NANYANG TECHNOLOGICAL UNIVERSITY

SCSE20-0057

Web interface for ASR transcriber

Submitted by: Nguyen Dinh Le Dan
Matriculation Number: U1720241A

Supervisor: Assoc Prof Chng Eng Siong
Examiner: Assoc Prof Pan, Sinno Jialin

Submitted in Partial Fulfillment of the Requirements for the Degree of Bachelor of Computer Science of the Nanyang Technological University

School of Computer Science and Engineering

2021

Abstract

With the rise of artificial intelligence and machine learning, speech-to-text (STT) technology has become more accurate and precise than ever. Since its conception, STT has proven to have a huge potential in many fields, one of them obviously being transcribing. Transcribing is an act of making a written copy of the content of an audio or a video. Even now, transcribing is still mostly a manual and laborious job, which can be extremely cost inefficient, time consuming and prone to human error. By leveraging the powerful ability of STT, one can significantly cut down on time loss and improve upon the accuracy and efficiency of the process. However, creating an STT model requires a substantial amount of data, both in terms of audio/video and transcription. The data then needs to be tracked, managed, and corrected accordingly so that it can be used to train the model. Current process if done manually has proven to be highly inefficient and time consuming. Thus, there is a need for a centralized interface that can both allow users to interact and manage data, as well as connect said data to the STT model. The Transcriptor was created as the solution to the problem. As a web application, the Transcriptor offers features like user management, upload/download transcriptions and built-in transcription editor. Initial testing shows that Transcriptor helps cut down on the time used to manage the data as well as increases productivity on manual correction of transcripts.

Acknowledgement

This Final Year Project was made possible thanks to the help and support of many people, who have not only helped me gain invaluable experience but also make this project a success. I am truly thankful to them.

First and foremost, I would like to express my gratitude to Prof Chng Eng Siong, who is not only my FYP supervisor but also for my Professional Internship. His advice and guidance played a huge role in my success in both, and although he is a very busy person, he always shows great patience whenever I stumble or make a mistake. I will always be grateful to him.

I also would like to thank Kyaw Zin Tun for his tremendous work in managing and cooperating multiple projects together. He also always supported me and gave valuable advice and suggestion throughout the project. Finally, I want to give a special thank to Mai Trung Duc, who are not in the team anymore, but has contributed a vital role in this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Since long ago there has always been a need for keeping official records of an event, whether it is a meeting, a hearing, or a court. With the rise of technology, nowadays it is standard to keep records using either video record, and/or audio record. Along with that, it is also sometimes required to have transcripts of said video/audio. Until now most of the transcribing has been done manually, with many services offer them on the market. Some examples include Rev [19] or Translationservice.sg [20]. These services not only add more cost to operating, but are also time-consuming, prone to human error, not to mention the risk of operational security when there is sensitive information on said video/audio.

1.2 Motivation & Problem

Recently, artificial intelligence, or specifically machine learning, has been used to conduct speech-to-text (STT) with great success. With the help of STT, we can automate the process of transcribing, converting hours of audio into text in an instant, saving time and cost. There are many services on the market offer speech-to-text with neigh real-time response and incredible accuracy like Google or Azure right now. However, it does not solve our problem of operational security. These services run on a 3rd party cloud provider, and we cannot guarantee the security of the information once uploaded. Additionally, there is a unique problem lurking in the process. Although the services mentioned above offer STT in many different languages supporting many different accents, namely Google who provides service for 200 languages and 13 English accents, none of them have a Singaporean English accent option. With Singaporean being a unique accent of English with many slangs, it can have a significant effect on the accuracy of the result. Thus, there

arises a need for a home trained STT model, catering for Singaporean accent. However, in order for an STT model to have a good accuracy, it requires a tremendous amount of data. Furthermore, the results of the model need to be tracked, managed, and corrected if necessary so it can be used to re-train. Currently, these are all being done manually, which is laborious, time-consuming and prone to error. Therefore, we need a centralized interface to manage authorization, upload audio/video for inputs, track said uploaded files, and allow for viewing and editing the results of the results if necessary.

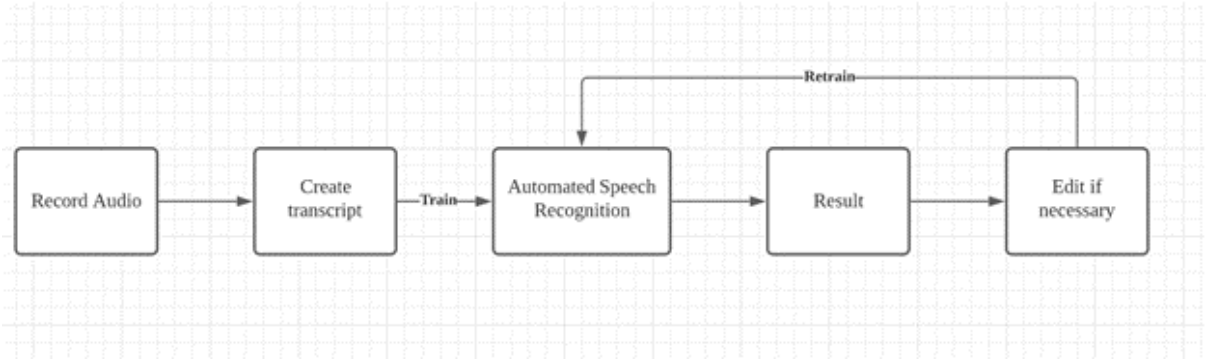

Figure 1.1: Flow of training an Automated Speech Recognition system

1.3 Scope

The scope of this project restricts to the web interface, and many methods and technologies used are only applicable to web developing. However, the idea and flow can be used for other types of application like native app or mobile app.

1.4 Report Organisation

This report comprises of 5 chapters, the overview of which can be summarized as follow:

- Chapter 1 provides an introduction to the project, any relevant background or motivation of the project, and the scope.
- Chapter 2 discusses relevant solutions to the problem that the project is trying to solve, their strengths and weaknesses if any.
- Chapter 3 gives an overview on what the project can offer in terms of ideas and concepts, as well as how it can achieve such ideas and concepts with regards to technology and implementation.
- Chapter 4 discusses the actual output and implementation of the project.

- Chapter 5 concludes the report with future directions, remaining work for this project if any, and lesson learnt from the project.

Chapter 2

Literature Review

2.1 Early version of Transcriptor

There was an early version for this project with the name Transcriptor, which relied heavily on a waveform library called Waveform-playlist [3]. The library specifically supports displaying and editing transcriptions, which is precisely the purpose of the application. However, since it is a highly specialized library, there is little to no room for customization. Meaning the library does not have the ability to customize the behavior of the waveform, the transcription list, or any of its functionalities. Moreover, implementing new features also proved to be a difficult task, namely multi-channel audio support. Not to mention, this library was written in pure JavaScript with no React interface. Therefore, the team needed to use jQuery to implement it, which defeated the purpose of using React as a framework in the first place, and inflated the project.

2.2 STT as a tool for transcribing

STT as a technology is not a new concept, but in fact has been studied and researched for a long time. In recent years, with the rise of AI and machine learning, the technology has become increasingly better, with really good accuracy and neigh real time result for a moderate amount of data. Services that offer STT APIs like Google, which supports real time cloud transcribing with supports for 125 languages and variants [14], or Microsoft Azure, which has roughly the same set of features and even allows for customizing the model [13], has been in used for a while and gained a lot of success. The rise in popularity of these services has since powered a lot of products. Transcribe by Wreally is a fairly well-known product with customers from NASA or Stanford [18], and besides traditional transcribing, it also has supports for dictation, which is when the audio is spoken directly into the microphone to produce the text, useful for when the audio is not clear. It also

has a moderately complex audio and text editor which allows users to manually transcribe from scratch. Even though its STT backend can detect many languages, there are difficulties when it comes to English with Singaporean accent. Furthermore, even though the editor is sufficient enough to create a transcript from scratch, the timestamps for sentences have to be written entirely manually, and there is no intuitive and efficient way to edit and modify an already transcribed file. TranscribeMe is another well-established transcribing service which leveraged the STT technology. However, unlike Wreally, it does not have a built-in editor, making it unable to support on site editing of transcripts.

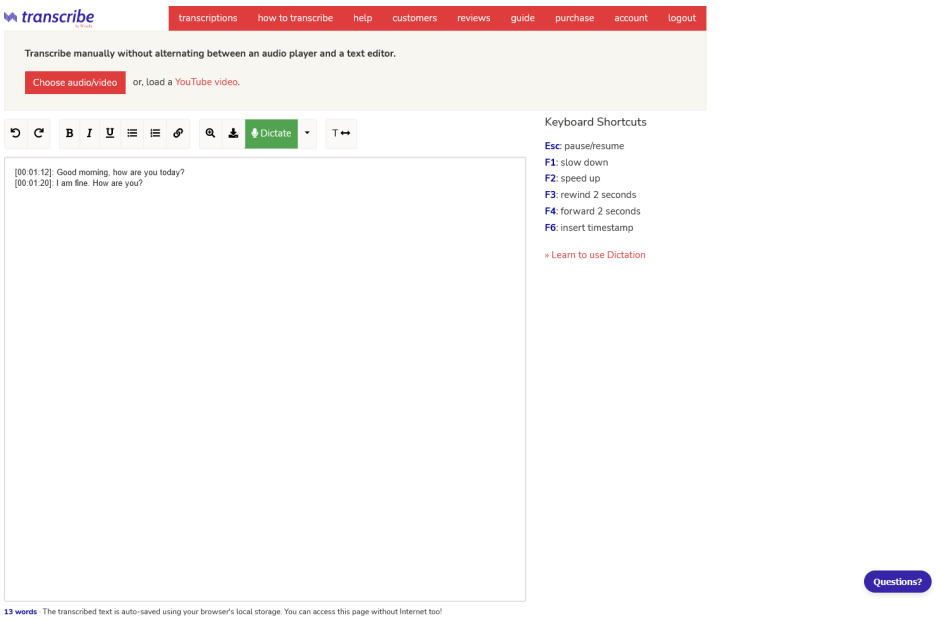

Figure 2.1: Wreally's editor, with support for dictate.

2.3 Transcript editor

Even though there are a lot of automated transcribing services, none of those have a built-in transcription management system with a complex and powerful built-in editor to support fast and efficient modifying of data. The closest comparison to a transcript editor available are subtitle editors. These allow users to assign and edit subtitles to videos. Each subtitle has a text, a start and end time. The challenge of these editors is how to make the creating and editing subtitles process seamless and efficient. SubPlayer is a subtitle editor, and it is entirely open-source [15]. It has a really unique design, which leverages on the presentation of the audio waveform and the positions of the subtitles on said waveform to enable users to interact with it and modify the subtitles timestamp more efficiently and accurately. KapWing [16] is an online video editor which has supports for adding subtitles. While not having a unique design like SubPlayer, its editor allows the user to pinpoint exactly the timestamp in the audio or video and set it to the subtitles

timestamp. One common point of these 2 examples is that the user does not have to manually set the timestamp for the subtitles, but rather they can do that either visually by a presentation on a waveform, or a button that accurately sets the timestamp to the current time. This not only allows the process of setting and viewing timestamp faster and more intuitive, but also prevents input errors when it comes to timestamp format and limit. While subtitles have similar attributes to transcripts in that they have a text content, start time and end time, these editors cannot act as a substitute for a transcript editor since they do not have support for audio files.

