4302306

Supervisor: Prof. Derek McAuley

Module Code: G53IDS

2020/05

Federated Modern Messaging Client Using Email as a Transport Mechanism

4302306

Supervised by Prof. Derek McAuley

School of Computer Science

University of Nottingham

May 2020

Submitted in partial fulfillment of
the conditions for the award of the degree **BSc Computer Science.**

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature ___________________

Date ______ / ______ / ______

Abstract

With social media style applications such as WhatApp and Facebook becoming the most popular form of communication, especially for small groups, security and privacy concerns have grown in recent times. The vast majority of services such as these have the prerequisite that users must have an email account. Email in its current form, alongside the existing clients for it, does not provide a rich enough feature-set to rival the social media style communication platforms, especially when it comes to group conversations with many participants. This project aims to take advantage of the federated communication network that is inherently available through email to create a new client application that enables messaging in a style similar to Facebook Messenger or WhatsApp, but using email protocols and addresses as the underlying transport mechanism. By integrating directly with email accounts, and providing end-to-end encryption as standard, this project presents an alternative form of communication that ensures that the privacy of users is protected.

Acknowledgements

I would like to thank my supervisor, Prof. Derek McAuley, for his advice and support on this project. I would also like to extend thanks to all those who gave their time to participate in the user testing process.

Contents

Abstract **i**

Acknowledgements **i**

List of Figures **v**

List of Tables **v**

1 Introduction **1**

1.1 Motivation 1

1.2 Project Aim and Objectives 2

2 Related Work **4**

2.1 Analysis of Existing Commercial Solutions 4

2.2 Analysis of Previous Email Client Research 4

3 Description of Work **6**

3.1 Requirements Specification 6

3.1.1 Functional Requirements 6

3.1.2 Non-Functional Requirements 8

3.2 Development Methodology 8

4 Design **10**

4.1 High-Level System Architecture 10

4.2 Message Threading Model 11

4.2.1 Message Schema 13

4.3 User Interface 14

5 Implementation **16**

5.1 Languages & Tools 16

5.1.1 Build Tools 17

5.2 Application Architecture 18

5.3 Security 19

5.3.1 Cryptographic Algorithms 19

5.3.2 Key Exchange 20

5.3.3 Key Storage 22

5.4 User Interface 22

5.4.1 React Component Structure 24

5.4.2 Redux 25

5.4.3 Improving Perceived Performance 26

5.5 Service 27

5.5.1 Sending Email 27

5.5.2 Receiving Email 28

5.5.3 Message Schema 29

6 Quality Assurance & Evaluation **31**

6.1 Continuous Integration 31

6.2 Code Style 31

6.3 Unit Testing 32

6.4 Functional Testing 32

6.5 User Acceptance Testing 33

6.5.1 Stage 1 34

7 Reflections & Summary **36**

7.1 Project Management 36

7.2 Contributions and Reflections 38

7.3 Future Work 38

Bibliography **39**

List of Figures

4.1 Low-fidelity wireframe showing the key user interface elements 15

5.1 Flowchart illustrating the build process 17

5.2 Application architecture diagram 18

5.3 Sequence Diagram showing key exchange process for end-to-end encryption 21

5.4 Login screen UI implementation 23

5.5 Main conversation screen UI implementation 23

5.6 Component composition tree for the Login Page 24

5.7 Component composition tree for the Main Page 24

5.8 The Redux Flow 25

5.9 Flowchart illustrating the message sending process 26

5.10 Flowchart illustrating the message sending process 27

6.1 Continuous Integration pipeline stages on Travis CI 31

6.2 Example of some of the unit tests written for the software 33

7.1 Project Gantt Chart 36

7.2 Burnup chart showing number of completed work items over time 37

List of Tables

2.1 Feature comparison of existing commercial messaging applications 4

1. Introduction

1.1 Motivation

A 2015 study found that smartphone users spend an average of 32 minutes per day on WhatsApp [1], highlighting the fact that in modern times, social media and instant messaging applications are considered by many to be one of the most convenient and favoured methods of communication [2]. Users enjoy the simplicity and ease of use of the small group communication that these applications provide, however relying on these social media services for communication also presents a number of privacy concerns. For example; many users are unaware that messages sent over Facebook do not have end-to-end encryption by default, and that this feature must be manually enabled for individual conversations [3]. The result of this is that Facebook has the ability to analyse the content of these unencrypted messages and use the data that it collects in whatever way it deems fit, such as harvesting data on users in order to target political advertising to them, a practice that was revealed as part of the Facebook-Cambridge Analytica scandal in 2018 [4].

In addition, even in applications which do claim to use end-to-end encryption, such as WhatsApp [5], if these are closed-source (as is the case for WhatsApp), it is impossible for users to truly verify the extent with which their personal data is protected. It is widely reported that WhatsApp uses the Signal protocol for end-to-end encryption [5], however WhatsApp users must trust that their private keys are not sent to the WhatsApp servers, which would allow WhatsApp (and its parent company Facebook) to decrypt and read the messages.

Despite the continued growth of social media services, email is still relied upon as the backstop communication method. Creating a social media account requires users to have an email address, and as such, social media users can be considered as a subset of email users. Since email already provides a communication method between people, it is seemingly unnecessary for people to sign up to third party communication providers such as WhatsApp, with its inherent associated risks as previously discussed. However, there are not currently any alternatives which increase the usability of email and retain its federated benefits. In addition, email is not encrypted by default, and current solutions to this are complex to set up and use, which means that they see little adoption [6].

Aside from the fact that there were 3.9 billion email users in 2019 [7]; representing over

50% of the world's population, using email protocols and addresses as a means for communication has a number of advantages [8]. Email is by design, the ultimate federated service, with email servers around the world working both independently and together to achieve the goal of delivering messages. In addition, email inherently supports self-hosting of mail servers, giving total control to users who require this - removing any need for reliance or trust of third-party providers.

Unfortunately, email clients have failed to keep pace with our reliance on email, and most have seen few major changes since their inception [9]. A problem with email in its current form is that it does not effectively handle group communication with explicit message threading in the same way that social media platforms do which makes conversations with numerous participants difficult to follow. The mismatch between the user interfaces for email clients and users' needs for handling email has been extensively documented, and one recurrent theme in work to improve the experience is that messages should appear as explicit conversations rather than existing independently [10].

Due to the underlying way in which emails are currently sent and received, simply changing the user interface is not enough in order to truly improve the user experience. Therefore, a system that combines the power of email as a federated transport mechanism with the usability of social media messaging appears to be worthy of further investigation and implementation with the intention of solving some of the issues presented above.

1.2 Project Aim and Objectives

The aim of this project is to design and develop an application that enables small, closed-group communication, using email as the underlying transport mechanism for messages, to mitigate the privacy concerns that exist in existing communication applications that use centralised architectural models.

The key objectives that have been identified as necessary steps to achieve this aim are as follows:

1. Investigate and design a suitable data structure for storing and transmitting message threads.
2. Investigate and implement a suitable architectural model for the system such that the federated aspect is preserved.
3. Design and build an intuitive user interface for the messaging system.

4. Implement the application logic for sending and receiving messages utilising the data structure designed in Objective 1.

5. Investigate and implement a suitable method of securing the messages in transit, to ensure that messages can only be read by the intended recipients.

6. Investigate and implement a method of allowing files, such as images, to be sent in message threads securely and efficiently.

2. Related Work

2.1 Analysis of Existing Commercial Solutions

A large number of commercial solutions that facilitate small group messaging, with explicit conversations and message threading built-in, already exist on the market for both mobile and desktop devices. Many of these services provide a much richer feature-set and user experience than can be realised using email alone with current clients. However, a problem is that none of these applications provide the federated, decentralised model that is such a benefit of email, instead each relying on their own servers for message transport and delivery. This model also requires users to create new accounts for each service; giving over their personal data in the process.

A selection of popular existing applications has been put together to highlight their individual strengths and weaknesses, which should be taken into consideration when designing the alternative. The results of this research is shown in Table 2.1.

	WhatsApp [11]	**Facebook Messenger [12]**	**Slack [13]**	**Signal [14]**
E2E Encryption	Yes	No by Default	No	Yes
Requires Account	Yes	Yes	Yes	Yes
Group Chats	Yes	Yes	Yes	Yes
Advertising	No	Yes	No	No

Table 2.1: Feature comparison of existing commercial messaging applications

2.2 Analysis of Previous Email Client Research

There has been some previous work to attempt to improve the email experience, however none of these attempts have focused on using email to create an interface similar to that of the solutions outlined above. One example of an attempt to build an email client that solves some of the core problems with email, including "lack of context" due to inadequate message threading systems, is the ReMail project from IBM research [15]. Whilst this project does attempt to improve the email reading experience, it leaves the way in which email is used fundamentally unchanged. Rather, the result of the ReMail project is a client that fundamentally still works like, and has all the quirks of traditional email communication, as opposed to the traits of modern messaging applications that are increasingly becoming the preferred way for people to communicate, especially within groups.

Another piece of research which highlights the complexity that can arise from email message threading in its current form is Bernard Kerr's paper on 'Thread Arcs' [16]. This research aims to assist users in understanding the structure of email conversations by presenting a visual representation of the way in which messages are threaded. Whilst not actually attempting to solve the issues with email threading, this study does present some of the key problems with the current methods of handling message threading and suggests a need for improvements.

This project will take a different approach to improving on current email clients. Instead of simply creating a new client interface for reading email, this project aims to ignore the current way in which emails are constructed internally. Instead, it will simply use email as the ultimate federated network of users, since a large proportion of internet users have an email account, to act as the transport mechanism for messages being sent using a newly designed schema that is optimised for group communication with explicit message threading within conversations.

3. Description of Work

3.1 Requirements Specification

The proposed solution to the problem described is to build a client application that incorporates the features of modern messaging clients which makes them so appealing, whilst using email as the underlying transport mechanism to take advantage of its distributed and federated architecture. Below is an outline of the functional and non-functional requirements that will be used to guide the design and implementation of this system. The requirements as defined below will be used as the basis for validation through User Acceptance Testing as the project progresses.

3.1.1 Functional Requirements

1. Login
    - 1.1. The application allows users to log in using any email account with IMAP & SMTP support.
    - 1.2. The application remembers login credentials after the first login.
2. Start Conversations
    - 2.1. The application allows users to start a new conversation.
    - 2.2. The application allows users to add an unlimited number of people to the new conversation, using their email address.
    - 2.3. The application allows users to set a name for each person that they add to the conversation.
    - 2.4. The application allows users to set a name for the conversation.
3. Send Messages
    - 3.1. The application allows users to send text messages in a conversation.
    - 3.2. The application allows users to send messages containing unicode characters, including emoji.
    - 3.3. The application allows users to send images and other file attachments.

3.4. The application shows a ‘sending’ indicator until the message has actually been sent.

3.5. The application allows users to ‘tag’ other participants in messages.

4. Receive Messages

    4.1. The application presents an operating-system-level notification when a message is received if the application is not currently in focus.

    4.2. The application highlights conversations with unread messages, removing this highlight once the conversation has been opened.

    4.3. The application retrieves any messages received while the app was not running when it is reopened.

5. View Conversations

    5.1. The application displays a list of all conversations that a user is part of.

    5.2. The application lists conversations ordered by the time that a message was last received within each conversation.

    5.3. The application displays messages for a specific conversation if it is selected by the user.

    5.4. The application allows users to read messages received in all conversations.

    5.5. The application allows users to view message attachments received in all conversations.

    5.6. The application displays messages within a conversation in the order in which they were sent.

    5.7. The application displays the name of the user that sent each message.

    5.8. The application displays the time that a message in the conversation was sent.

    5.9. The application allows users to change the name of an existing conversation.

    5.10. The application allows users to add new participants to an existing conversation using their email address.

6. Message Replies

    6.1. The application allows messages to be replied to directly by the user.

    6.2. The application shows replies to a message as a thread attached to the message.

    6.3. The application only allows top-level messages to be replied to. Replies cannot have their own reply thread.

    6.4. The application does not mark a conversation as having unread messages if only replies are unread in the conversation.

