filtering channels are fast and less computationally expensive), while the transcript filter only fires when the user exhibits explicit behaviour to search for messages and manually presses the return key (as filtering through an entire channel's past transcript is slow, and doing so live upon text field value change may slow down the entire program or unnecessarily intensify network load).

5. UI Design

The following section contains screenshots from our Slack app, as well as descriptions of the flow of user experience within the app.

One of our main objectives is to closely follow Apple's HIG for macOS apps. Therefore, we designed most screens with translucent elements. This allows us to be in line with other native macOS apps. To highlight this, screenshots are intentionally taken under a colorful desktop background, highlighting our app's incorporation of vibrant background blur effects.

The application flow begins with the welcoming screen. Here, the user can proceed to the sign in or registration flow. After completing one of either flow, the user can access common Slack functionalities.

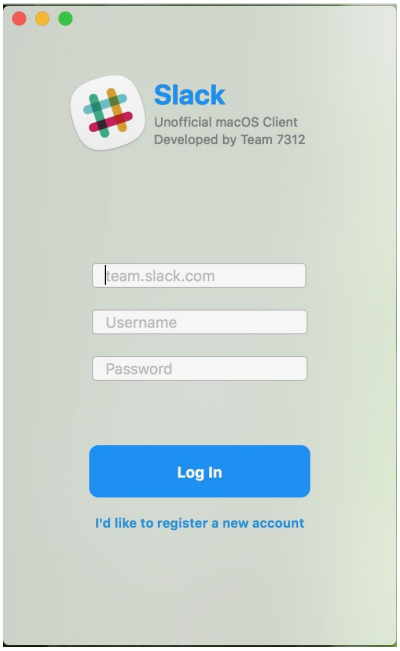

Figure 5.1: Welcome Screen

When the user first opens up the app or wants to add a new Slack group, this is the screen that they will see (Figure 5.1). They have the option of signing in to an existing Slack group or registering for a new group (Figure 5.2). If the user has entered the correct credentials, they will navigate to the main screen of our app (Figure 5.3).

The welcoming screen is designed to have minimal actions attributed to it so that the user is not presented with too many options, especially considering that this is the portal to the main section of the app. The user is presented with 3 fields: team URL, username, and password. Each of these fields provide hints for easy understanding of which information to enter.

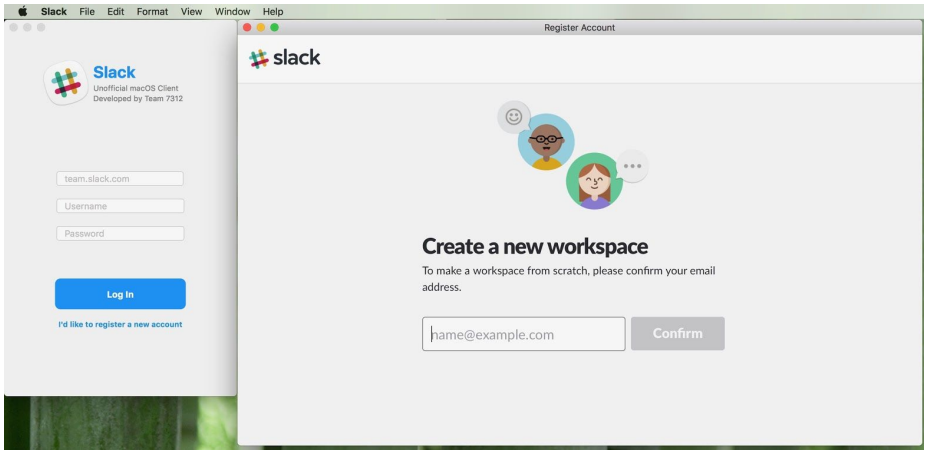

Figure 5.2: Registration Screen

Our registration screen (Figure 5.2) wraps the web version of the official Slack registration interface. We believe the official Slack registration process is well-designed, and kept it as is. It opens in a separate window within our app, rather than through a web browser. Therefore, the user would not have to have another browser tab open, minimizing confusion. Because there are no source lists or sidebars in the registration screen, we did not inject additional stylesheet entries to give it a transparent or translucent appearance.