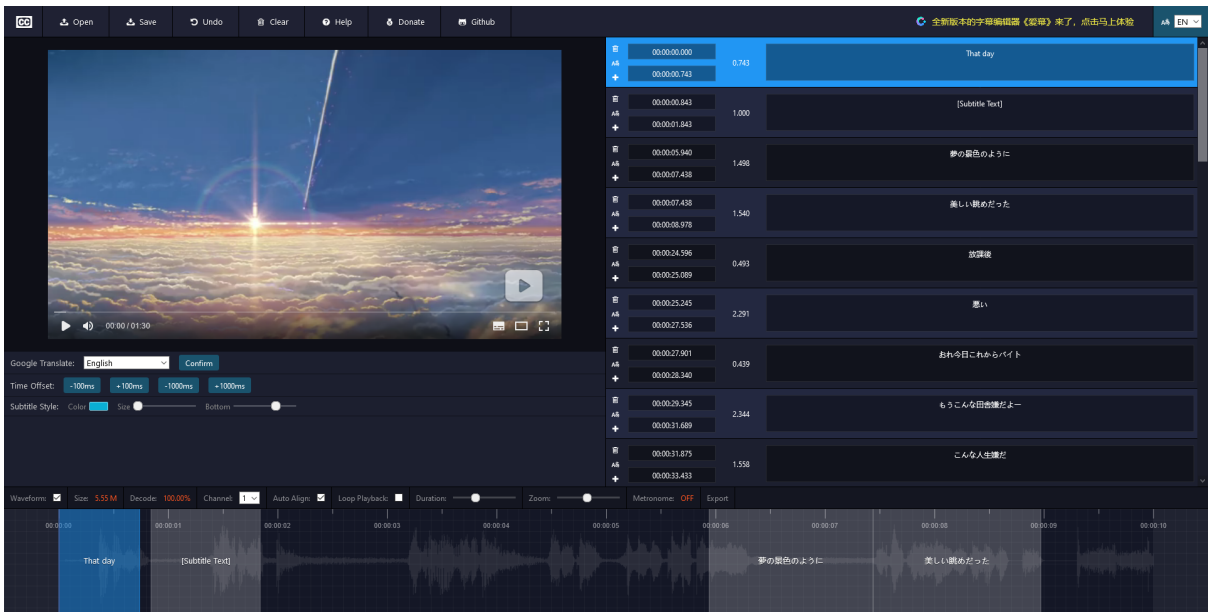

Figure 2.2: Subplayer's editor

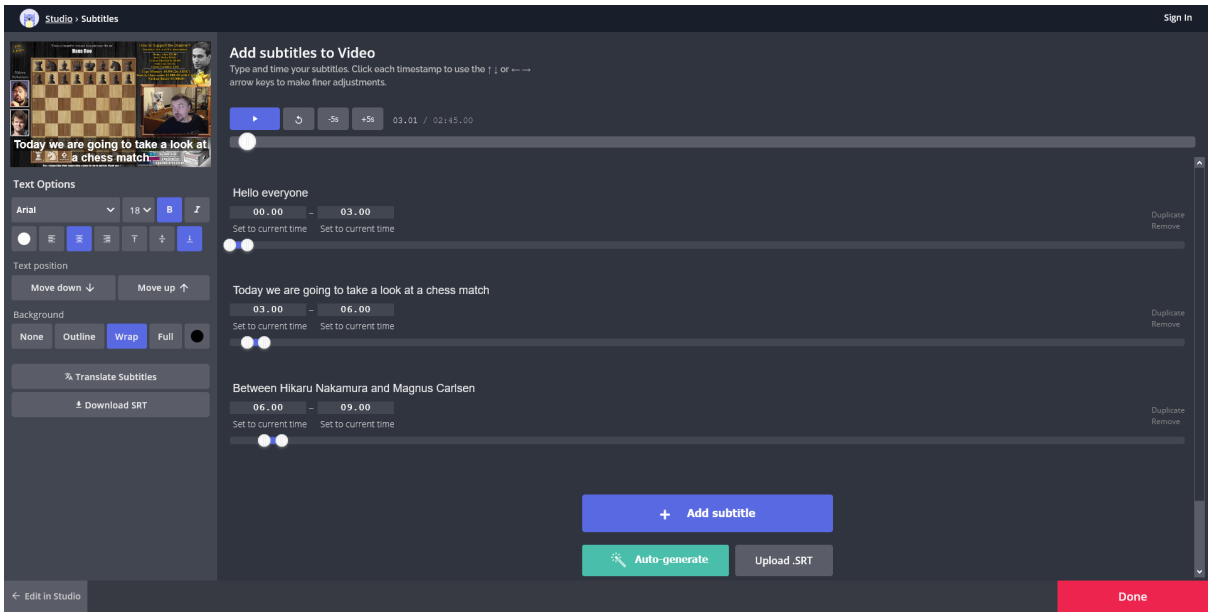

Figure 2.3: Kapwing's editor

2.4 Waveform Visualizer

A vital component of the transcription editor is the waveform visualizer to visualize the audio. Currently this can be achieved by drawing the content of the audio data into a graph-like waveform using the Canvas feature of HTML5. However, aside from correctly interpreting the audio data, the waveform needs to be interactive as well, and to implement that can take a lot of time. Waveform-playlist [3] not only draws waveform but also has a built-in editor for transcriptions. However, the library is not very robust and customizable. It cannot support drawing one channel of the audio, and only takes in 1 input as the form of Javascript File/Blob. The Wavesurfer [22] library is one of the most popular and well-supported waveform library, with support for many input types, well-versed and highly versatile interfaces, which allows for high customizability and maintainability.

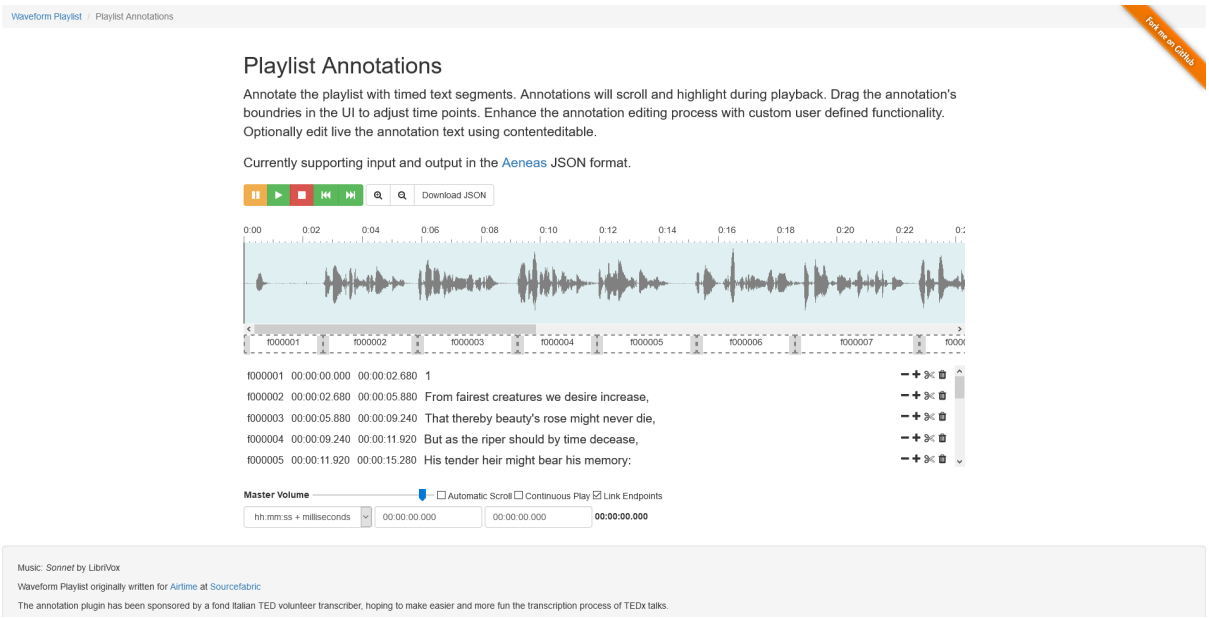

Figure 2.4: Waveform-playlist's default interface

Figure 2.5: Example of wavesurfer's implementation

2.5 Summary of past solutions

Name	Advantages	Disadvantages
Old version of Transcriptor	Meets all the requirements	Lack of customizability and upgradability due to library choice.
		Required jQuery support, inflating the project unnecessarily.
STT API service (e.g. Google, Microsoft Azure)	Good overall accuracy and response time.	Only an API, still needs to build frontend support.
	Simple implementation.	Requires money to use and unable to guarantee data privacy.
Transcribe by Wreally	Uses STT technology to transcribe.	Difficulty with Singaporean accent.
	Have an editor to create transcripts manually and dictate support.	Unintuitive editor design. Lack a straightforward way to upload and modify an existing transcript.
TransribeMe	Uses STT technology to transcribe.	No editor support.
SubPlayer	Creative design for the editor using the waveform, making editing timestamp intuitive.	
KapWing	Creative and useful approach in a button to allow setting timestamp to the current time.	No support for audio only.

Table 2.1: Advantages and disadvantages of past solutions

Chapter 3

Proposed Approach and Project Specification

3.1 Transcription's Components

A transcription in this project's context refers to a combination of an audio file and its corresponding transcript file. The audio file in can be in the form of any audio file, but in the Transcriptor's case, only MP3 and WAV format are allowed. The transcripts are represented using the TextGrid format. The TextGrid format is first used by Praat, an application that does phonetic analysis, and it is leveraged in this project as a mean to represent a transcript. Each TextGrid will have some general metadata of the audio like how long the audio is, and how many speakers are there. Then, each speaker will have an array of entries, each entry represent a sentence with start time, end time, along with the text content.

```
item [1]:
    class = "IntervalTier"
    name = "S2_bob"
    xmin = 84.93
    xmax = 1081.03
    intervals: size = 3
    intervals [1]:
        xmin = 84.93
        xmax = 86.81
        text = "wto school"
    intervals [2]:
        xmin = 1074.83
        xmax = 1076.92
        text = "yeah i heard from england"
    intervals [3]:
        xmin = 1080.00
        xmax = 1081.03
        text = "oh really"
item [2]:
    class = "IntervalTier"
    name = "S2_Alice"
    xmin = 7.21
    xmax = 2412.53
    intervals: size = 500
    intervals [1]:
        xmin = 7.21
        xmax = 8.16
        text = "hello"
    intervals [2]:
        xmin = 11.39
        xmax = 12.33
        text = "yes"
```

Figure 3.1: Sample Text Grid entries

Each transcription uploaded will be be stored in a backend database, the details of which will be discussed in section 4.1. Depending on whether the user provides a TextGrid file or not, the backend will optionally pass the audio file to a STT API service to process

and return a TextGrid file, which then will be stored and linked to said audio file.

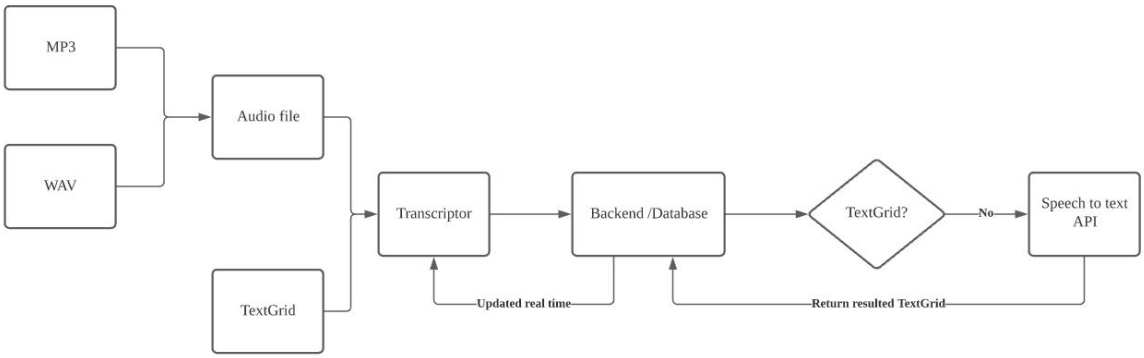

Figure 3.2: Transcriptor's input and output flow.