3.1.2 Non-Functional Requirements

1. The application is cross-platform; compatible with Windows, MacOS, and Linux operating systems.
2. The application is quick and simple to set up.
3. The application ensures that messages sent between participants are end-to-end encrypted using suitable cryptographic techniques.
4. The application performs well, with minimal perceived loading times and is quick to responsd to user interaction.
5. The application has a user interface that is attractive, consistent, and intuitive to navigate and use.

3.2 Development Methodology

This project will use an Agile [17] development methodology, to allow for flexibility in response to any unforeseen obstacles that may occur, in such a way as to minimise risk to the success of the project as a whole. In particular, the Kanban [18] method of project management will be utilised, along with elements of the Extreme Programming (XP) methodology [19]. In line with the Agile Manifesto, which states that working software is the primary measure of progress [17], continuous delivery of working software will begin as early as possible, with additional features being added to a minimum-viable-product incrementally.

This will be achieved through limited 'Big Design Up Front' as would be the case with a Waterfall development model [20]. Instead, incremental design will be favoured, since this allows for design decisions to reflect the problems encountered at different stages of development, that may not be foreseen up front without prototyping and research, thus allowing the project plan to adapt to changing circumstances.

The benefit of the Kanban methodology in particular is that any card can be taken from the project backlog at any time and placed on the Kanban board, removing the rigidity that can be enforced by fixed-length sprints in other Agile frameworks such as Scrum. This is ideal for this project, as the future direction will likely be dictated by research undertaken as the project progresses, such as research into the use of email protocols.

The XP concept of Test-Driven Development will be used throughout the project, as a means of ensuring that the code written meets requirements, and to ensure that any problems are found as early in the development cycle as possible, to minimise effort required in fixing them. Unit tests will be written alongside the code that they are designed to test, and a Continuous Integration (CI) pipeline will be used to ensure that all tests are passed before feature branches can be merged into the master branch. This form of regression testing ensures that new feature code does not cause breaking changes for existing code which is vital in ensuring that the software is of the highest possible quality.

4. Design

Although carrying out a significant amount of up-front design has been avoided, there are some high-level design decisions that have been researched and made in advance, to underpin the development process. These key design decisions surrounding system architecture, the message threading model, and user interface design are outlined below.

4.1 High-Level System Architecture

High-level architectural design decisions for this project have been heavily influenced by the requirement that the entire system should be federated, not relying on services from any individual provider. A number of possible architectural models were proposed as a result of research and these possibilities are outlined below.

1. A web based system, hosted on a cloud service provider such as Amazon Web Services (AWS) [21], Google Cloud Platform (GCP) [22] or Microsoft Azure [23].
2. A locally installed application which spins up a compute instance on a cloud service provider on demand for each user for handling computationally expensive work.
3. A locally running web-based user interface which communicates via standard web protocols such as HTTP with a separate locally running service process.
4. An application which encapsulates the frontend user interface and backend service into a single executable that can be run locally.

Each of the models listed above present their own advantages and disadvantages. Although Model 1, an entirely web-based system, would mean that the software is accessible from anywhere, by any device with a web browser, which would improve the user experience, it also means that users have no choice but to trust the backend of this software, running in the cloud, with their data, which means that the federated aspect of the system is lost and privacy concerns are not resolved.

In Model 2, users would have more control over their data in the cloud, as they are responsible for managing the cloud compute instance, however implementing this model would mean that users are required to have in-depth knowledge of AWS or similar, which detracts from the usability of the system, violating the non-functional requirement that the application should be easy to set up and use.

The solution proposed in Model 3 does not rely on any remotely hosted software, and so maintains the integrity of the federated system, ensuring that the user always has full control over their data. Furthermore, by keeping the service as a separate entity, it would be relatively simple to access it from other devices, for example a mobile phone on the same network as the host PC, using a consistent API. However, by requiring users to manually start the user interface, navigate to it in the browser, and then start the service, it increases the barrier to entry for less technical users.

The architectural concept proposed in Model 4 will be easy for users to set up and run since it will consist of running a single executable. The chosen method of implementation for this model is an Electron application, which allows for web technologies to be used to build a desktop application [24] meaning that UI development will be fast compared to, for example; the intricacies of building attractive user interfaces in Java. A disadvantage of using this model is that the application will only be available on desktop devices.

It is clear that the final decision will need to be a compromise, and that any architecture will have flaws. It was decided that the most important requirements that the system architecture should fulfill are ease of use, to ensure that the software is accessible to as many users as possible, and that the system should be entirely federated. Therefore, the most suitable architectural model, and the one to be used in this project is Model 4, the Electron application. Despite the fact that this limits the platform to desktops only, it provides a good basis for build a proof-of-concept that presents the future of communication.

4.2 Message Threading Model

An important design aspect of this project is designing a suitable message threading model to maximise functionality and usability. Indeed, one of the motivations for completing this project is that email as used in its present form does not handle complex threaded conversations well due to its linear and static structure. The need for message threading stems from the requirement that users are able to directly reply to messages, which is more suited to a dynamic conversation structure, not just sequential messages. In designing this system, it was noted that current messaging software products take various different approaches to message threading. These existing approaches must be evaluated in order to ascertain the most appropriate for implementation of this project.

There is a considerable amount of previous work on how best to handle message threading, particularly within the scope of email. Lewis & Knowles stated that: "While user clients typically insert in messages structural information useful for recovering threads, inconsistencies between clients, loose standards, creative user behavior, and the subjective nature of conversation make threading systems based on structural information only partially successful." [25]. This project aims to avoid the issues of client inconsistencies and loose standards by providing a specialised client application and well-defined data schema to be sent in the email body, with email headers and other such structural information playing no part in the structuring of conversations in the new system.

It is important to realise that, depending on the approach taken to message threading, one message can have multiple replies, each initiating a separate message thread. For this reason, message threads are best modeled using a tree structure [26]. In email, there does not exist an explicit reference from a message to its replies, so the relationship can be described as unidirectional from reply to parent. Some email clients provide a feature to generate these reverse links, which are useful to allow a user to see if replies already exist before composing their own [26], though this functionality is by no means universal. Outside of email, there is a mix of bidirectional and unidirectional relationships between replies and their parents.

In order to design the message threading model for this project, a review of two popular existing social messaging applications with different threading models was conducted, highlighting the advantages and disadvantages of their threading models. The results of this research are summarised below.

- Slack Model [13]
  - A message provides links to its replies.
  - Only one level of replies is allowed. Replies cannot be replied to.
  - Replies can only be accessed via their parent message.
- Facebook Messenger Model [12]
  - Replies can be traced back to the parent from any depth, but a parent message does not provide any links to its replies.
  - Replies can be replied to up to an infinite depth.

It is interesting to note that during development, the message threading model used by Slack went through many different iterations involving both conceptual structure and user interface design [27]. This research at Slack found that when users were allowed to reply to replies, as is the case in the Facebook model, threads quickly became very complex and difficult for users to follow. This led to the approach currently implemented in Slack where each thread is restricted to one level of replies [27].

The final design decision is a tradeoff between functionality and usability. Whilst allowing greater depth of replies provides users with more flexibility in the way that they construct conversations, it also detracts from usability, with it becoming more difficult for users to follow the natural flow of discussions that they are interested in.

It is proposed that this project will initially implement a threading structure similar to that of Slack, with replies being limited to one level deep, though a top level message can have an unlimited number of sequential replies at this depth. Following this implementation, user testing will be conducted to ascertain whether this model allows enough flexibility for users to hold discussions, whilst still being easy to use. If the results of this testing indicate a need for a more complex threading model then the design will be revisited.

4.2.1 Message Schema

With the threading model decided, it is now necessary to design a message schema that supports it. In designing the schema for the messages to be sent via email in this application, there were a number of factors to be taken into consideration. These included minimising the amount of data that would need to be sent in each message, and ensuring that messages can be queried quickly and effectively to minimise search times for the user. The schema must be able to handle the threading model proposed in Section 4.2, and has been designed with all aspects of this in mind. Details on the implementation-specific details of these message schemas can be found in Section 5.5.3. The final solution uses the following message schemas:

Message

```
subject: EMAILSOCIALMESSAGING:{conversationId}
body:    {
           messageId,
           conversationId,
           parentId,
           timestamp,
           senderId,
           content,
           participants,
           conversationName
```

```
         }
```

Public Key Request

```
subject: EMAILSOCIALMESSAGING-KEX
body:    {
           conversationId,
           senderEmail,
           participants
         }
```

Public Key Response

```
subject: EMAILSOCIALMESSAGING-KEY
body:    {
           conversationId,
           senderEmail,
           key
         }
```

Conversation Key Response

```
subject: EMAILSOCIALMESSAGING-CONKEY
body:    {
           conversationId,
           encryptedKey
         }
```

4.3 User Interface

An large part of this project is the creation of a simple and intuitive user interface, so that it can be used without a need for extensive instruction and training. Based on an analysis of the functional requirements outlined in Section 3.1.1, elements of the user interface that are considered as essential for meeting the requirements are as follows:

- Account credentials form
- ‘New Conversation’ button

- List of conversations
- Message viewing area
- Sender Indicators
- Message Composer
- ‘Add Attachment’ button

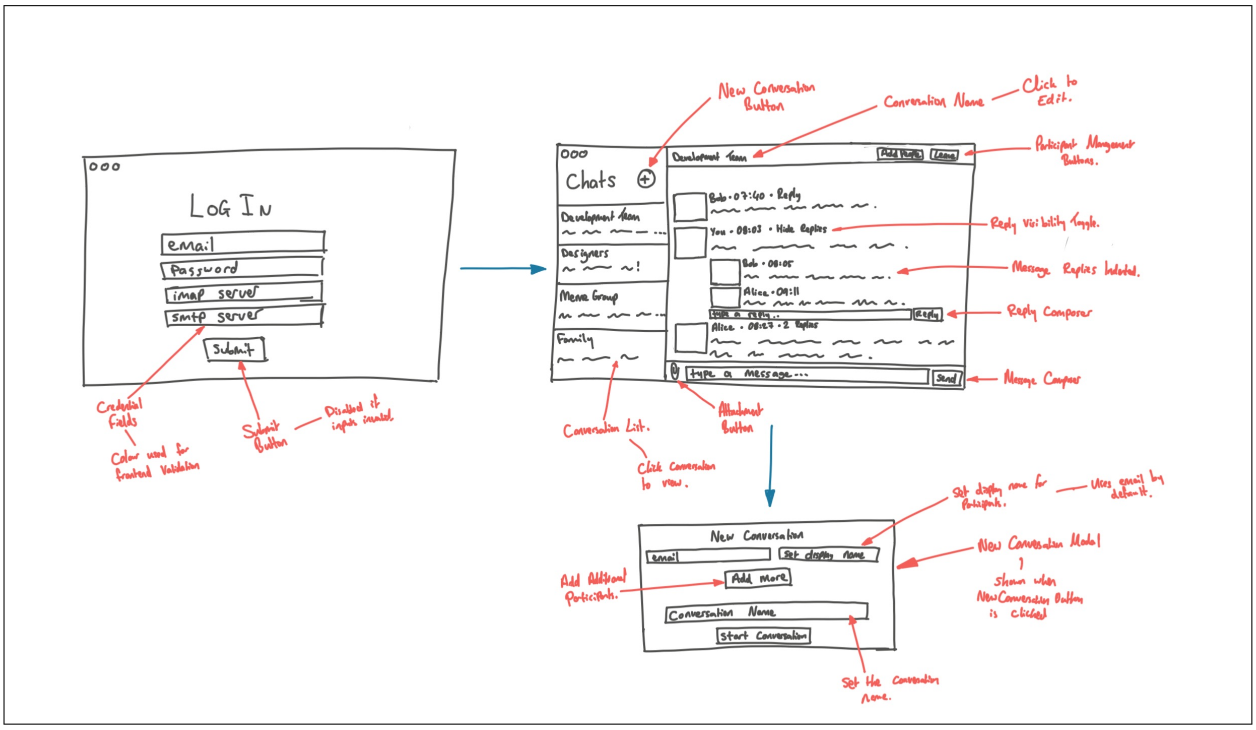


Figure 4.1: Low-fidelity wireframe showing the key user interface elements

Using these essential UI elements some low-fidelity wireframes were produced, seen in Figure 4.1. The wireframes have deliberately been created at a high-level, without large amounts of detail, to allow for changes as the project progresses through the Agile development process. These wireframes will be used to support building a user interface using React, however it is noted that the specifics of the design will likely change over time as the project evolves and user feedback is gathered. The designs should, however, aid in decomposing the design into user interface components and allow for the development of a Minimum Viable Product.