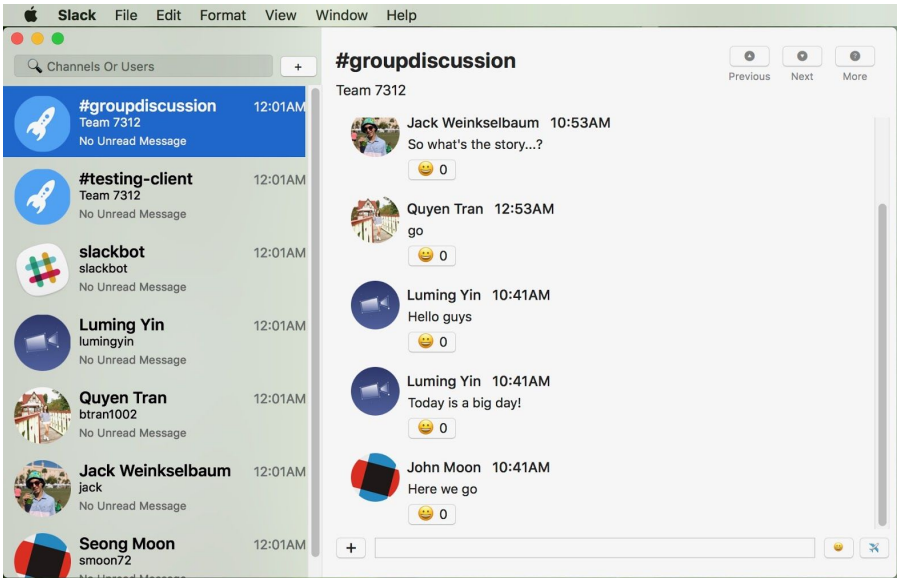

Figure 5.3: Main Screen

Figure 5.3 illustrates the main screen that a user will be greeted with once they are signed in. The left sidebar has a list of channels, as well as a list of people who the user has had private messages with before.

The main part of the screen is the transcript view, which mostly contains a list of messages from the selected channel. It also includes a text box for users to craft their message in. We believe giving the right pane a solid white background improves readability of the important messages that comes with a channel. This would also allow for greater contrast between the two columns, thus providing a visual cue to the user where their focus should be.

Underneath the message transcript, the smiling emoji button is used for finding emojis, which are a central part of Slack. The airplane emoji is used for sending the message. Since emoji are heavily integrated with Slack, we thought that we would make the send button an airplane to signify the message being sent. Similar to the official app, the user can also press the ‘return’ key on keyboard to send the message.