3.2 Project Methodology

Among the solutions presented in chapter 2, even though each one does have some of the features needed, there is no solution that meets every requirements.

The earlier version of Transcriptor, even though it was built with the requirements in mind, was limited by the libraries and framework it used. It could not satisfy some requirements, one of which was showing a multi-channel waveform, which shows the lack of customizability and maintainability of the old application. Moreover, the front end design proved to be unintuitive, and the performance was not good with stuttering being common occurrences due to the fact that the old waveform used to render all of the content at once. Therefore, a rebuild was needed in order to solve these problems and build a more stable and maintainable application.

As for other STT services' usage in transcribing, all the applications even though have good accuracy on the result, lacks a built-in text editor to edit the transcription if necessary. Additionally, a user management system was needed in order to digitally manage and distribute transcripts to different personnel to be more time efficient. Lastly, there is still a question of operational security as these are all 3rd party services, and data confidentiality cannot be guaranteed.

Transcriptor was built as an answer to all the missing requirements. The purpose of the Transcriptor is two-fold: to act as a data management service for the training of an ASR system, and an interface to interact with said system, which can be used by both the developing team and the system's users once it is deployed for general use.

The application was developed as a web interface, and allows users to manage and view

transcriptions that they have access to. It also comes with a robust and powerful editor for easy editing of transcripts. Along with that, Transcriptor also allows usesr to upload their own audio recording to the ASR service for transcribing. Transcriptor aims to solve all the problems that are mentioned above, while being easy to use and access by being hosted on web, rather than being a desktop app.

In our new version of Transcriptor, we improved the current user's management system existed in the old version of Transcriptor to allow for persistent login session, rebuilt the editor's graphical interface to be more intuitive and easier to navigate by taking inspirations from the subtitle editors mentioned in 2.3, and used a new waveform visualizer in Wavesurfer that is more robust and supported so that it we can have more flexibility in developing. In addition to that, Transcriptor was connected to a backend server hosted by our team so that to ensure data confidentiality.

3.3 Definitions and Assumptions

3.3.1 Definitions

For this project, the following terminologies are commonly used:

- Transcriptor: The name used for the application.
- Text Grid: A type of file specifically used for recording data for a transcript, which includes start time, end time, and content of each sentence.
- Transcription: A combination of an audio file and a transcript associated with it.
- Sentence: The start, end and content of a spoken sentence in the audio file, which corresponds with one entry in a Text Grid file.
- Transcriber: An automated system that produces a transcript of an audio file, consisting of many sentences from the audio.
- ASR: Short for Automated speech system, which automatically converts an audio piece into text.
- STT: Short for Speech to text, can be used interchangeably with ASR.
- Web framework: a software framework that is designed to supports a development of a web application by cutting down on common overhead processes when developing web applications while providing standard ways to build and deploy.
- Frontend: Refers to the website that runs on the client machine.
- Backend: Refers to the server and the project's database.

- API: Short for Application Programming Interface, which refers to the interface through which different applications can communicate.

3.3.2 Assumptions

We made the following assumptions when this project was started considering the circumstances, to both cut down on scope as well as speed up the development process.

- The only mean of accessing the app is through a computer, either a desktop computer or a laptop, and not a mobile or tablet. Since the main use of this application will be in an office setting, optimization and design responsiveness for mobile and tablet web viewing was not considered, which can take a long time and will not contribute much to the application considering the setting.
- The team using the application has access to another administrator application that manages different accounts and assigns transcripts. Without it, transcripts cannot be assigned and no data will show up on the application.
- User has stable internet connection while using the application. Even though most of the processes can be done client side, the application needs to continuously communicate with the backend to update the data, thus a stable internet connection is needed.
- User only accesses the application using Google Chrome browser or Mozilla Firefox. Other browsers most likely will work, but unexpected behaviours might occur since the application is tested mainly on the two aforementioned browsers due to their popularity.

3.4 Software Development Cycle

Given the dynamic development environment where the detailed requirements can be changed and added overtime, and the size of the small development team, the Agile framework was chosen for the project management. The agile approach in software management emphasizes on short but rapid development cycles, which allows the development team to break the requirements down into smaller tasks and deliver them in small size. This helps the small development team to not get overload on too big of a scope and enables them the opportunity to test each module and fix problems related to that module separately as opposed to testing the whole project which can be overwhelming. The short and rapid development cycles also enables a flexible and adaptive scope as the application can be updated frequently, thus making incorporating user's feedback easier. In fact, in a study in 2015, Pedro Serrador and Jeffrey K. Pinto have found that the more agile

approach a project take, the more successful the project was reported to be, based on stakeholder satisfaction and the efficiency of the project [12].

Each development cycle can be called a *sprint*, each sprint is usually 1-3 weeks long. For this project, we used a 1-week sprint. Each sprint will begin with user feedback evaluation or new features request. Then the requirements for that sprint are finalized, and the development can begin. After the development phase is finished, the application will then be tested and subsequently deployed when all the bugs are fixed. In order to easily manage and keep track of the status of each task, the development team used a *KANBAN board*. A KANBAN board is a visual representation of the tasks for each sprint, which are usually represent by cards. Each card will then go through different predetermined stages, representing the task current status. The stages can vary from team to team, but for us we use *TO DO*, *IN PROGRESS*, *DEVELOPMENT DONE*, *IN TESTING*, *READY FOR DEPLOYMENT*, and *DEPLOYED*. This not only helps the team manage and visualize their current tasks better, but also allows for better team work and cooperation between teammates. Roadblocks can be easily spotted and identified, so that the sprint won't be affected.

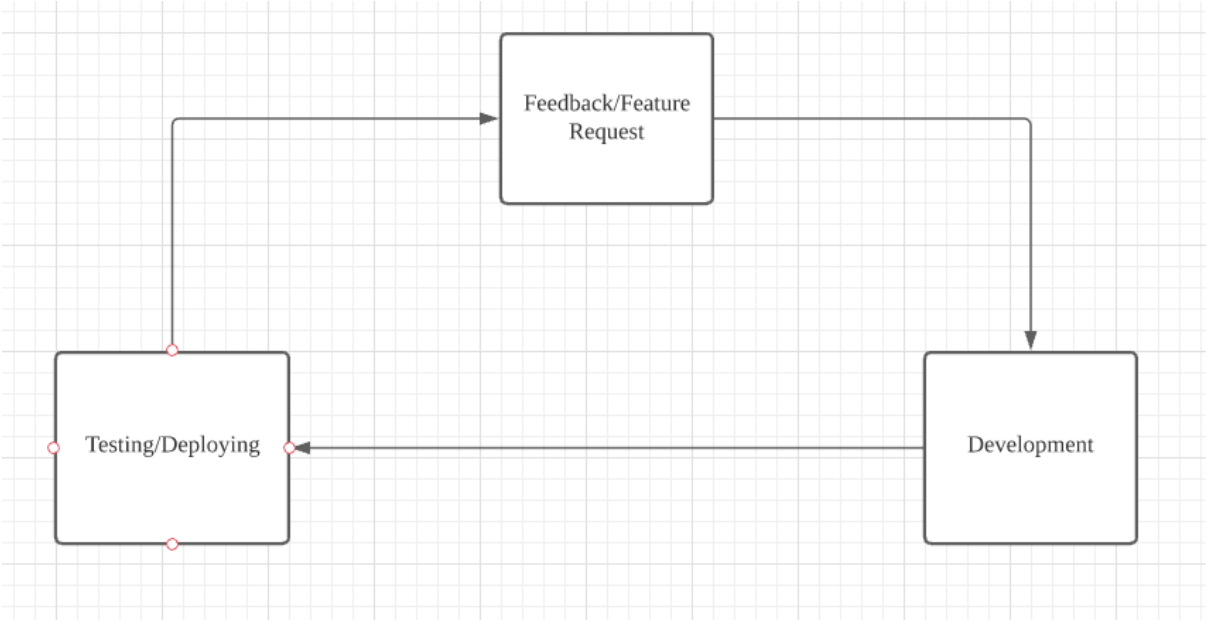

Figure 3.3: Agile development cycle used in this project

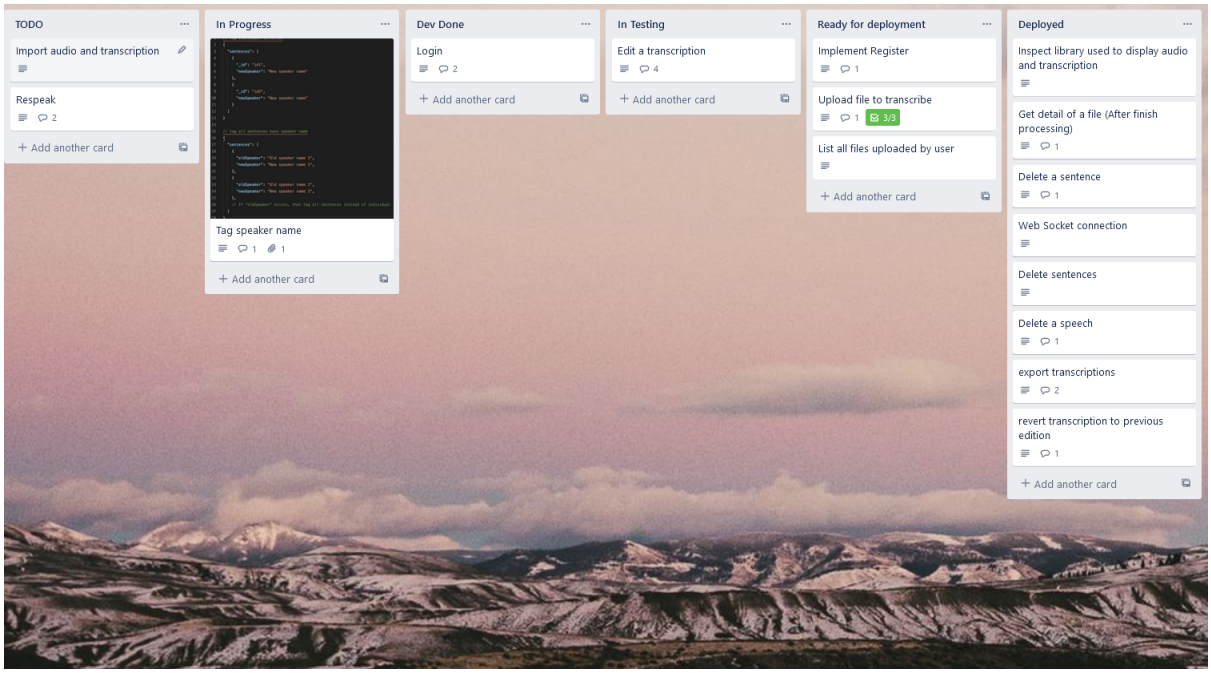

Figure 3.4: Sample KANBAN board used in this project

3.5 System Architecture

The system architecture used for this project is one employed by the Redux library, which is an improved take on the classic Model-View-Controller (MVC) architecture. The Redux architecture has been and being used by many web applications with great success, specifically ones that are written in React.

In a traditional MVC architecture, there would be 3 main components to the system: *Model*, *View*, and *Controller*. The view will display the data in the model, and the controller has the control to those data. The figure below best describes how the data flow in an MVC architecture.