5. Implementation

5.1 Languages & Tools

As discussed previously, the application is going to be built with Electron. As a result of this, the choice of programming languages was limited to just two; JavaScript or TypeScript. TypeScript is a relatively new language, developed by Microsoft, that adds strict typing to the traditional web programming language, JavaScript [28]. Although the type checking provided by TypeScript is often found to lead to less bugs in software, it was decided that JavaScript should be used for this project. There are two main reasons for this decision. Firstly, TypeScript would require further learning before any implementation could be done, which would slow development of a minimum viable product (MVP). In addition, issues can be encountered if JavaScript libraries are to be used which do not have type definitions. Being able to use JavaScript for both the frontend user interface and the backend service means less context-switching and a more rapid development process.

Rather than using vanilla JavaScript, to aid in rapid development of the user interface, a Frontend JavaScript Framework was used. There were three candidates of frontend framework: React [29], Vue [30], and Angular 2 [31]. Vue has the smallest developer community of the three, so it was decided that the decision should be made between React and Angular 2, to ensure a good availability of supporting libraries and supporting documentation [32]. In deciding between React and Angular 2, a number of factors were considered. React allows for extremely fast development with JSX markup, is extremely performant due to the React Virtual DOM, and encourages Functional Programming which results in code that is easy to test and highly reusable. It is largely accepted that Angular 2, as a much larger framework has a steep learning curve, which could slow down development, especially given prior experience with React. Since there are few drawbacks of using React over Angular 2, and given prior experience with using React to build large scale applications in an industrial environment, the decision was made to use React in the project to allow for rapid development using modular, reusable UI components.

In addition to the core functionality provided by React, Redux [33] was chosen as the library for application-level state management. The primary reason for choosing Redux over other React state management libraries was that its immutable state model makes for extremely simple and efficient testing, which avoids lots of time being spent on writing complex unit tests for global state, which can be better spent on writing application source code.

5.1.1 Build Tools

In addition to the core tools being used for implementation, there are a number of build steps that are required to generate an executable for the application, which are supported by a chain of build tools. The build tools used in this project consist of the following:

- **Babel** is used to transpile the JavaScript from ES6, with JSX, into Electron-compatible ES5 [34].
- **Webpack** is then used to bundle the transpiled JavaScript, as well as any other required assets such as images, or stylesheets [35].
- **Electron Builder** then packages the application bundles into self-contained executable Electron applications for Windows, Linux and MacOS environments [36].

Separate Babel and Webpack configurations have been created for building each of the Electron processes that form the complete application, discussed further in Section 5.2. An overview of the entire build process can be seen in Figure 5.1.

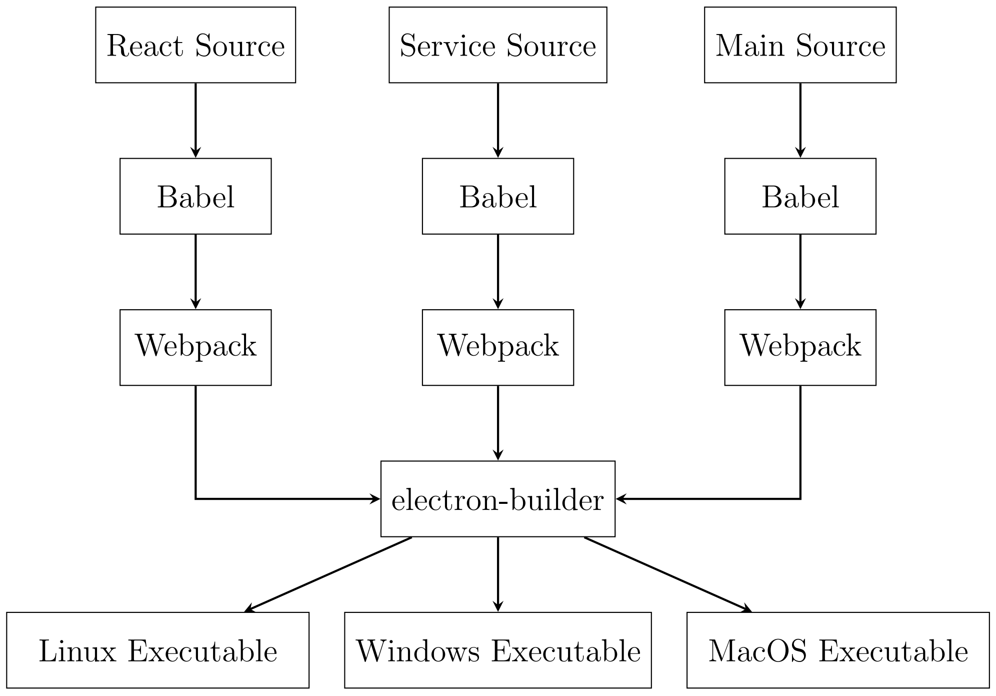


Figure 5.1: Flowchart illustrating the build process

5.2 Application Architecture

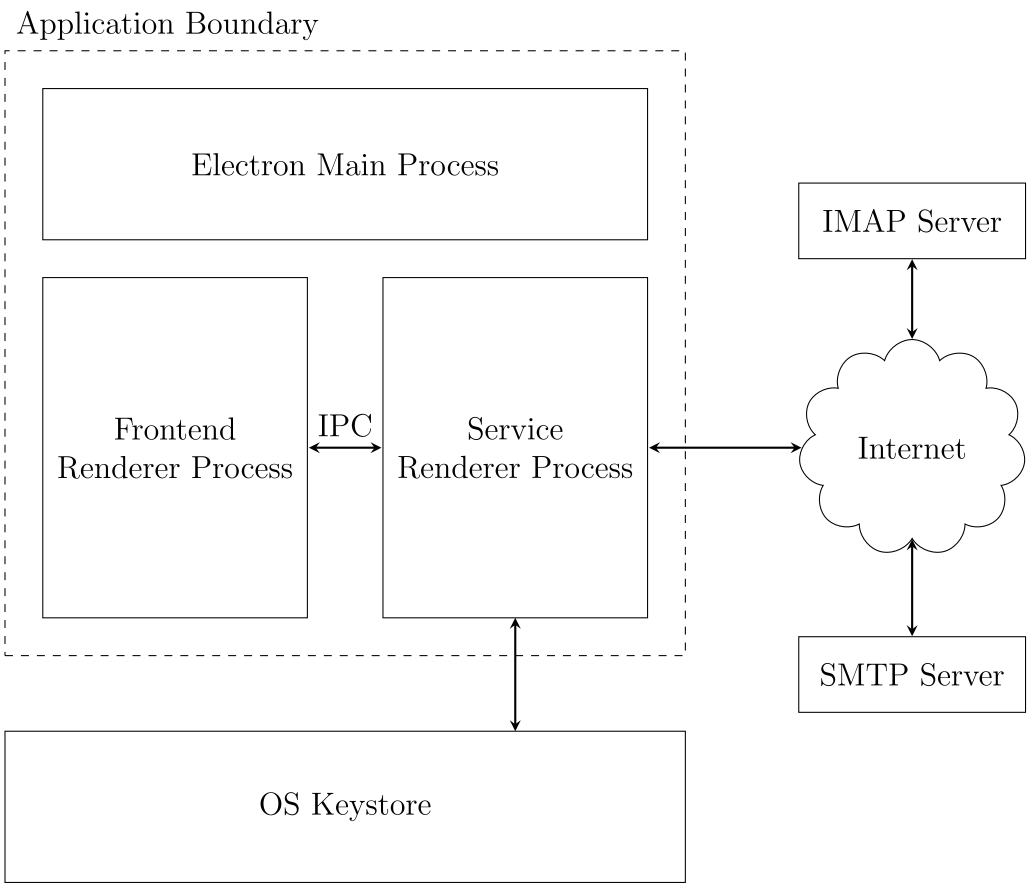


Figure 5.2: Application architecture diagram

As discussed in Section 4.1, the application is being developed using Electron. This design decision brings some specific stipulations regarding the application's internal architecture. It was decided that the frontend user interface part of the application should be decoupled from the backend service component. This ensures that the code is maintainable, and it also means that if the application was to be ported to a different platform in the future, such as mobile devices, the backend could be reused with minimal modification and just connected to a new native frontend.

It was decided that the best way to achieve this decoupling in the context of an Electron application would be to utilize two different renderer processes; one for the frontend and one for the service. Electron has two different types of process: the 'Main Process', and 'Renderer Processes' [37]. The main process is the parent process to all of the renderer processes, and it is the only process that is able to call native GUI APIs [37]. Each renderer process has the ability to optionally display a web page inside a `BrowserWindow`. Importantly, every application must have exactly one Main process but can have multiple renderer processes.

The main process contains the application UI thread, and therefore it is imperative that it is not blocked by long-running operations in order to avoid the whole application freezing until the main process becomes available again [38]. With this restriction in mind, the application was architected to make use of a main process and two renderer processes; one to house the frontend React app, and one to host the backend service responsible for managing the sending and receiving of email.

These two renderer processes are able to communicate with each other using Electron's built in Inter-Process Communication (IPC) functionality, which uses 'named pipes' internally to ensure fast and secure communication [39]. The inter-process communication in Electron is implemented through the event-driven architecture that underpins much of the core Node.js API [40]. This message-based IPC allows messages with optional data payloads to be sent between processes, and requires event listeners to be defined in the receiving process, to perform some task when messages are received.

In this project, IPC event listeners in the service renderer process are all contained in one file, for ease of maintenance, and often call methods on other objects that exist within the service, such as the `MailManager`. In the frontend process, in keeping with React best practices, the event listeners are most commonly written in the `componentDidMount()` method [41] of the component that requires the data from the service. This means that the event listeners are registered as soon as the component has been mounted to the React Virtual DOM. once received, the data is often then used to update the component state by calling `setState()` [42].

5.3 Security

One of the requirements for the application is that it must implement end-to-end encryption of messages that are being sent. This security security requirement presented a number of implementation challenges and contributed to a large amount of development time. Given that security and privacy were key motivators for completing this project, the security model of the application and its implementation is discussed in considerable detail.

5.3.1 Cryptographic Algorithms

An important technical decision that had to be made was the cipher that would be used for encryption of messages, as well as the key derivation function to be used to generate a symmetric key. The decision was taken to use the Advanced Encryption Standard, specifically AES-256, cipher since it is widely available, well documented and tested, and generally considered to be secure [43]. In addition, the ‘AES Instruction Set’ is integrated into many processors, meaning that encryption and decryption can be carried out extremely efficiently [44].

Rather than building the AES algorithm from scratch based on the specification, and risk making implementation errors that result in it being insecure, the project uses the OpenSSL implementation, via the Node.js crypto library. Since AES is a block cipher, it was necessary to choose a mode of operation to use when encrypting messages. The mode chosen was Galois Counter Mode (GCM) [45] as it effectively turns the block cipher into a stream cipher and so can be used to encrypt messages of any length. In addition, this mode of operation has the added benefit of generating a message authentication tag in the same pass as encryption, to ensure integrity, giving confidence that the message has not been modified. When messages are received, the authentication tag is also present, and if the authentication tag is invalid for the received message then it is silently discarded to protect against attacks. The 256-bit encryption key is generated using the PBKDF2 algorithm, with an input password and salt, each of 64 cryptographically random bytes, and SHA512 used as a digest. The number of iterations of PBKDF2 has been set at 100,000 as a tradeoff between speed of key generation and cryptographic security. Crucially, a new initialization vector, of 128 cryptographically random bytes is generated each time a message is encrypted, to mitigate the risk of crib-dragging in a known-plaintext attack.