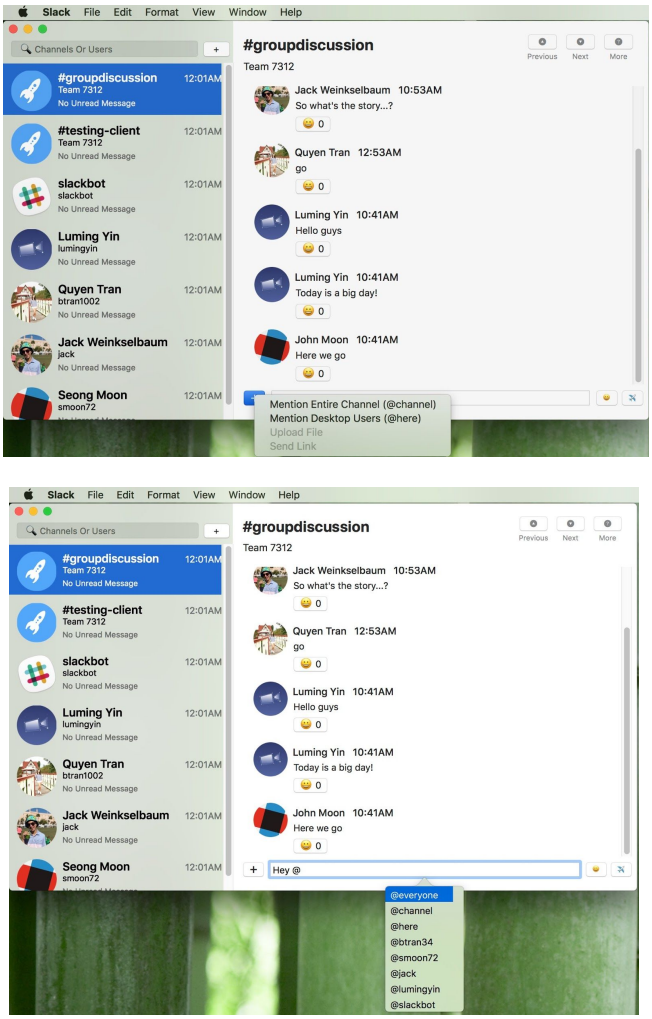

Figure 5.4: New Button Right Pane Menu

This plus button is used for two types of actions. One is to send out a group mention and the other is to attach stuff such as files or links. We felt as though putting the plus button next to the text box would provide visual cues that what the button does is related to messaging.

Figure 5.5: Mention Autocomplete Dropdown

As illustrated in Figure 5.5, as the user begins to enter the '@' symbol on keyboard, an autocomplete dropdown is presented, and the user can select one of the entries within the auto completion list, or keep typing to further narrow down potential entries. Then, the user will be able to press the 'return' key and specifically mention a group member. To be in line with native macOS apps, all context menus, as shown above, are translucent.

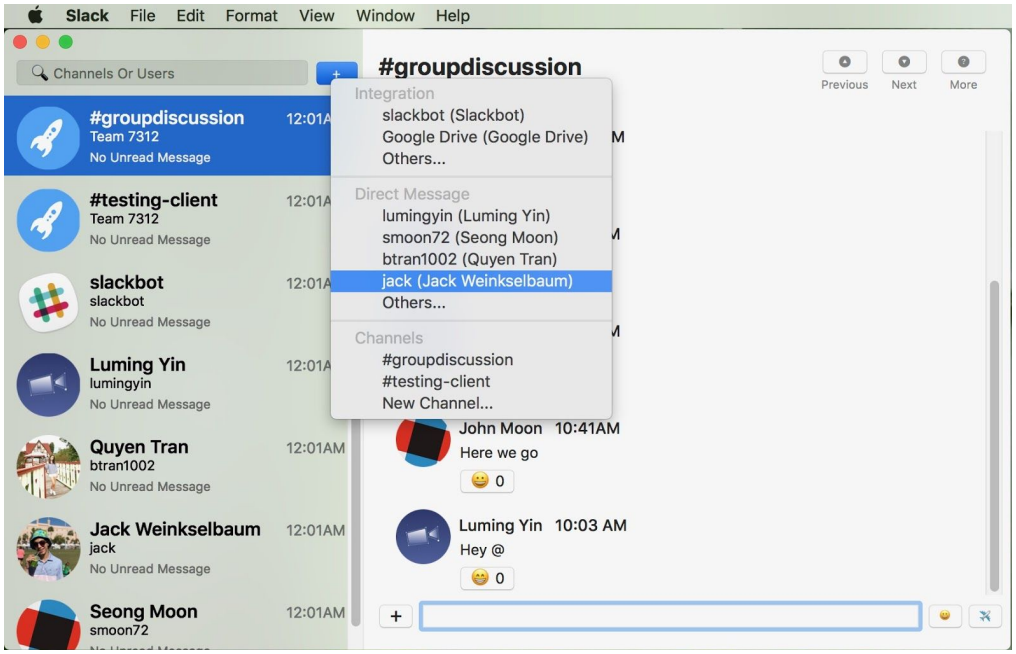

Figure 5.6: New Button Dropdown

The plus button in the left pane is used for quick actions. These quick actions are split up into integrations, direct messages, and channels. This segmentation allows the user to quickly gaze at the menu to find what they need to quickly. This button provides for quick access to commonly accessed things for the user.

In hopes of following Apple's design philosophy, we added translucency to the app when appropriate. We also added drop downs that are in line with that philosophy as well. Overall, we wanted to make sure that the user feels as though they are using a Mac app instead of a ported web app. This begins with the login screen and continues throughout the app from there.

References

[1] “Slack API: Real Time Messaging API,” in Slack API, 2018. [Online]. Available: https://api.slack.com/rtm
[2] “AppKit: Core App,” in Developer Documentation, Apple, 2018. [Online]. Available: https://developer.apple.com/documentation/appkit/core_app
[3] “AppKit: Data Management,” in Developer Documentation, Apple, 2018. [Online]. Available: https://developer.apple.com/documentation/appkit/data_management