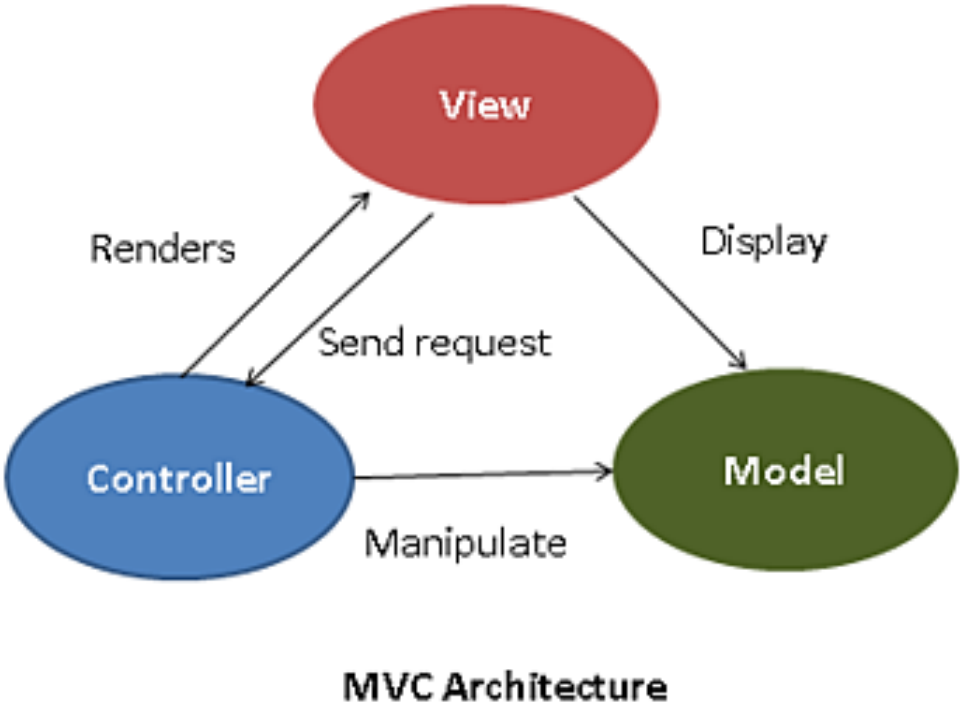

Figure 3.5: Data flow in an MVC architecture

As one can see, the data flow in MVC can be bidirectional. A view can issue a request to the controller, and a controller can request the view to render the data. This when used in a complex application like Transcriptor has proved to be difficult to debug and manage. On the contrary, the Redux architecture employs a strictly unidirectional data flow. Each application has one single *store* that has all the application's *states*. The UI will then render the states. If one wishes to make changes to the states, the UI will trigger an *action*, which then will be dispatched to a *reducer*, which will then create and replace the old store. The figure below will provide a better representation of the process.

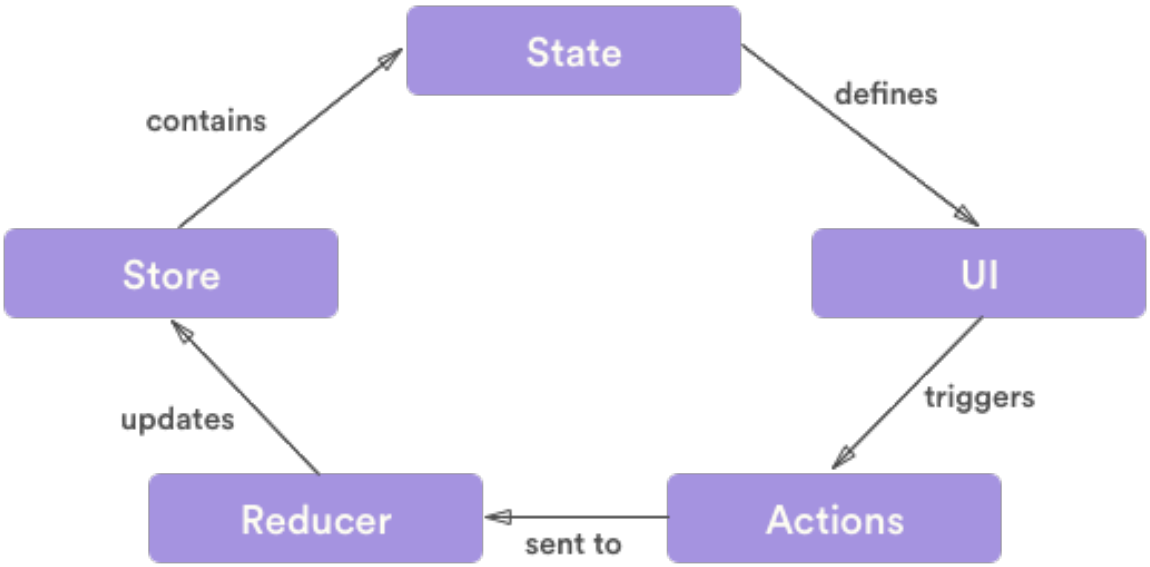

Figure 3.6: Data flow in the Redux architecture

As the data can only flow in one way, it is easier to pinpoint which component of a system is responsible for an error. This helps make the debugging process faster and more efficient, while still keeping the project relatively simple and straightforward [21].

3.6 Component Specification

3.6.1 Authentication

- Only allows user to access the dashboard if they have the appropriate credential.
- Allows user to sign up for an account using officially issued emails.
- Allows persistence log in without compromising user's credentials.

3.6.2 Upload

- Allows user to upload an audio file and optionally a Text Grid file. After uploading the transcription will be displayed in the transcription list.
- If no Text Grid is provided, the audio will be processed by the ASR. After completed the status and Text Grid will be updated in the list.

3.6.3 Transcription List

- Display all transcriptions that the user has access to based on their credentials, whether it is through uploading or getting assigned.
- Allows basic operations on each transcription like deleting, downloading, or open the editor with a specific transcription.

3.6.4 Editor

- Display and visualize the content of a transcription.
- Ability to playback the audio file.
- The start time, end time, content and speaker of each sentence can be edited in an intuitive and efficient way.
- Any changes will be automatically updated to the server, with support for a one-time rollback.

3.7 Tools and Technology

The below table shows the hosting environment setup for the Transcriptor

Cloud service	Microsoft Azure VM
Hardware	8GB of RAM, 2 vCPU cores
OS	Linux (Ubuntu server)
Runtime Environment	NodeJS 12.13.1
Disk size	30GB
Container	Docker
URL	https://transcriptor.speechlab.sg

Table 3.1: Hosting environment

This portion of the report will discuss about the technologies used to build the Transcriptor user interface. Along with usability, the technologies chosen also considered how fast and seamless they can be integrated, as well as how stable and supported they are, so that Transcriptor can be developed within a reasonable timeline without compromising on the features, while also has good maintainability down the line. With that in mind, following will be the list of core technologies that are used in the project, with an explanation on why they were chosen in the context of Transcriptor.

Component	Language/Library
Web Framework	ReactJS/Javascript
HTTP client	Axios/Javascript
State manager	Redux/Javascript
Waveform Virtualization	Wavesurfer/Javascript
Audio Player	HTMLAudioElement API/Javascript
Performance enhancement	React-virtualized/Javascript
CSS framework	Material-UI/Javascript

Table 3.2: Languages and Libraries used

3.7.1 Javascript

Javascript is a programming language mainly used for web development [6], and it is the only programming language used for developing this project. Since Transcriptor is a web application, Javascript is the only option since it is the only programming language for front-end web development. However, with rich framework support like React or Angular, along with new and robust APIs, this did not prove to be a problem but quite the contrary, as we were able to handle processes and calculations on client side that are traditionally handled by the server and later passed to the front end. This helps speed up the development process and make the project simpler as there is not as much dependency upon the backend.

3.7.2 React

React is an open-source frontend JavaScript library for building user interfaces and web applications [8]. It is maintained and developed by Facebook and is widely used and contributed by many developers. When developing a web application, choosing the correct framework is a very important step and usually decides the success of the project. Currently there are a lot of libraries that helps with building frontend web application, like Angular, Vue.js, Reactjs. The team chose Reactjs for this project, since it is one of the most popular and powerful JavaScript libraries for building web applications, with huge support for external libraries, making the development process as easy and fast as possible. The version of Reactjs being used is 16.12.0, with all the latest stable features of React.

3.7.3 Axios

Axios is a http client library used for sending and receiving http requests and responses with built-in protection for common http attacks like Cross Side Request Forgary [1]. Aside from the usual types of http request like *GET* or *POST*, the library also has a versatile interface that allows control to the payload's headers, authentication, parameters,

etc..., effectively making crafting complex request payload faster and easier. By using Axios, the team was allowed to focus more on the semantic side of the http communication without the need to focus on the technical implementation side, which increases the speed of the development process.

3.7.4 Redux

Redux is a state container for JavaScript Applications [11], and one of the most popular accompanied libraries to React. With a complex project like Transcriptor, it is important to have a state manager that is predictable, centralized and debuggable, which is exactly what Redux supports. All of Transcriptor's states can be managed in a centralized manner, meaning all states will be stored in a "store", making it the single source of truth, while any kind of state change needs to be dispatch to a "reducer", eliminating any kind of race conditions. Therefore, the application remains predictable, consistent and easy to test and debug. Redux helps developing and maintaining complex React apps like Transcriptor a lot easier and faster.

3.7.5 Wavesurfer

Wavesurfer is a customizable audio waveform visualization library written in JavaScript, which uses the browser's Web Audio API and HTML5 Canvas [22], allowing it to work natively on browsers that supports HTML5 without having to install external programs like Flash. Wavesurfer is the most popular waveform visualization library currently, with nearly 40 000 installs on NPM weekly [9]. Since the Transcriptor will involve working with audio a lot, having a robust and well-supported waveform visualization library is very necessary. Unlike many other libraries, Wavesurfer supports taking in input in the form of decoded AudioBuffer, along with the standard File/Blob input. This is especially important for the support of multi-channel waveform, which was one of the early problems with this project. Additionally, Wavesurfer also has support for interacting with waveform, API for playing audio, as well as support for external plug-ins like timeline and drag and drop, which also plays a vital part in the Transcriptor usability.

3.7.6 HTMLAudioElement

HTMLAudioElement is a Javascript interface which allows access to the HTML5 `<audio>` element, and the methods to manipulate them [5]. Before the appearance of HTML5, audio and video player has always relied heavily on Flash. However, with HTML5 comes native API support for video and audio player. HTMLAudioElement is one such API, which has supports for a wide range of audio types. It is also easy to use and configure, provides powerful and flexible interface to the audio player, such as setting current time or

volume or playback speed. It is native to browsers thus no third party program needed when using the Transcriptor, and lastly it ensures the Transcriptor to be future-proof unlike Flash which has its support removed at the end of 2020.

3.7.7 React-virtualized

React-virtualized is a react library for creating react elements that can efficiently render a large list or table of data [2]. While building any kind of application, performance is one of the most important yet difficult problems to solve. While developing Transcriptor, a performance issue was noticed when the list of transcripts was too long and the web page has to render too many components. This will lead to an unresponsive and stuttering interface, and overall a bad user experience. React-virtualized is an attempt to solve the problem of rendering a large list of items in React applications by creating and rendering only the components that are currently shown on screen, while swapping out and removing unseen components. This optimization improves the performance of the application by a huge margin. React-virtualized is a powerful react library aims to solve the performance problem for rendering a large list in React applications, with many powerful prebuilt components like List or Table. It is also one of the most used libraries for this purpose, with more than 678 000 downloads weekly on NPM [10].

3.7.8 Material UI

Material UI is a react library for building both beautiful and versatile react elements [7]. Along with a rich and powerful features list, Transcriptor needs to also have an aesthetically pleasing design while maintaining responsiveness. With that in mind, Material-UI was chosen to handle the aesthetic look of Transcriptor. Material-UI is library specifically for pre-designed and powerful React component, which allows the team to shift focus on the functionality of the application without spending too much time on the aesthetic look, making building and designing the project with React much faster and easier. Material-UI was built based on the Material design system created by Google [4], which helps the Transcriptor to have a modern and high-quality look, while also obeying golden design rules that was set Google to ensure a good frontend application.