5.3.2 Key Exchange

One important consideration that had to be taken into account when designing the security model for the application was that groups can contain multiple users, and end-to-end encryption must be in place between all participants in a group conversation, whilst minimising computational and communication overhead as much as possible. The first implementation idea was to utilise a Diffie-Hellman key exchange to establish shared secrets between each pair of participants, which are then used to encrypt messages. Whilst this implementation would work well between two participants, in a conversation with multiple participants, every message would have to be encrypted multiple times, with the different key for each recipient, and then sent separately to each recipient. This would make message sending slower and key management would be more difficult.

To avoid this overhead, it was decided that each conversation should utilise a single ‘Conversation Key’ which is known by all participants, and all messages within a conversation are to be encrypted and decrypted with this key using a the AES-256 symmetric cipher. There still remains the problem of ensuring that all participants receive the Conversation Key without it ever being sent across the network in plaintext.

To achieve this, asymmetric (public-key) RSA cryptography is first used to securely transport the Conversation Key between participants when a conversation is started, and then switching to faster symmetric encryption. Briefly, the client that wishes to start a new conversation will send a message to each participant requesting their RSA public key. When each of the public keys is received, the conversation key is encrypted with the public key and is sent back to the other participant. The participant then uses their private key to decrypt the message and gain access to the conversation key. The key exchange process utilises some of the message schemas defined in section 4.2. The sequence diagram in figure 5.3 shows the key exchange process between a group of participants.

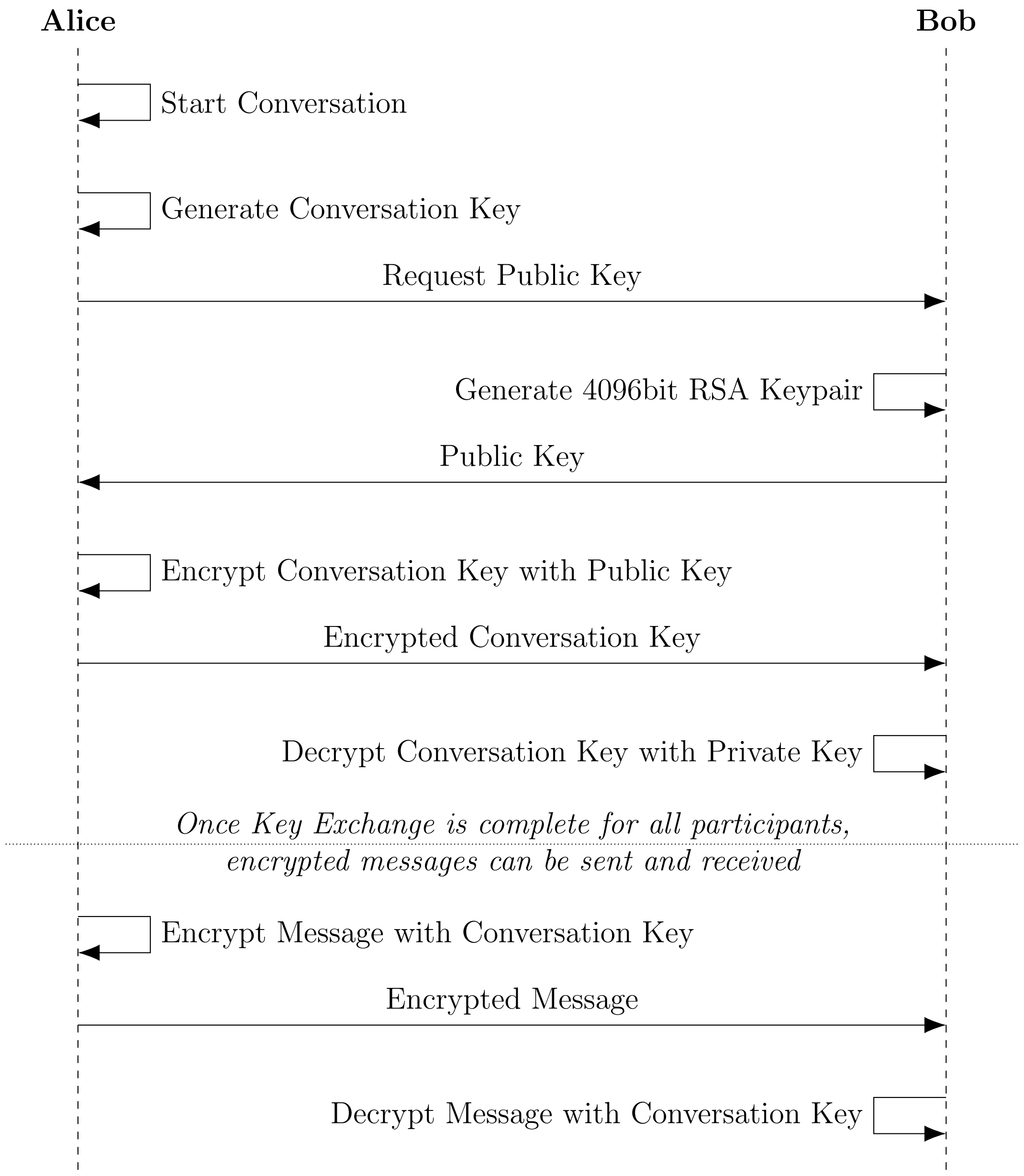

Alice
Bob
Start Conversation
Generate Conversation Key
Request Public Key
Generate 4096bit RSA Keypair
Public Key
Encrypt Conversation Key with Public Key
Encrypted Conversation Key
Decrypt Conversation Key with Private Key
Once Key Exchange is complete for all participants, encrypted messages can be sent and received
Encrypt Message with Conversation Key
Encrypted Message
Decrypt Message with Conversation Key

Figure 5.3: Sequence Diagram showing key exchange process for end-to-end encryption

5.3.3 Key Storage

It is essential that the symmetric Conversation Keys are kept confidential in order to ensure the security of the entire system. Therefore, thought must be given to the secure storage of these keys on client devices. The keys need to be stored on the users' devices to enable encryption and decryption however it is important that they are not stored in plaintext as they could be read by malicious actors and used to decrypt messages without the users knowledge. It was therefore decided that the keys should be encrypted, using a password based encryption key. A design decision was made that the most effective method of storing the keys securely would be to utilise the native credential manager of the operating system. On MacOS this is 'Keychain', on Windows it is 'Credential Vault', and on Linux, 'libsecret' is used. These credential managers handle the secure encryption and decryption of keys using the system password, and reduces the risk of implementation mistakes. In order to access these services through Node.js, the 'node-keytar' library was used [46]. Keytar provides a consistent JavaScript API to the native binaries required to access the OS level keychains for reading and writing of secrets. Keytar usses a key-value pair model for organising the secret keys, and so for this implementation, the 'conversationId' is used as the key to each pair, with the value being the corresponding conversation key.

5.4 User Interface

As discussed in Section 5.1, React is being used as the Frontend JavaScript framework for building the User Interface designed in Section 4.3. React allows for the development of reusable UI elements, called 'Components'. The user interface implementation in its current state can be seen in Figures 5.4 and 5.5. It can be seen from these figures that there are some differences between the original designs and the actual implementation. The most notable changes, and the reasons for them, are as follows:

- The login screen is implemented with a number of tabs that allow the user to select from common email providers instead of filling in mail server information manually. This is as a result of user testing, discussed further in Section 6.5.1, which showed the original design to be confusing.
- Due to time constraints, and certain tasks taking longer than expected as discussed further in Section 7.1, the attachment sending feature of the application was not implemented. Therefore, the attachment button from the original designs is not present in the current implementation.

- Messages are no longer adjacent to a profile picture for the sender, but rather a different coloured bar is used to represent each sender in a conversation. The reason for this change is that without the attachment sending mechanism being implemented on the service as mentioned above, there is no way to transport the profile pictures between participants. The first version of the application simply excluded the profile pictures, however during user testing, it was mentioned that it was difficult to identify at a glance who had sent which messages within a conversation. Therefore, the coloured indicators were introduced as an alternative.

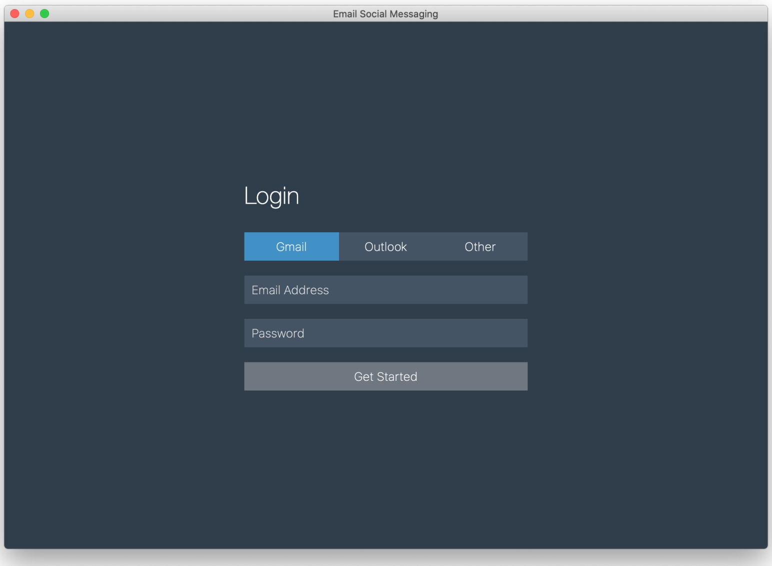
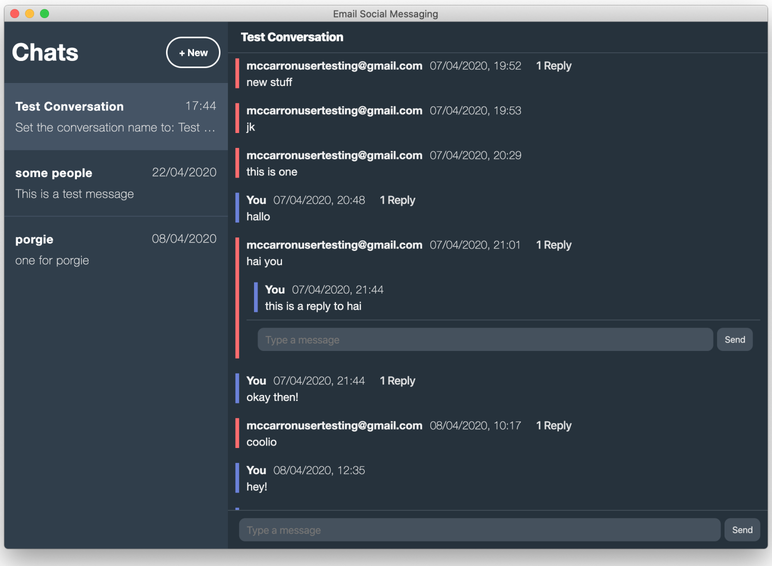


Figure 5.4: Login screen UI implementation



Test Conversation 17:44
Set the conversation name to: Test …

some people 22/04/2020
This is a test message

porgie 08/04/2020
one for porgie

Test Conversation

mccarronusertesting@gmail.com 07/04/2020, 19:52 1 Reply
new stuff

mccarronusertesting@gmail.com 07/04/2020, 19:53
jk

mccarronusertesting@gmail.com 07/04/2020, 20:29
this is one

You 07/04/2020, 20:48 1 Reply
hallo

mccarronusertesting@gmail.com 07/04/2020, 21:01 1 Reply
hai you

You 07/04/2020, 21:44
this is a reply to hai

You 07/04/2020, 21:44 1 Reply
okay then!

mccarronusertesting@gmail.com 08/04/2020, 10:17 1 Reply
coolio

You 08/04/2020, 12:35
hey!