Chapter 4

Implementation

4.1 Backend Communication

Even though the backend server is not in the scope of this project, in order to fully discuss the Transcriptor's implementation, we need to discuss the communication protocol between it and the Transcriptor. The backend is built using the NestJS framework, and is made up of a database consisting of transcription data and user's information, different API endpoints for authentication, getting transcription list, etc..., and a socket connection for real-time communication purposes.

4.1.1 Database

All of Transcriptor's data is stored using MongoDB, which is a NoSQL document-based database [17]. There are 3 main documents in the Transcriptor database: *users*, *transcription*, and *sentences*. The users document stores all of the user's information, including the username, display name, password hash, along with the list of allowed transcriptions. The list of transcriptions corresponds with the transcription document, where all the transcription data is stored. That includes the audio file and the list of sentences belongs to the transcript. The actual sentences data is stored in the sentences document, like start time, end time and text content.

The Transcriptor will not have direct access to the database, rather it is abstract by a layer of endpoints provided by the Nest server. Depending on the type of endpoint and its arguments, the database will be queried accordingly by the server and the response will be sent back to the Transcriptor.

4.1.2 API endpoints

The backend server has multiple API endpoints with different purposes, ranging from authentication to transcriptions editing. The API endpoints can be separated into 2 categories, public and private endpoints. The public endpoints can be successfully called without providing a token, and those include the login and register endpoints. The private endpoints are authorized using the Bearer Authentication scheme of the HTTP protocol, and the token used is the JWT token that is returned from the login and register call. The private endpoints are all the remaining endpoints except login and register, and failure to provide the bearer token or providing an expired token when calling them will result in a `403: Forbidden` response code.

Authentication

The backend has one endpoint for login and and one for registering. The data will be passed through the body of a POST HTTP request to ensure confidentiality. After successfully authorizing the user name and password, the server will respond with a JWT token, and that token is necessary for all the other calls that require authorization.

Requesting transcription List

The transcription list is server-side paginated, and based on the user's choice from the interface, the endpoint needs to be provided with the page number, along with the number of items per page. By using the bearer token, the server can figure out which user is requesting, and upon getting the list of authorized transcription, the server will do the pagination and return the paginated data through the response.

The server also has an endpoint for transcription download request. The audio file along with the TextGrid file will be zipped together server-side. After finishing the zipping process, the client will be notified and the content can be downloaded.

Uploading a transcription

In order to upload a transcription, the request requires an audio file in the form of a *blob*, and an optional TextGrid file. When called, the server will use the provided bearer token to figure out the user, and add the new transcription to the allowed list. If no TextGrid is provided, the server will further call the ASR backend to process the audio file. Upon completion, the new TextGrid file will be added to the transcription, and the updated status will be sent to the client using a socket connection, ensuring that the status can be updated in real time on the client.

Editing sentences

The backend has a range of endpoints to facilitate the editing process in the frontend, from creating a new sentence, editing a sentence to deleting a sentence from the database. Each sentence has its own ID separate from the transcription, therefore the request only needs to provided the sentence ID. Upon receiving a request, the server will make the necessary queries to the database, and respond with the new data to let the client know that the queries are successful.

4.2 Authentication

Accounts in the Transcriptor are used to manage access to the Transcriptor dashboard, as well as access control to different transcripts. This ensures that only authorized users are allowed to access the Transcriptor, while giving admins the ability to manage access to different transcripts and assign them to the correct individual. The user's account will be kept track using a JSON Web Token (JWT), which is used to authenticate the request made to the backend. User's login session will be maintained using this token, which is stored using the localStorage API in the web client. This allows the application to authenticate the user automatically if they signed in recently, enhancing the user experience. However, the token will be invalidated after 30 days without any activities to ensure security. The token will also be removed and invalidated after the user has manually signed out.

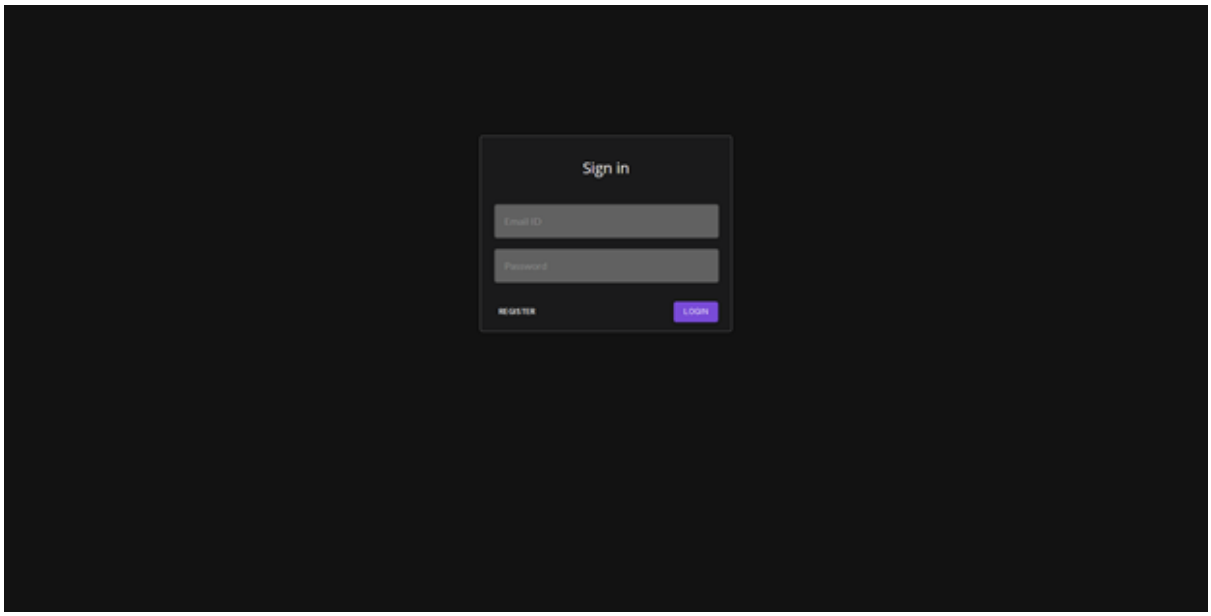

Figure 4.1: Sign in screen of Transcriptor

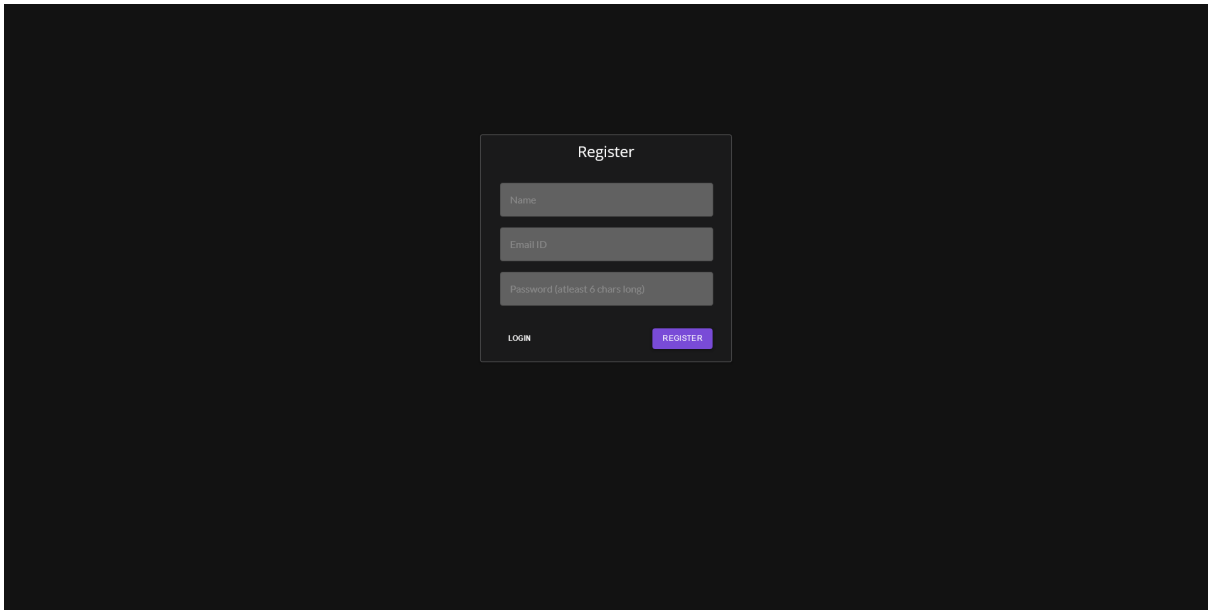

Figure 4.2: Sign up screen of Transcriptor

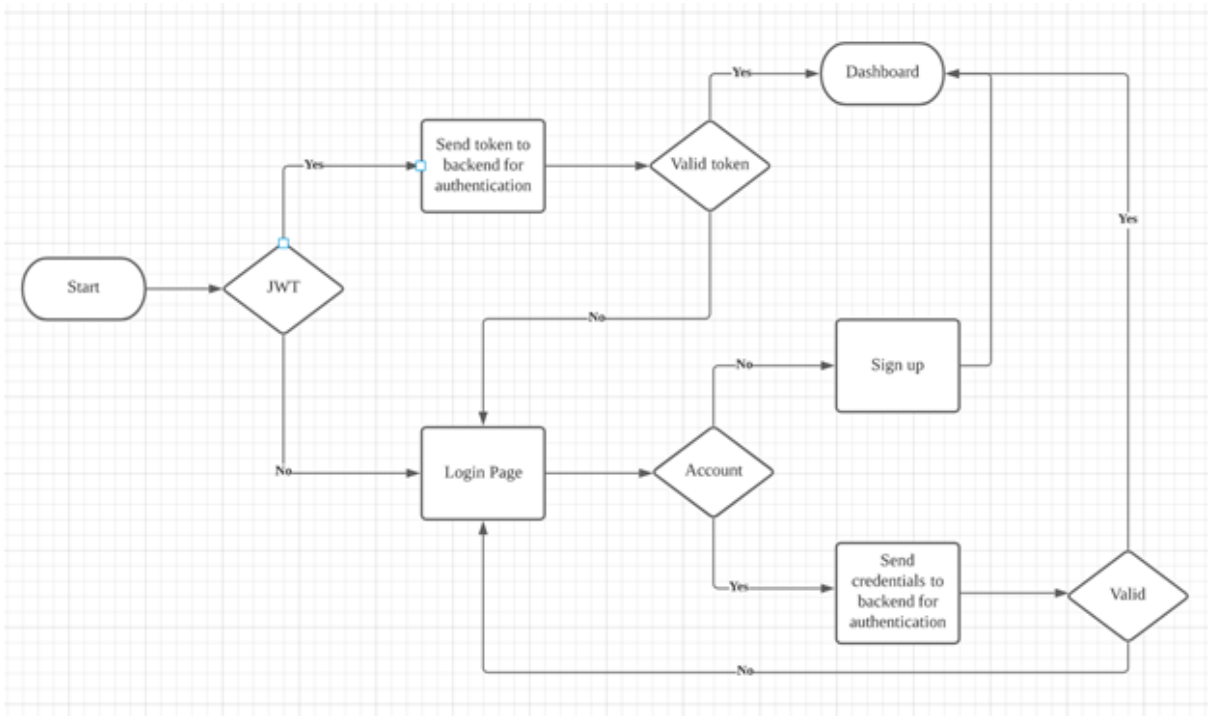

Figure 4.3: The authentication flow

4.3 Transcription List

The Transcriptor dashboard gives the user an overall list of the transcriptions which they are allow access, some of which can be assigned by an admin or uploaded by themselves. The user can then choose to edit, delete, or download the transcription as a zip file. The

transcription list is server-side paginated, meaning the pagination process itself is being done in the backend. The client will request for the page number as well as the number of items on a page, and the backend will calculate and send a response accordingly. This ensures that the client does not download too much unnecessary data, improving performance and efficiency.

Upon requesting for the data, the client will receive a list of transcription's metadata, including the name of the audio file, creation date, duration, and a transcription ID. Each entry will be represented as a card in the list. Only when the editor is open that the audio file and the transcript are downloaded, the details of which will be discussed in more details in the editor section.

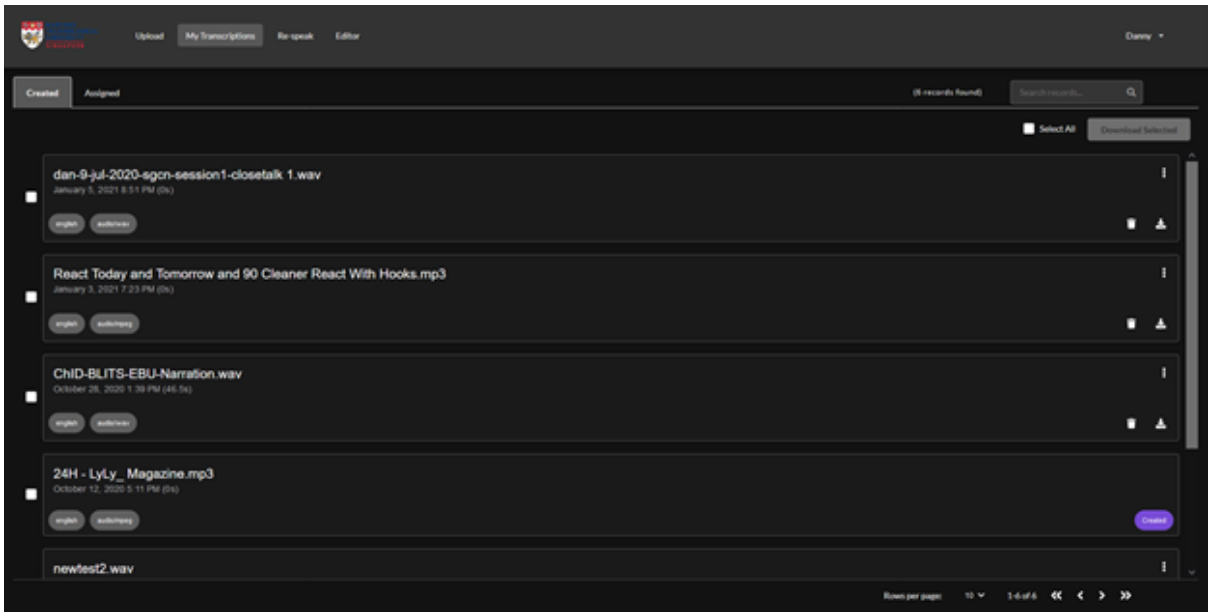

Figure 4.4: The transcription list page

4.4 Upload

Transcriptor allows user to upload an audio file in the form of MP3 or WAV, along with an optional transcript in the form of a TextGrid file. If no TextGrid file is provided along with the audio file, the file will be sent to the text-to-speech API to be processed, and the resulting TextGrid can be accessed afterwards on the dashboard. If a TextGrid file is provided, the transcript will be listed on the user's list, and the file won't be sent to the text-to-speech API. Either way, the user can then view, edit, delete, and download the transcripts along with the respective audio files.

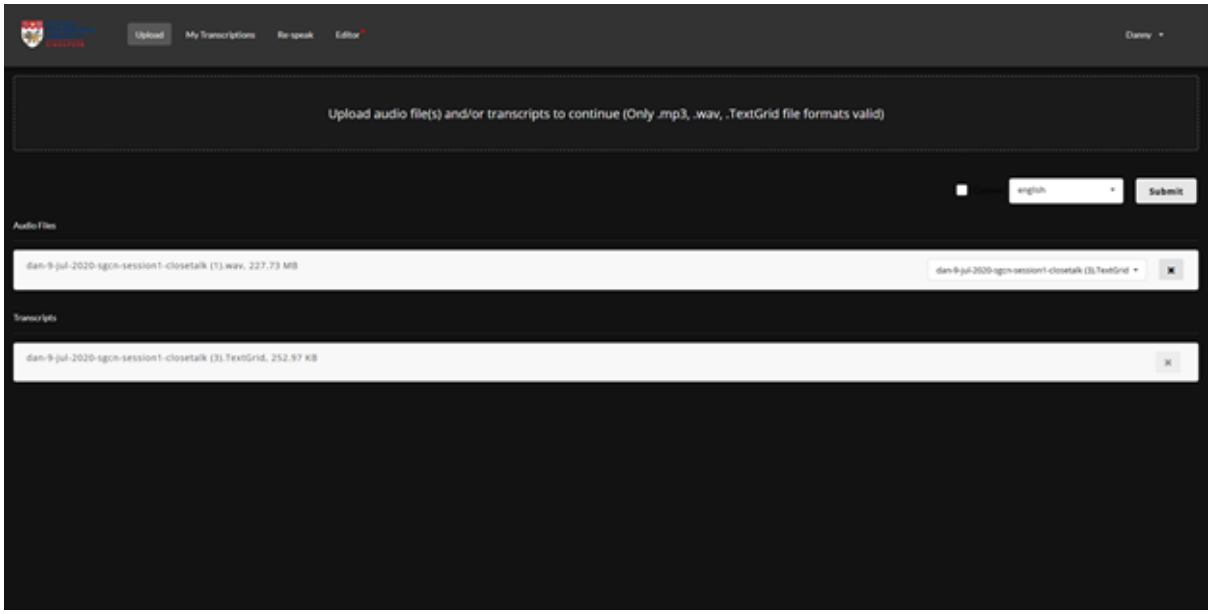

Figure 4.5: Uploading files to Transcriptor

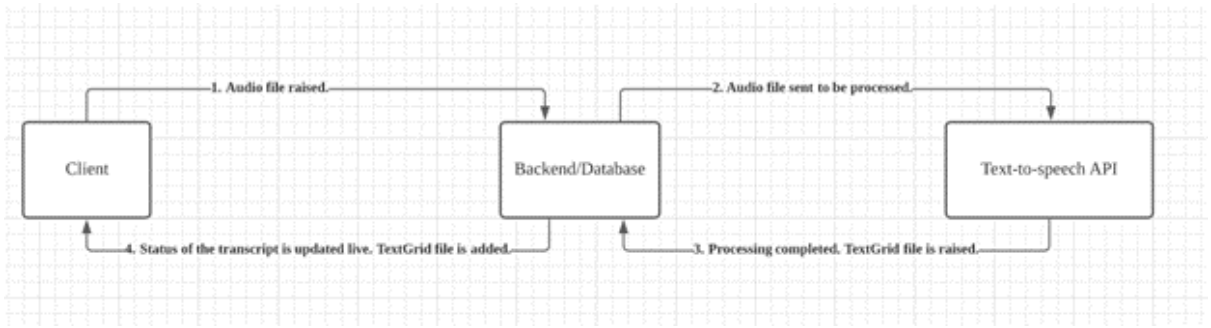

Figure 4.6: The flow of audio files if no TextGrid is provided

To view a video demonstrating the uploading process, visit `https://youtu.be/xKVrRNTkqsQ`.

4.5 Editor

The editor is the most important and complex component of the Transcriptor. This is where the user can listen to the audio file, view the transcripts of said audio, and perform any corrections if needed.

Visit `https://youtu.be/6cv_NwRhB38` for a demo of the editor's capability.

4.5.1 Data downloading and caching

The Transcriptor has a client-side cache that stores previously downloaded audio files. Upon opening the editor, the application will attempt to fetch the requested audio file

from the cache using their transcription ID. If the cache misses, the client will then request for the audio file from the server. The reason the audio files were chosen to be cached but not the transcripts is because a large portion of the loading time is downloading the audio file, and unlike the transcripts which are prone to changes from different sources, the audio file will likely stay the same throughout the time the transcription exists. After finishing downloading, the audio file will be added to the cache using the transcription ID as the key. The cache has a limit but since it varies between different operating systems, browsers and platforms, there are no reliable ways to implement a mechanism to check. However, if a cache exceeds its quota, it will throw an exception. If one is caught, the cache will be freed up using the least recently used policy (LRU). With the cache implemented, the editor's loading time is observed to have cut down significantly, on average about 40%.

File type	File size (MB)	Without cache (seconds)	With cache (seconds)
MP3	2.2	5.4	2.6
MP3	91.6	22.5	10.4
Wav	238.8	65.6	35.3

Table 4.1: The average loading time of the editor with and without cache

4.5.2 Editor's component

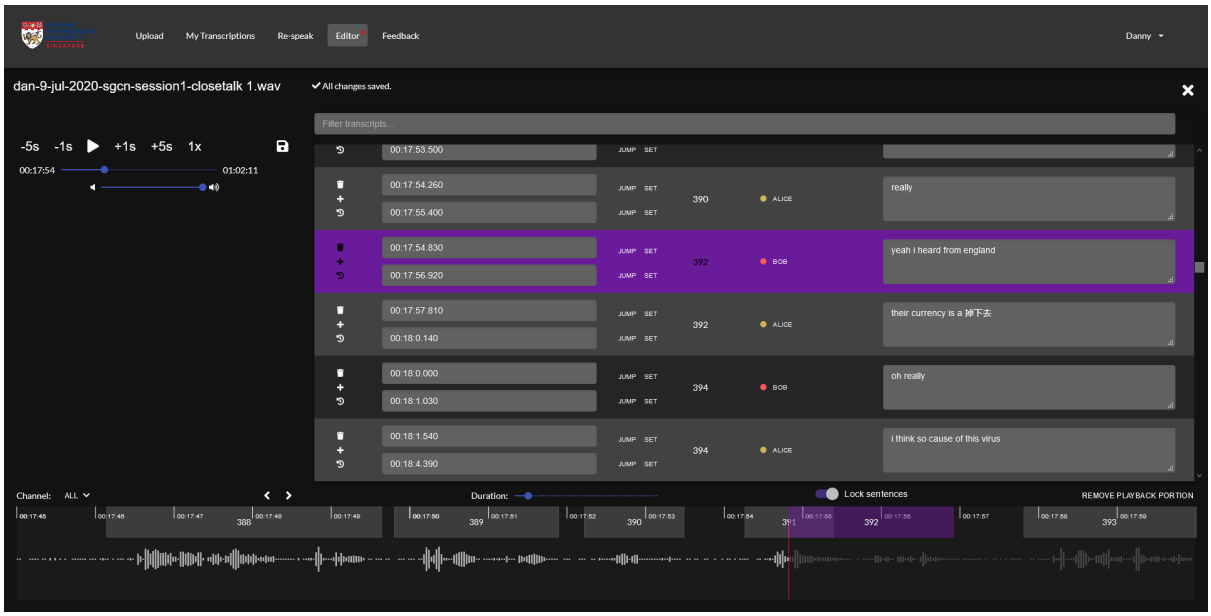

Figure 4.7: The Transcriptor's editor

There are 3 main components to the editor: the playback control, the transcript list, and the waveform, each has a different set of purposes. The editor also has a toolbar that displays the name of the transcription currently in edit, along with an indicator to show

whether the editor is saving or if the saving is successful. There is also a close button which will close the editor and redirect the user back to the transcription list.