Figure 5.5: Main conversation screen UI implementation

5.4.1 React Component Structure

One of the first tasks in building the User Interface was to break the design down into its constituent Components. React has a powerful composition model, and as such, some Components are constructed as a composition of other Components. An example of this model being implemented in the project can be seen with the `ConversationArea` Component. This component is created through the composition of three smaller components: `ChatHeader`, `ConversationViewer`, and `Composer`.

The application's user interface is logically organised into two different screens, the Login screen and the Main screen, as per the initial designs in Figure 4.1. Therefore, it was decided that these two screens should be distinct entities at the implementation level. The two screens are implemented as high-level components, which are navigated between through the use of the React Router library [47]; the Login page redirecting to the Main page wehn a login is successful. The full extent of the composition of components in the application is shown by the component composition trees in Figures 5.6 and 5.7.

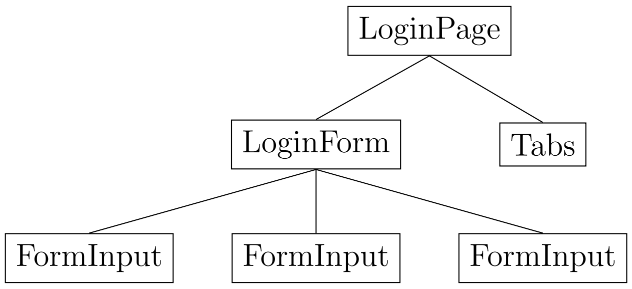


Figure 5.6: Component composition tree for the Login Page

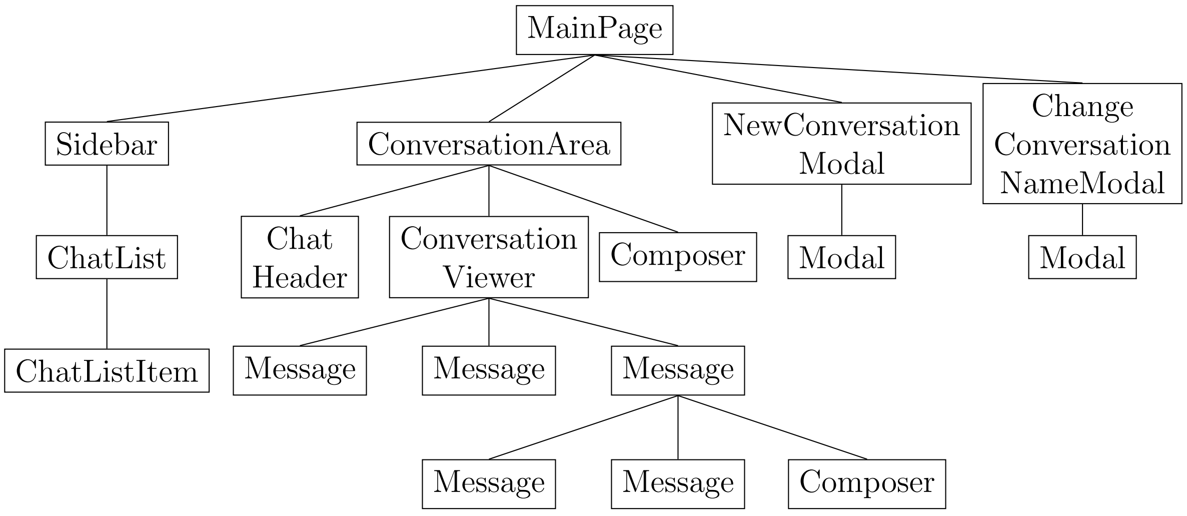


Figure 5.7: Component composition tree for the Main Page

5.4.2 Redux

Redux has been used for application-level state management; that is, any state that is shared between and used by many different UI components in the application. The Redux model consists of a single Store, which contains the application state. The state is immutable, and the only way to update it is to emit an 'Action', which describes what has happened on the UI, and can contain an optional data payload. In turn, 'Reducers' are used to specify how the current state should be transformed, depending on which action is emitted. Reducers are pure functions which take the previous state and an action as parameters, and return the new state which is used to update the store [48]. The core concept of the Redux data flow is illustrated by Figure 5.8 [49].

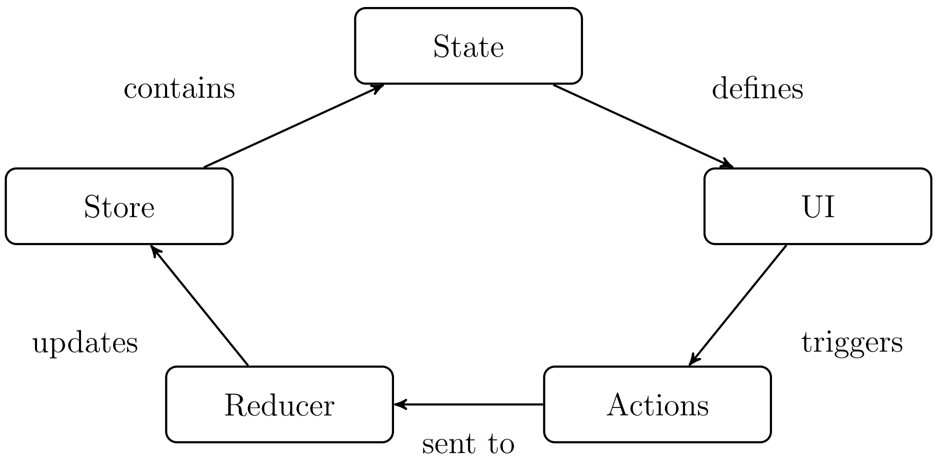


Figure 5.8: The Redux Flow

The use of Redux in this project provides a number of benefits. Firstly, since the key elements of Redux, actions and reducers, are just plain JavaScript objects and pure functions, respectively, unit testing becomes incredibly easy to perform without the need for complex mocking. Secondly, Redux removes the need for the React concept of 'lifting state up', whereby state is kept in a high-level component, and is passed down to child components through `props`. React code using this model becomes increasingly difficult to maintain, with state kept in different locations, and many components simply passing their props to their children without using them. Furthermore, since the state can only be updated by emitting an action, all changes to the state are centralised and happen one by one in a strict order; meaning that there are no race conditions to be aware of, the state is always predictable, and debugging is simplified [50].

To aid in software maintenance, the Redux reducer is split into separate functions, each of which is responsible for handling a separate 'slice' of state, an example of functional decomposition. These reducers are then passed to the `combineReducers()` function.

The two separate slices of state handled by reducers in this application are 'conversation' and 'user'. The user reducer handles information about the currently logged in user, such as their email address; handling actions such as `USER_LOGIN`. The conversation reducer is responsible for transforming the state regarding the conversations that are loaded from the mail server, handling many actions including `SELECT_CONVERSATION`, `LOAD_CONVERSATIONS`, and `NEW_MESSAGE`.

5.4.3 Improving Perceived Performance

When building the user interface, it was important to ensure that users' perceived the application as being fast and responsive, since this was one of the non-functional requirements. Activities such as sending emails, and fetching emails from IMAP mailboxes take some time, and by carefully considering user interface elements, the perceived time for these actions to be completed can be minimised.

Initially, the focus was placed on enhancing the user experience during fetching of messages from the mail server. It is suggested that loading spinners lead to a better perception of loading times than providing no animated feedback whatsoever [51]. Therefore a decision was made to implement loading spinners whilst waiting for the conversation info to be fetched immediately after login, and whilst waiting for messages to be fetched when a conversation is selected. Two animated loading indicators were implemented using the `react-spinners` library [52], and these will ensure that users are aware that application is working in the background to load content and should increase perceived performance.

These loading spinners were present in the version of the application used in the first stage of User Acceptance Testing, and given the feedback from this testing, it would seem that they were largely effective. However, as is discussed in more detail in Section 6.5.1, many users found that the process of sending messages felt slow and unresponsive; to the extent that, in some cases, users sent messages multiple times because they thought that they had not initially been sent correctly. The reason for this poor user experience was due to the process that was used for adding sent messages to the user interface, which is illustrated in Figure 5.9.

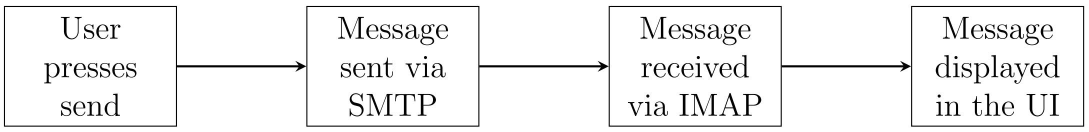


Figure 5.9: Flowchart illustrating the message sending process

In order to improve this user experience, changes were made to the message sending process. Under the modified process, as shown in Figure 5.10, messages will be displayed in the UI as soon as they are sent, but with an additional ‘Sending’ indicator. Once the message has been sent and then subsequently fetched from the mailbox, the stage at which the message would be shown under the original system, the sending indicator is removed.

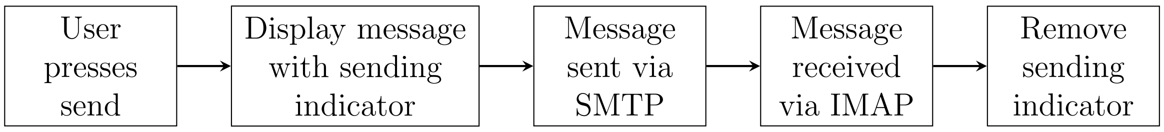

User presses send
Display message with sending indicator
Message sent via SMTP
Message received via IMAP
Remove sending indicator

Figure 5.10: Flowchart illustrating the message sending process

5.5 Service

Aside from the user interface, the other core element of the application is the backend service, which is implemented as an Electron renderer process, but does not display a UI window. The service has two primary roles; firstly, to handle the sending of all email messages as required, and secondly, to handle the fetching of incoming emails from the mail server. These two sets of functionality are implemented independently of each other, and are orchestrated through a `MailManager` object. A `MailManager` object is instantiated when a user submits their login credentials, and is responsible for creating instances of the `ImapClient` and `SmtpClient` classes, which are discussed below in further detail. The `MailManager` exposes methods which are called in response to IPC events from the frontend. The `MailManager` is also responsible for registering event listeners to the `ImapClient` object contained within it, to act upon events that are emitted, performing the necessary tasks in response.

5.5.1 Sending Email

Emails are sent through the Simple Mail Transfer Protocol (SMTP) [53] as this is the recognised standard for sending of email and is supported by all email providers. It was decided early in the implementation process that an existing library should be used to handle email sending, to avoid the complex and time-consuming task of building a client that implements the SMTP protocol from scratch based on the specification in RFC 5321 [53]. The SMTP client is responsible for establishing a two-way transmission channel with a SMTP server, and transferring mail messages to that server for it to handle.

Research revealed two existing and maintained JavaScript libraries for sending email messages with SMTP. These are Nodemailer [54] and smtp-client [55]. The smtp-client library works at a relatively low-level, requiring each SMTP command to be manually invoked and sent to the server. It was decided that this low-level of control was not required for this project, and that the higher-level abstraction provided by Nodemailer would be more suitable.

To maintain a good level of separation of concerns within the code and to simplify access to library functions, all access to the nodemailer library is carried out through a dedicated class, `SmtpClient`. This class handles the creation of a Nodemailer `Transporter` instance, verification of account credentials, and sending of messages. It should be noted, however, that this class is not responsible for the construction of messages in-line with the schema defined for this application, to ensure that it is versatile and reusable. The generation of messages in the correct format is handled by separate parts of the codebase, and is discussed in Section 5.5.3.

5.5.2 Receiving Email

There are two protocols currently in use for accessing messages from mailboxes, IMAP [56] and POP3 [57]. The key difference between these two protocols is that when a client reads messages from the server using POP3, it downloads the messages, stores them locally, and removes them from the server. Using the IMAP protocol, messages are stored permanently on the server and are not removed unless unless explicitly deleted by the client. Based on the architecture of this application, whereby all messages are stored on the server, and fetched only when a conversation is loaded, it was immediately obvious that IMAP should be used for mailbox access in this project.