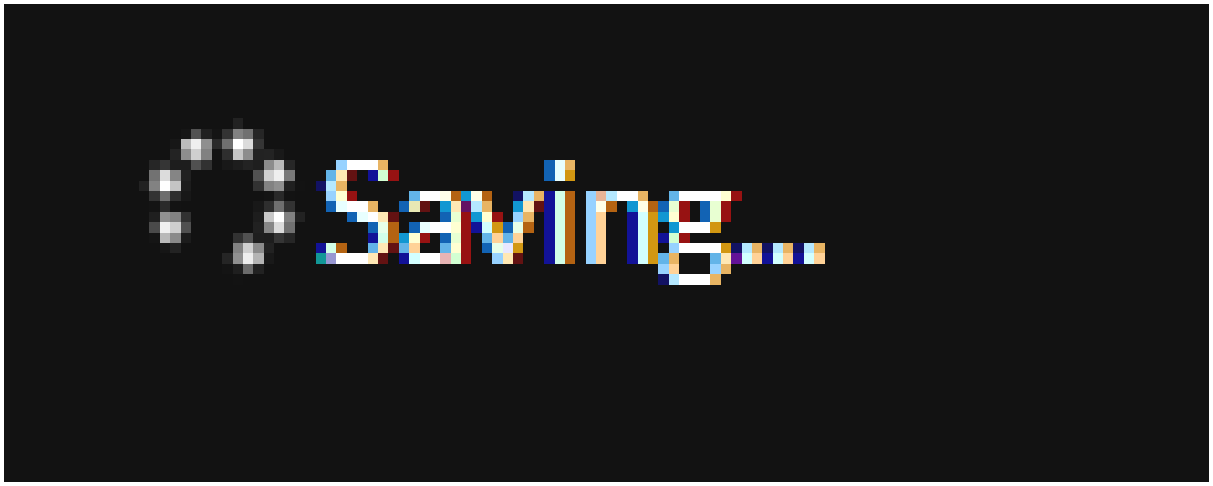

Figure 4.8: The saving indicator

Audio playback control

The playback control allows the user to control the audio playback associated with the transcript in edit. Basic functionalities like play, pause, volume control and timeline control, along with some advanced functionalities like jump forwards/backwards 1 or 5 seconds, change playback speed help the user to traverse the audio at ease, making listening to and editing the transcript easily and efficiently. There's also a save shortcut to download the transcription.

Figure 4.9: The editor's audio control

Transcript list

The transcript list displays to the user the list of sentences that are in the transcript attached to the current audio file. Each sentence is represented by a row, and each row

displays the start and end time of that sentence, its content, and some controls to help make the editing process easier. Additionally, as the audio plays, the relevant sentence based on the current time will be highlighted to make it easier for the user to distinguish which sentence is in play. On top of the list there is a search bar to filter the sentences by content to help the user navigate the list.

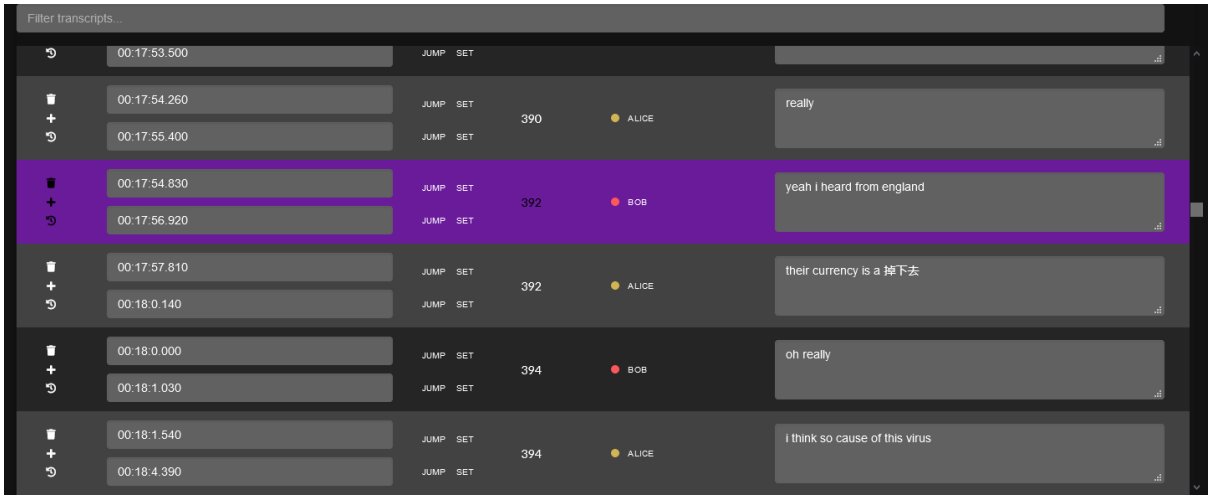

Figure 4.10: The editor's transcript list

To the left of each row there are 3 buttons. The delete button will delete the sentence that the button is pressed on. The add button will add a new sentence immediately after the end time of the chosen sentence, and the revert button will revert the text content of the sentence to the last version recorded in the server.

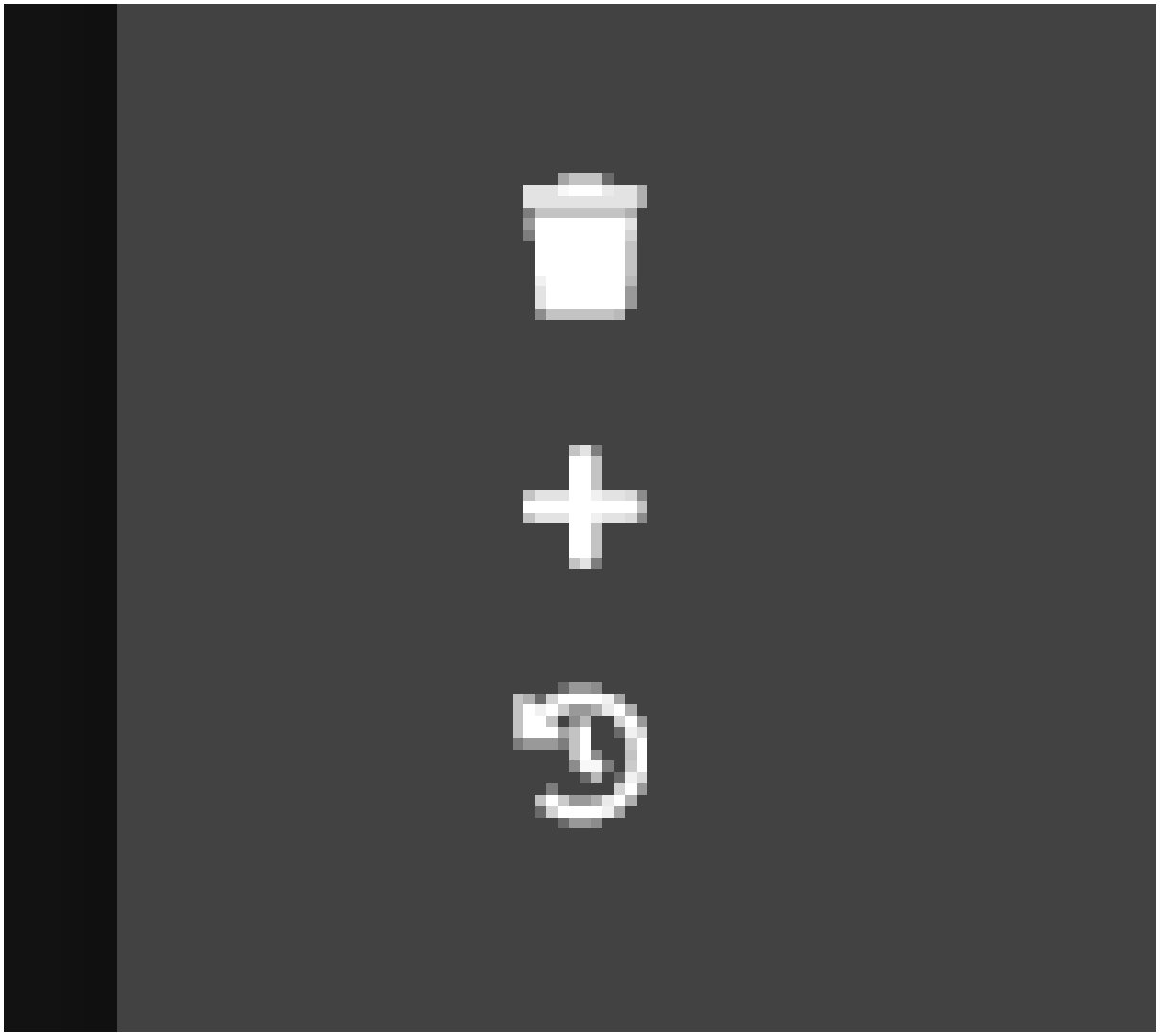

Figure 4.11: The delete, add and revert buttons

Immediately to the right are the start and end time of the sentence, along with a pair of jump and set buttons, one for start and one for end. The start and end time are displayed in `Hour:Minute:Second` format, with precision to the thousandth of a second. The jump button will jump the current audio play time to the chosen timestamp, while pressing the set button will set the timestamp to the current audio play time. This helps traversing between different sentences easier, and makes setting the start and end time less tedious, since the user can just pause at the right moment that they deem a sentence starts or ends, and set the time to that, instead of needing to navigate a timestamp picker.

Next there is the ID of the sentence used to distinguish between different blocks in the waveform, which will be explained later in more details in the waveform section. The speaker name is also displayed and color coded, each color for each speaker. The application will pick the color using a predetermined list of 10 colors, in order to have the best contrast possible between different speaker. After 10 of them are used up, the

color will be randomized using a simple hex generator algorithm. User can choose to edit the speaker name as well, and the new speaker name can be applied to only the current sentence, or all of the sentences that currently have the same speaker name. If a new speaker name is detected, the application will automatically assigned it a new color.

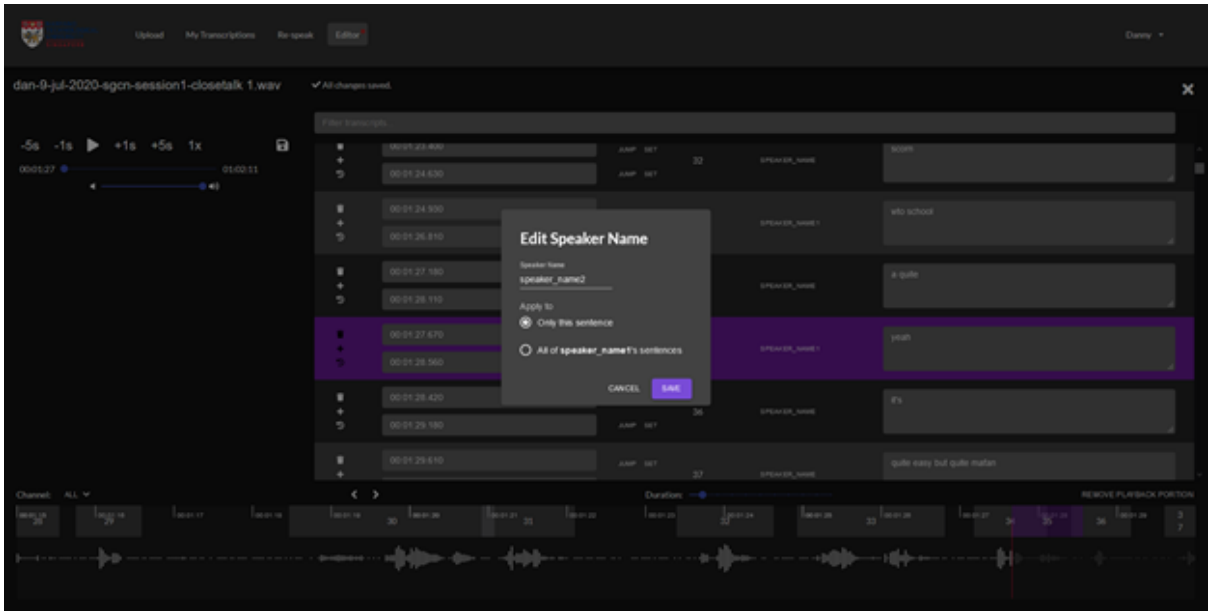

Figure 4.12: Editing a speaker name

Lastly, there is the content of the sentence. The user can edit the text like any normal text box, and the updated content will be automatically saved to the server after 500ms of no new modification. This is to ensure the update process is made as frequent as possible, but not to the point of wasting API calls which will increase server's traffic and affect both server-side and client-side performance.