Preliminary research indicated two suitable libraries to simplify the use of IMAP in JavaScript. The first of these is node-imap [58] which allows programmatic access to an IMAP server, but works at a relatively low-level, requiring manual invocation of the commands used in the IMAP protocol. On the other hand, imap-simple [59] is a wrapper around the node-imap library which provides a simplified programming interface. Due to the simplified interface, it was initially decided that imap-simple should be used in this project to speed up development time. As development of the software progressed, it became apparent that, since imap-simple "is missing a great deal of functionality from node-imap" [59], it was not going to be possible to use it in this application. Specifically, the problem with imap-simple was related to being able to use the `onmail` event listener that is invoked when new messages arrive in the mailbox. As a result, the underlying IMAP communications in this application are performed using node-imap.

Similar to the sending of email messages; a class, `ImapClient`, has been implemented to ensure that interactions with the node-imap library are contained to one specific place in the code, to increase maintainability. It is important to note that the node-imap library makes extensive use of the Node.js event-driven programming model, emitting events when there is new data to respond to. The primary role of the `ImapClient` class is to register listeners to the various events that are emitted by node-imap and ensure that the correct actions are performed in response. The class also provides methods which can be called to register new event listeners at runtime, based on specific events.

The node-imap library provides two key methods, `search()` and `fetch()`, which are used together to find and then download specific subsets of messages from the mail server. These methods allow for conditions to be applied to queries of the mail server, such as 'unseen messages only' or 'only messages with this subject'. After initiating a fetch, the messages found are emitted from the node-imap instance via events. In turn, these messages are transformed and collated by the ImapClient, before being emitted as events from the ImapClient itself, which is a subclass of `EventEmitter`, to be handled by the MailManager.

IMAP inherently has a concept of 'SEEN' messages; a flag is set on the server for each message to denote whether a the message has been seen by the user or not [56]. When developing the methods to allow querying of the mailbox, it was important to consider under which circumstances messages should be marked as seen on the server. When messages are first received or are fetched when opening the application, their 'SEEN' flag should not be modified. This ensures that messages which are marked as unread in the UI remain unread even if the application is closed and reopened. An exception to this is if a new message is received in the conversation that is currently open in the UI. In this case it is assumed that the message will have been read by the user and should therefore be marked as 'SEEN'. Messages are also marked as 'SEEN' when they are fetched as the result of a conversation being opened.

Currently, the ImapClient class is also responsible for decrypting messages once they have been fetched from the server. Whilst this allowed for rapid development of a minimum viable product, ideally; under the principle of Single Responsibility of the SOLID design principles[60], the code for encryption and decryption should be in a separate module which can then be imported and used wherever necessary.

5.5.3 Message Schema

Implementation of the message schemas designed in Section 4.2.1 is via a number of generation functions defined in `threader.js`. These functions are exported from the module and utilised within the `MailManager`. Messages are created by using the data passed to the generate functions as values in JavaScript objects and these objects are then converted into JSON (JavaScript Object Notation) [61] strings.

Although the message schema designed in Section 4.2.1 is ultimately used for organising and displaying messages within the application, it is not used directly in the emails that get sent by the software. The reason for this is due to the chosen implementation of message encryption. Instead, ordinary messages are sent containing a message body with the following schema:

```
subject: EMAILSOCIALMESSAGING:{conversationId}
body:    {
           conversationId,
           message,
           iv,
           authTag
         }
```

In this model, `conversationId` is used to identify the conversation that the message belongs to. `message` contains the JSON representation of the message object defined previously in the design stage; encrypted, and represented using Base64 encoding [62]. `iv` is a Base64 encoded representation of the initialisation vector used to encrypt the message, and `authTag` is a Base64 encoded representation of the Galois Message Authentication Tag. The entire object is then converted to a JSON representation to be placed in the email body. When a message is received, the `message` component is decrypted using the Conversation Key, and supplied initialisation vector, with the authentication tags compared to ensure integrity.

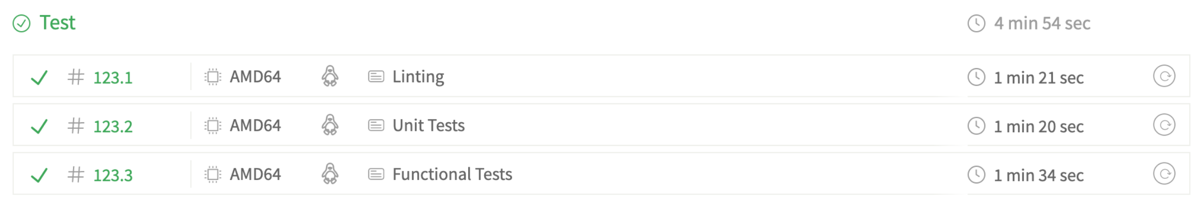
As identified in the design stage, messages and conversations should both have unique IDs. This has been achieved through the use of Universally Unique Identifiers (UUIDs), specifically UUIDv4 [63]. It was decided to use version 4 of UUID since it is based solely on random numbers, and since a namespace is not required as is the case in v3, it allowed for the simplest implementation.

It should be noted that these generation functions also generate the subject lines for messages, which include a tag identifying the messages as being either ordinary messages, or messages sent as one of the various stages of the key exchange process, as well as the message ID in the case of ordinary messages. The reasoning for putting the message ID in the subject of messages is to allow messages to be queried on the mail server quickly, without having to parse the entire message body.

6. Quality Assurance & Evaluation

6.1 Continuous Integration

Throughout the project, the use of Continuous Integration has been a vital element of the quality assurance process. Travis CI [64] was chosen as the Continuous Integration service to be used, and a CI pipeline was developed at the start of the development process to maintain good code quality. The stages of this pipeline include running a linter, unit tests, and functional tests, and can be seen in Figure 6.1 The CI pipeline runs whenever a Pull Request (PR) is opened on the project's GitHub repository from a feature branch, and all stages of the pipeline must pass before the PR can be merged into the master branch. Using features in GitHub, the master branch was protected. This means that the ability to push any commits directly to the master branch is disabled, to ensure that the PR process, including the CI pipeline is adhered to for every commit.



Test 4 min 54 sec

✓	# 123.1	AMD64	Linting	1 min 21 sec
✓	# 123.2	AMD64	Unit Tests	1 min 20 sec
✓	# 123.3	AMD64	Functional Tests	1 min 34 sec

Figure 6.1: Continuous Integration pipeline stages on Travis CI

6.2 Code Style

As part of the quality assurance process, it is important that code is written in a consistent style across all areas of the project. The code style for this project is based on the AirBnB JavaScript styleguide [65], with some modifications to match styles preferred by the Prettier code formatter, in addition to some custom linting rules specific to this project. The style rules are enforced by ESLint, an open-source linter for JavaScript . The full ESLint configuration can be found in the `.eslintrc.json` file in the source repository. By keeping consistent code style, maintenance of the project becomes simplified, and code remains clear to read and understand. It should also be noted, that in the interests of readability and maintainability, the source code is documented inline with comments explaining the purpose of key sections of code.

6.3 Unit Testing

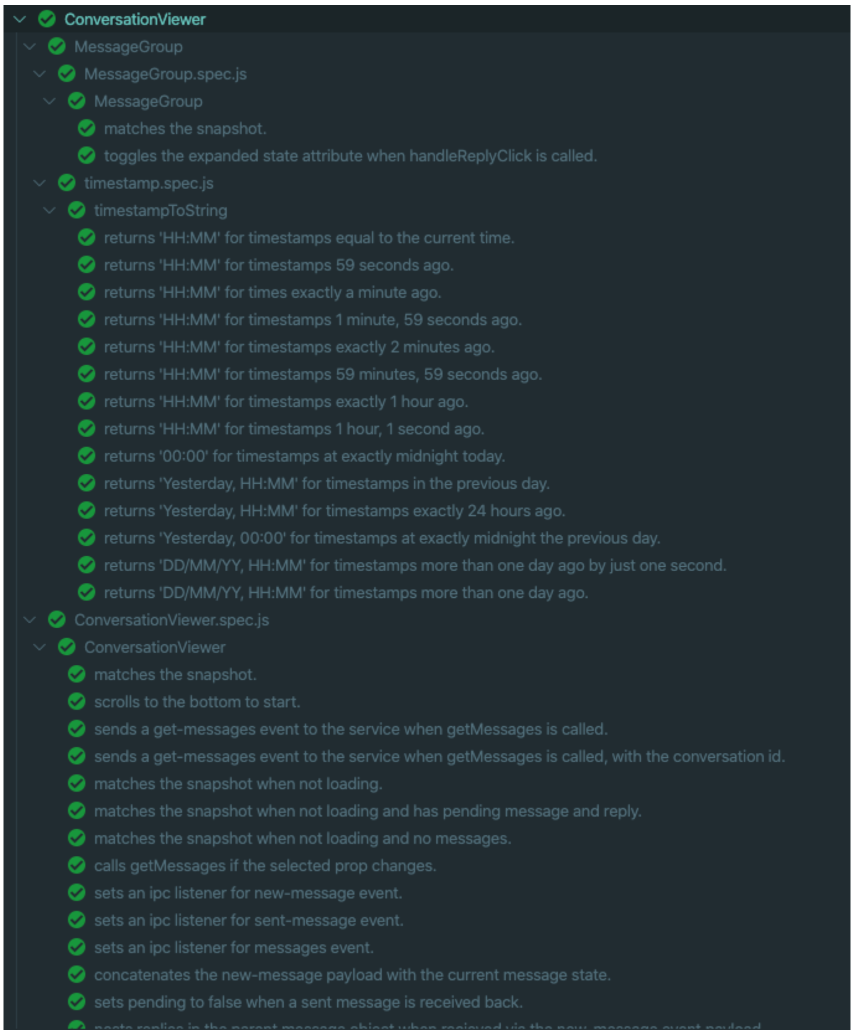
To ensure that the code meets functional requirements and operates as expected, unit tests were written alongside new code throughout the development process following the XP concept of Test-Driven development [19]. By writing unit tests at the same time as the code, errors can be identified and resolved quickly, resulting in a development process that is faster overall.

The unit tests have been written using Jest, an open source test framework for JavaScript. The primary reason why Jest was chosen as the testing framework for this project was due to its Snapshot Testing feature [66], allowing tests to take a snapshot of a React component's output and then compare all future test runs to this snapshot. This leads to much cleaner test code, which can be written and maintained more easily. In addition, the tests for React components use the Enzyme library for manipulating, traversing, and simulating interaction events on the component output [67].

In total, around 200 unit tests were written for this project, focusing on the most important elements of functionality. Line coverage of the unit tests currently sits at 85%, with all tests passing, which increases confidence that the tests are able to reliably identify problems. An example of some of the tests that have been written can be seen in Figure 6.2. A priority of any future work on this project should be to attempt to increase unit test coverage, specifically in the service side of the application, to further increase confidence in the functionality of the code.

6.4 Functional Testing

In addition to the unit tests described previously, it was intended that this project would make use of automated Functional (End-to-End) tests in order to ensure that the functional requirements of the system are met, as outlined in Section 3.1.1. It was decided that Spectron, the de facto tool for automating functional tests for Electron applications [68], would be used to aid in this. However, owing to the complexity of the application architecture and build pipeline, some problems were encountered whilst setting up this tooling that took a considerable amount time to resolve. Not wanting these issues to negatively impact overall progress, development began alongside working on resolving the issues with Spectron. Whilst the problems were eventually resolved, with a simple test being written as a proof of concept, at this stage a considerable amount of implementation



```
✓ ConversationViewer
  ✓ MessageGroup
    ✓ MessageGroup.spec.js
      ✓ MessageGroup
          ✓ matches the snapshot.
          ✓ toggles the expanded state attribute when handleReplyClick is called.
    ✓ timestamp.spec.js
      ✓ timestampToString
          ✓ returns 'HH:MM' for timestamps equal to the current time.
          ✓ returns 'HH:MM' for timestamps 59 seconds ago.
          ✓ returns 'HH:MM' for times exactly a minute ago.
          ✓ returns 'HH:MM' for timestamps 1 minute, 59 seconds ago.
          ✓ returns 'HH:MM' for timestamps exactly 2 minutes ago.
          ✓ returns 'HH:MM' for timestamps 59 minutes, 59 seconds ago.
          ✓ returns 'HH:MM' for timestamps exactly 1 hour ago.
          ✓ returns 'HH:MM' for timestamps 1 hour, 1 second ago.
          ✓ returns '00:00' for timestamps at exactly midnight today.
          ✓ returns 'Yesterday, HH:MM' for timestamps in the previous day.
          ✓ returns 'Yesterday, HH:MM' for timestamps exactly 24 hours ago.
          ✓ returns 'Yesterday, 00:00' for timestamps at exactly midnight the previous day.
          ✓ returns 'DD/MM/YY, HH:MM' for timestamps more than one day ago by just one second.
          ✓ returns 'DD/MM/YY, HH:MM' for timestamps more than one day ago.
  ✓ ConversationViewer.spec.js
    ✓ ConversationViewer
        ✓ matches the snapshot.
        ✓ scrolls to the bottom to start.
        ✓ sends a get-messages event to the service when getMessages is called.
        ✓ sends a get-messages event to the service when getMessages is called, with the conversation id.
        ✓ matches the snapshot when not loading.
        ✓ matches the snapshot when not loading and has pending message and reply.
        ✓ matches the snapshot when not loading and no messages.
        ✓ calls getMessages if the selected prop changes.
        ✓ sets an ipc listener for new-message event.
        ✓ sets an ipc listener for sent-message event.
        ✓ sets an ipc listener for messages event.
        ✓ concatenates the new-message payload with the current message state.
        ✓ sets pending to false when a sent message is received back.
        ✓ nests replies in the parent message object when recieved via the new-message event payload
```

Figure 6.2: Example of some of the unit tests written for the software

work had been completed, with functional testing being carried out manually. The decision was made that the complex work of writing a backlog of automated functional tests would likely put the project behind schedule. Therefore, functional testing continued to be carried out manually after each feature implementation, with the intention of adding functional tests if time became available. Unfortunately this was not the case, and the writing of automated functional tests is one piece of work that should be considered prior to further work being carried out on the project.

6.5 User Acceptance Testing

In order to assess the quality of the software product, User Acceptance Testing was conducted, where a number of people were asked to use the software to perform a number of tasks whilst under observation, and then answer some questions about the experience. The User Acceptance Testing was planned to take place in two stages:

1. First group of users test the software as soon as it is a Minimum Viable Product.
2. Second group of users test the software at a later stage, once feedback from the first stage has been acted on.

The first stage of this plan was carried out as intended, and the outcomes discussed below, however the second stage of UAT could not be conducted due to the COVID-19 pandemic, which made carrying out observations of users infeasible. Had it taken place, the focus of this stage of UAT would have been to identify whether the issues that arose from the first stage had been sufficiently resolved, as well as testing of additional features. This change of plan really highlights the importance of the iterative cycles encourage by the Agile development methodology, where testing was not just reserved for the end of the project, and so at least some testing had taken place prior to the pandemic.

6.5.1 Stage 1

The key objectives of this stage of User Acceptance Testing were primarily to assess the extent to which some of the Non-Functional requirements had been met, which are more difficult to assess through other forms of testing since they are somewhat subjective, but also to ensure that the software works as expected in a real-world environment, including on different operating systems, as users tested the application on their own devices, which included both Windows and MacOS. This stage of UAT was carried out with four participants. The full set of instructions used during the testing process and the follow-up questions asked can be seen in Appendix 1.

Key Observations/Feedback

The user testing process resulted in a large amount of feedback being gathered. The full set of notes, taken during the testing sessions can be found in Appendix 2. The most important and recurring feedback has been summarised below.

- Confusion about the display name when creating a new conversation.
- Messages take a long time to appear in the chat after being sent.
- Some users struggled to find the ‘reply’ feature.
- Most users found changing the conversation name by clicking on the current name to be intuitive.

- All users struggled to find where the new direct replies could be found.
- Most users said that they would not have known how to find the login credentials for their own email account.
- Many users suggested that it should be easier to differentiate who the sender of messages is.
- Every user suggested that they would like to be able to send attachments.

Response to UAT

The results of this phase of user testing were extremely insightful in understanding both the areas where the application was succeeding as well as areas that required improvement. Following the iterative development cycles provided by the Agile methodology, the feedback from user testing was used to feed back directly into the development priorities for the next iteration of work.

It should also be noted that in Section 4.2, it was suggested that the mesasge threading model may need to be modified based on the feedback from participants in user testing. During user testing, none of the participants thought that the message threading model was unsatisfactory, and therefore it is assumed that no changes need to be made to this element of the project.

It was decided that addressing users' concerns with existing functionality should take priority over the implementation of brand new features. This is in line with the philosophy that it is much easier and faster to fix issues with software as early as possible, and that making these changes at a later stage in the development process could lead to more work being required unnecessarily. Based on this, a list of feedback-based development priorities was created to direct further development work which were all subsequently completed. The priorities are outlined below:

- Improve the login process so that common email providers can be used without the need for mail server settings.
- Display sent messages in the chat immediately, with a sending indicator until the transport process has completed.
- Make new messages more obvious.
- Add coloured indicators next to messages to indicate which user sent them.

7. Reflections & Summary

7.1 Project Management

As outlined in Section 3.2, the project was managed using the Kanban Agile methodology [69]. The Kanban methodology has allowed for the flexibility of being able to take any card from the project backlog at any time. This has been extremely useful, and has allowed for changes to be made to the project plan in response to challenges and changing plans following further research. Although an Agile way of working was adopted, the rough timeline for the key tasks in the project was established as part of the initial planning, and is shown in Figure 7.1.

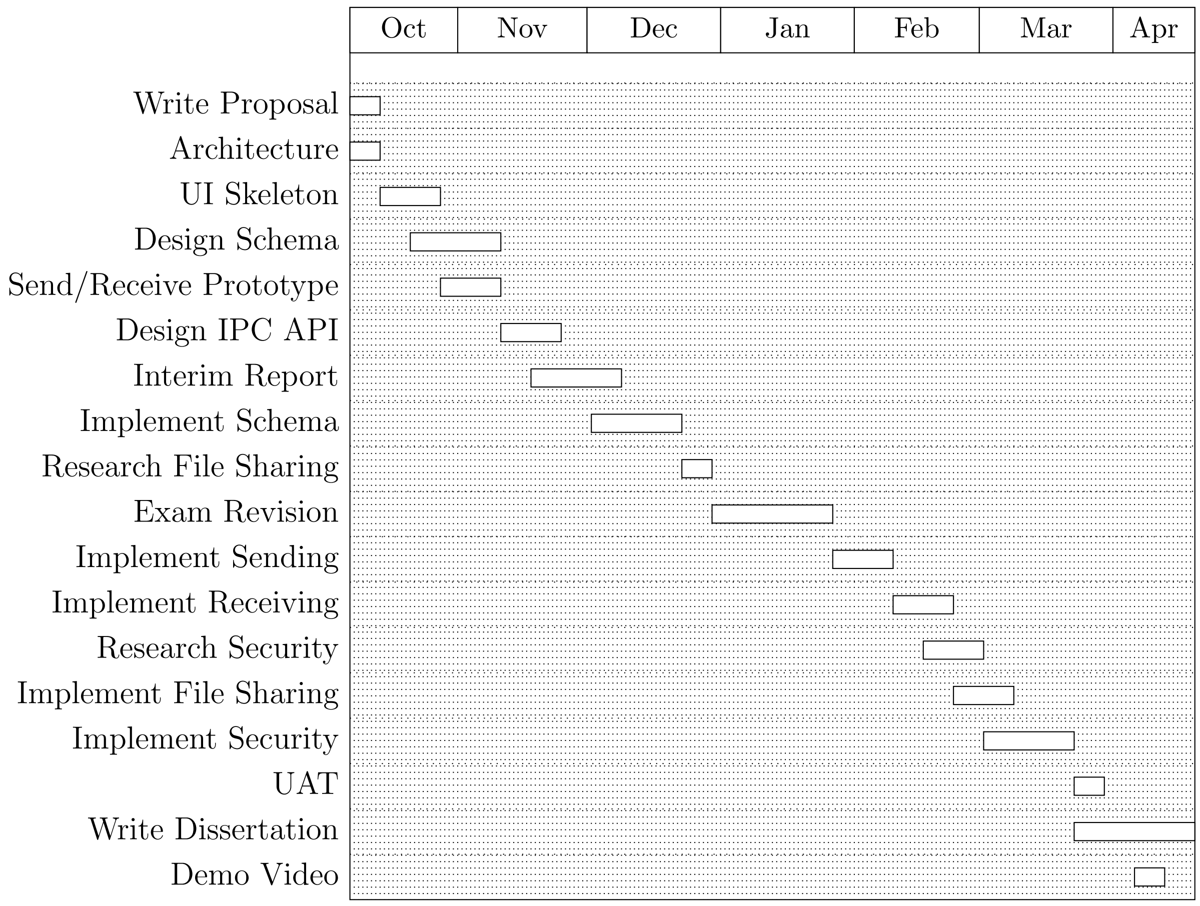


Figure 7.1: Project Gantt Chart

Despite best efforts to accurately estimate the amount of time that would be necessary to complete certain tasks, due to lack of experience, some of these were underestimated which meant that, due to time limitations, it was necessary to remove some features from the project scope to ensure that those features which are implemented are completed to a high quality. The Gantt chart served well identifying when tasks were overrunning and could potentially impact the timely delivery of the project as a whole. As the project began falling behind schedule, with tasks such as prototyping the message sending and receiving process taking considerably longer than expected due to the complexitites of the node-imap library, it was decided that file sharing functionality should be excluded from the project scope. This decision was made so that a more essential feature, end-to-end encryption, could be implemented fully and to a high standard, and this was completed on schedule.

Throughout the course of the project, progress was tracked using an online Kanban board on Trello. The benefit of this board is that it was possible to generate a burnup chart, seen in Figure 7.2, showing the number of work items (cards) completed over time.

Work over the course of the project was relatively consistent, with some time taken for exam revision from December through January. The COVID-19 situation towards the end of the project presented some disruption and uncertainty, however this was primarily during the report writing stage and therefore did not significantly affect progress in terms of development.

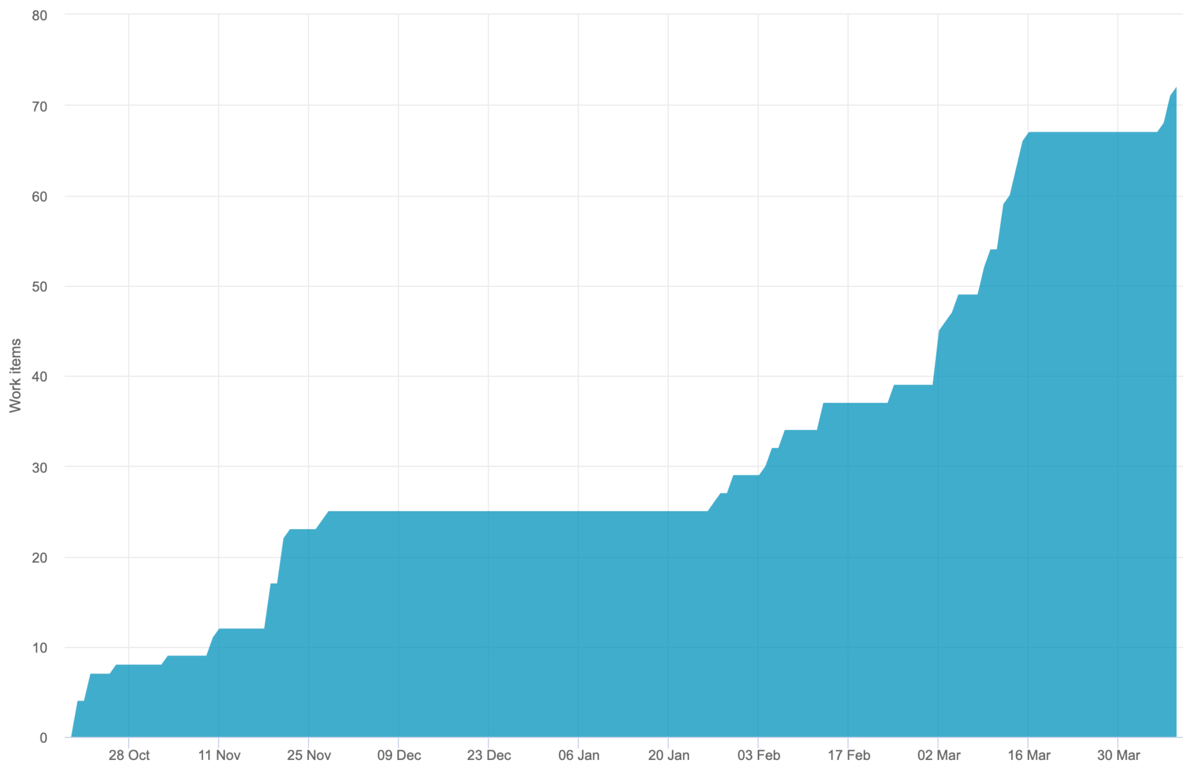


Figure 7.2: Burnup chart showing number of completed work items over time

7.2 Contributions and Reflections

Overall, I believe that this project has been a success, with high quality, well tested code, culminating in a software product that meets the vast majority of the initial requirements. Writing unit tests often took a considerable amount of time, and this was not factored into the time estimates for task completion, which did lead to changes to the project plan, however changes were made successfully and did not cause major detriment to the project as a whole. The project presented a number of challenges, including developing a user interface that supports an excellent user experience as well as working with complex communication protocols such as IMAP. Designing and implementing effective solutions to these has stretched my abilities as a software engineer and enabled me to explore areas that I had not previously had the opportunity to work in.

7.3 Future Work

Despite the significant work that has taken place so far, there is some further work necessary before this software could become a commercially viable product, and there are additional features that could be added. The immediate focus for future work should be to implement the features that were removed from the project scope for various reasons. Following this, further improvements can be made. In addition, the software would benefit from improvements to the current unit and functional testing coverage, to ensure that all further developments are sufficiently regression tested. Some of the key items of work to be completed in the future are; to allow users to send attachments, as was originally planned, which would help to increase adoption of this platform as a new messaging client. In addition; to improve the application's performance, message fetching should be paginated, so that only a subset of a conversation's messages are fetched to begin with (the most recent), and more messages are fetched as the user scrolls back.

Bibliography

[1] Christian Montag, Konrad Błaszkiewicz, Rayna Sariyska, Bernd Lachmann, Ionut Andone, Boris Trendafilov, Mark Eibes, and Alexander Markowetz. Smartphone usage in the 21st century: who is active on WhatsApp? *BMC research notes*, 8(1):331, 2015.

[2] Karen Church and Rodrigo De Oliveira. What's up with whatsapp?: comparing mobile instant messaging behaviors with traditional SMS. In *Proceedings of the 15th international conference on Human-computer interaction with mobile devices and services*, pages 352–361. ACM, 2013.

[3] Messenger Secret Conversations Technical Whitepaper. Technical report, Facebook, May 2017.

[4] Carole Cadwalladr and Emma Graham-Harrison. Revealed: 50 million facebook profiles harvested for cambridge analytica in major data breach. *The guardian*, 17:22, 2018.

[5] WhatsApp Encryption Overview Technical Whitepaper. Technical report, WhatsApp, December 2017.

[6] Steve Sheng, Levi Broderick, Colleen Alison Koranda, and Jeremy J Hyland. Why johnny still can't encrypt: evaluating the usability of email encryption software. In *Symposium On Usable Privacy and Security*, pages 3–4. ACM, 2006.

[7] Email statistics report 2019-2023. Technical report, Radicati Group, February 2019.

[8] Mike Hanson, Dan Mills, and Ben Adida. Federated Browser-Based Identity using Email Addresses. 2011.

[9] Steven L Rohall, Dan Gruen, Paul Moody, Martin Wattenberg, Mia Stern, Bernard Kerr, Bob Stachel, Kushal Dave, Robert Armes, and Eric Wilcox. ReMail: a reinvented email prototype. In *CHI'04 extended abstracts on Human factors in computing systems*, pages 791–792. ACM, 2004.

[10] Gina Danielle Venolia and Carman Neustaedter. Understanding sequence and reply relationships within email conversations: a mixed-model visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 361–368. ACM, 2003.

[11] Whatsapp. `https://whatsapp.com`. Online. Accessed 28-04-2020.

[12] Facebook Messenger. `https://messenger.com`. Online. Accessed 28-04-2020.

[13] Slack. `https://slack.com`. Online. Accessed 28-04-2020.

[14] Signal. `https://signal.org`. Online. Accessed 28-04-2020.

[15] Bernard Kerr and Eric Wilcox. Designing remail: reinventing the email client through innovation and integration. In *CHI'04 Extended Abstracts on Human Factors in Computing Systems*, pages 837–852. ACM, 2004.

[16] Bernard Kerr. Thread arcs: An email thread visualization. In *IEEE Symposium on Information Visualization 2003 (IEEE Cat. No. 03TH8714)*, pages 211–218. IEEE, 2003.

[17] Martin Fowler, Jim Highsmith, et al. The agile manifesto. *Software Development*, 9(8):28–35, 2001.

[18] David J Anderson. *Kanban: successful evolutionary change for your technology business*. Blue Hole Press, 2010.

[19] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[20] S Balaji and M Sundararajan Murugaiyan. Waterfall vs. v-model vs. agile: A comparative study on sdlc. *International Journal of Information Technology and Business Management*, 2(1):26–30, 2012.

[21] AWS. `https://aws.amazon.com`. Online. Accessed 29-04-2020.

[22] Google Cloud. `https://cloud.google.com`. Online. Accessed 29-04-2020.

[23] Microsoft Azure. `https://azure.microsoft.com`. Online. Accessed 29-04-2020.

[24] Electron. `https://electronjs.org`. Online. Accessed 30-04-2020.

[25] David D Lewis and Kimberly A Knowles. Threading electronic mail: A preliminary study. *Information processing & management*, 33(2):209–217, 1997.

[26] Jacob Palme. Message threading in e-mail software. Technical report, Citeseer, 1998.

[27] Hubert Florin. Threads in Slack, a long design journey. `https://slack.design/threads-in-slack-a-long-design-journey-a7c3f410ecb4`, 2018. Online. Accessed 28-11-2019.

[28] TypeScript. `https://typescriptlang.org`. Online. Accessed 30-04-2020.

[29] React. `https://reactjs.org`. Online. Accessed 27-04-2020.

[30] Vue.js. `https://vuejs.org`. Online. Accessed 27-04-2020.

[31] Angular. `https://angular.io`. Online. Accessed 27-04-2020.

[32] Stackoverflow Developer Survey Results 2019. `https://insights.stackoverflow.com/survey/2019`. Online. Accessed 01-05-2020.

[33] Redux. `https://reduxjs.org`. Online. Accessed 01-05-2020.

[34] Babel. `https://babeljs.io`. Online. Accessed 01-05-2020.

[35] Webpack. `https://webpack.js.org`. Online. Accessed 01-05-2020.

[36] electron-builder. `https://electron.build`. Online. Accessed 01-05-2020.

[37] Electron API Documentation - Application Architecture, v6.0. `https://github.com/electron/electron/blob/6-0-x/docs/tutorial/application-architecture.md`. Online. Accessed 30-11-2019.

[38] Electron API Documentation - Performance. `https://github.com/electron/electron/blob/master/docs/tutorial/performance.md`. Online. Accessed 30-11-2019.

[39] The Chromium Projects: Inter-process Communication (IPC). `https://www.chromium.org/developers/design-documents/inter-process-communication`. Online. Accessed 25-04-2020.

[40] Events — Node.js v14.0.0 Documentation. `https://nodejs.org/api/events.html`. Online. Accessed 25-04-2020.

[41] React.Component - React. `https://reactjs.org/docs/react-component.html`. Online. Accessed 25-04-2020.

[42] State and Lifecycle - React. `https://reactjs.org/docs/state-and-lifecycle.html`. Online. Accessed 25-04-2020.

[43] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[44] Joppe W Bos, Onur Özen, and Martijn Stam. Efficient hashing using the aes instruction set. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 507–522. Springer, 2011.

[45] David McGrew and John Viega. The galois/counter mode of operation (gcm). *submission to NIST Modes of Operation Process*, 20, 2004.

[46] keytar. `https://atom.github.io/node-keytar`. Online. Accessed 01-05-2020.

[47] React Router: Declarative Routing for React.js. `https://reacttraining.com/react-router/web`. Online. Accessed 25-04-2020.

[48] Redux Core Concepts. `https://redux.js.org/introduction/core-concepts`. Online. Accessed 01-05-2020.

[49] Lessons Learned Implementing Redux on Android. `https://hackernoon.com/lessons-learned-implementing-redux-on-android-cba1bed40c41`. Online. Accessed 01-05-2020.

[50] Redux Three Principles. `https://redux.js.org/introduction/three-principles`. Online. Accessed 01-05-2020.

[51] Samantha Persson. Improving perceived performance of loading screens through animation, 2019.

[52] React Spinners. `https://github.com/davidhu2000/react-spinners`. Online. Accessed 01-05-2020.

[53] J Klensin. RFC 5321 - Simple Mail Transfer Protocol. `http://tools.ietf.org/html/rfc5321`.

[54] Nodemailer Documentation. `https://github.com/nodemailer/nodemailer`. Online. Accessed 01-12-2019.

[55] smtp-client Documentation. `https://github.com/xpepermint/smtp-client`. Online. Accessed 01-12-2019.

[56] M Crispin. RFC 3501 - INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1. `http://tools.ietf.org/html/rfc3501`.

[57] J Myers and M Rose. RFC 1939 - Post Office Protocol - Version 3. `https://tools.ietf.org/html/rfc1939`.

[58] node-imap Documentation. `https://github.com/mscdex/node-imap`. Online. Accessed 01-12-2019.

[59] imap-simple Documentation. `https://github.com/chadxz/imap-simple`. Online. Accessed 01-12-2019.

[60] Robert C Martin. The Principles of OOD. `http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`. Online. Accessed 01-05-2020.

[61] T Bray. RFC 8259 - The JavaScript Object Notation (JSON) Data Interchange Format. `https://tools.ietf.org/html/rfc8259`.

[62] S Josefsson. RFC 3548 - The Base16, Base32, and Base64 Data Encodings. `https://tools.ietf.org/html/rfc3548`.

[63] P Leach, M Mealling, and R Salz. RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace. `http://tools.ietf.org/html/rfc3501`.

[64] Travis CI. `https://travis-ci.com`. Online. Accessed 01-05-2020.

[65] Airbnb JavaScript Style Guide. `https://airbnb.io/javascript`. Online. Accessed 01-05-2020.

[66] Jest Snapshot Testing, v24.9. `https://jestjs.io/docs/en/snapshot-testing`. Online. Accessed 01-12-2019.

[67] Enzyme Introduction. `https://airbnb.io/enzyme`. Online. Accessed 30-11-2019.

[68] Spectron. `https://www.electronjs.org/spectron`. Online. Accessed 01-05-2020.

[69] Andrew Stellman and Jennifer Greene. *Learning agile: Understanding scrum, XP, lean, and kanban.* " O'Reilly Media, Inc.", 2014.