Waveform

The audio and sentences are visualized by the waveform section. The waveform will only render a part of the audio at a time, both to improve visibility and performance. The duration of the section can be changed using the duration slider, ranging from 10 seconds to 1 minute. All sentences within the start and end time of the current section will be filtered and displayed on the waveform, each by a *block*. Each block is marked with the sentence ID corresponding to it's respective sentence shown in the list, and the one currently in play will also be highlighted. The position of the block is relative to the sentence's start and end time to the displayed section, and it can be drag and drop to reflect any changes the user might want to make. There's also a toggle to lock the blocks in place so that to prevent any accidental movement.

Figure 4.13: The editor's waveform

The waveform has an option to visualize the audio as the combination of all channels, or just a singular channel as a time. This is achieved by decoding the audio file into an `AudioBuffer` object, and using its `getChannelData()` method to extract each channel's data. This is especially useful for when there is audio file with multiple channels, and each is used for one speaker.

Figure 4.14: Visualizing only the second channel in the waveform

The user can also choose to jump back and forth between each section using the next and previous section. This is useful to navigate the waveform when a sentence is too long to fit in a single section. Selecting a portion of the waveform will make the audio playback to only play the selected portion, making focusing on a specific sentence easier. The selected portion can also be dragged and dropped around, and clicking on the *Remove playback portion* will deselect it.

Figure 4.15: Selecting a portion of the waveform

Chapter 5

Conclusions and Future work

5.1 Summary of achievements

In this project, a new version of the Transcriptor has been implemented. The application was overhauled as a whole, from the user interface to which libraries were used to implement. We were able to elevate the editing process to the next level by the use of the waveform blocks and the transcript list, eliminating the awkward user experience of the old editor. Whereas the old version used a stiff and very non-customizable library in *waveform-playlist*, the new version was built using a more robust and flexible library in *wavesurfer*, which not only allowed us to implement new features like rendering single channel data or selecting a small portion of the audio using the waveform, it also ensures that the Transcriptor can be easily maintained and upgraded in the future.

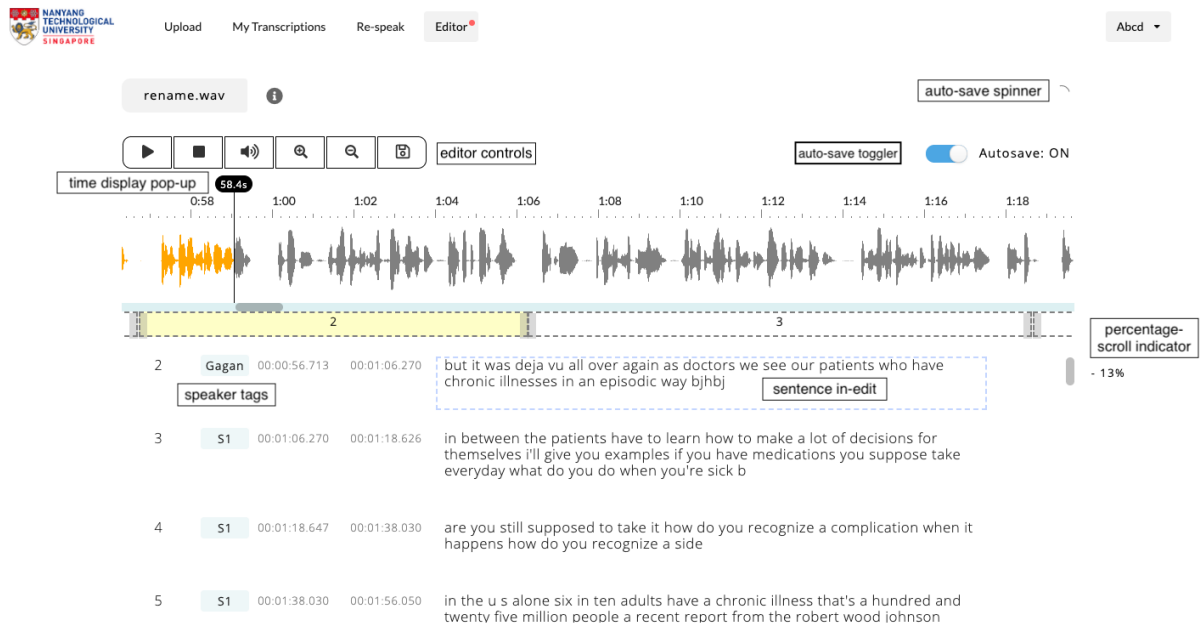

Figure 5.1: The old editor interface

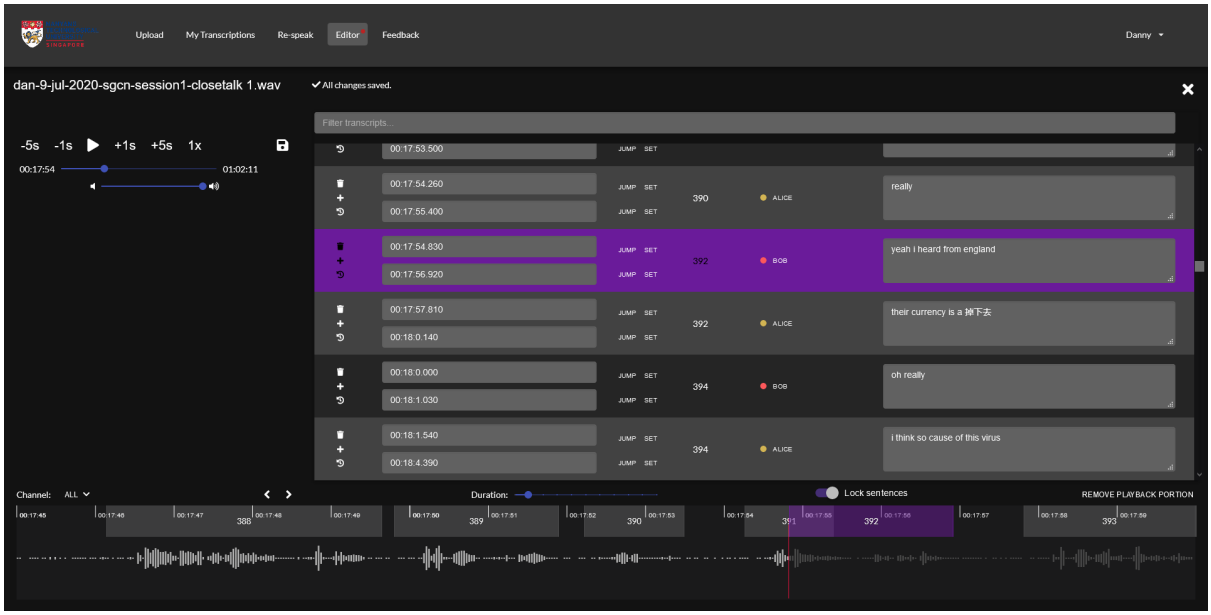

Figure 5.2: The new editor interface

The agile approach has made it easy to rework and improve the old version of Transcriptor, since each step of the process can be done separately in multiple iterations. The first iteration's focus was to rework the old application, implement the new library and redesign the editor. After the new editor's feedback has been received, the next iteration focused on improving it furthermore based on the feedback, as well as applying a new and consistent theme across the whole application. After that, the third iteration added some quality of life features, like persistent login, paginated transcription list, or selecting

a playback audio portion in the waveform editor.

The result is a new and improved version of the application, both aesthetically and usability. Test users who have used both versions have given feedback that the new editor design has made editing more efficient and convenient, both because of the redesign and the new features added. There was also no report in bad performance or stuttering, given some audio files can be 2 hours long with transcript contains almost 200 entries.

The React framework has made the developing Transcriptor a fast and efficient process, and allowed the project to progress as fast as it did, which displays the impact of the technology choices on the success of the project. This can be further cemented by the fact that the previous version of the Transcriptor was held back by the bad choice of library in *waveform-playlist*, and could only be expanded and upgraded by rebuilding. However, features are not the only deciding factor for a good application, but also performance and user experience. By using carefully placed API calls, along with the help of a performance enhancement library like *react-virtualized*, we were able to achieve a good user experience while still having a complex set of features.

By using the latest technologies, robust and well-supported libraries, the Transcriptor can be guaranteed to be easily maintained and upgraded in the future. All of Transcriptor's components are also abstract and modular, which can also help the maintainability of the application. The project repository also comes with a documentation of all the components, providing the detailed technological implementation of each.

5.2 Future work

Due to the scale and timeline of the project, a few aspects of the application were more focused than the others to ensure that the project can be finished within given time. Therefore, the application still has some aspects to improve upon. Future work for the project should explore the following options: 1) Audio streaming for the editor, 2) Offline capability of the editor and 3) Responsiveness of the application.

5.2.1 Audio streaming

Currently, the editor is downloading the entire audio file, and then also decode the entire audio file in order to visualize on the waveform and facilitate audio playback. This implementation technically works without any issues. However, the downloading and decoding process can take a long time, especially for big audio files. Furthermore, test users have responded that most of the times they will only use the editor for a small portion of a large audio at a time, potentially making the unused part of the audio wasted. Unfortunately, the current audio API for Javascript does not support partially

decoding an audio file, so implementing partial decoding is impossible. However, with the help of audio streaming, this can be solved. Instead of sending and receiving the whole audio file at one time, the server will split the audio into chunks, and gradually send it on demand. This will both reduce traffic from and to the server and the loading time on the client. In order to implement audio streaming however, both the server and client of the Transcriptor needs to be changed to accommodate for the streaming method, and it is more sophisticated than the simple whole file download. If implemented successfully, this can greatly improve user experience, as well as reduce the load on the backend service of Transcriptor.

5.2.2 Offline editing capability

With the current implementation, the Transcriptor will attempt to make real-time update to the database when there is any changes made in the editor. While this provides a good user experience as user doesn't have to worry about unsaved data, this implementation works on the assumption that the user has a stable internet connection. As soon as the connection is interrupted, the update attempts will fail, and any changes made will not reflect on the database. Even though the user can view the new data on their client, closing the application will result in loss of data. This can be solved by having a mechanism to detect a disruption in internet connection, and automatically save the data currently on screen to a local database, like `localStorage` or `IndexedDB`, allowing the user to continue using the application offline. This data will then be updated later when the connection is restored, and the application can revert back to making real time update. This feature can greatly elevate Transcriptor as an application, and prevent potential loss of data in the future.

5.2.3 Responsive design

As mentioned in section 3.3.2, the application works on the assumption that it is accessed only via a desktop computer. Therefore, the development of Transcriptor has mostly ignored the responsiveness design of the application, causing the interface to not behave as intended on smaller screen devices like phones or tablets. While most users will not be accessing Transcriptor by phones, having tablet or even a medium-size screen support will greatly enhance the user experience for the application. The process while can add a small usability to the application can potentially be a long and laborious process, potentially having to come up with new designs to accommodate for a decrease in screen size. Therefore, this aspect is not high on priority list and is only recommended to be done to round out the application as a whole, after all the key features have been implemented.

Bibliography

[1] *GitHub - axios: Promise based HTTP client for the browser and node.js.* URL: https://github.com/axios/axios. (accessed 12 Jan 2021).

[2] *GitHub - bvaughn/react-virtualized: React components for efficiently rendering large lists and tabular data.* URL: https://github.com/bvaughn/react-virtualized. (accessed 12 Jan 2021).

[3] *GitHub - naomiaro/waveform-playlist: Multitrack Web Audio editor and player with canvas waveform preview. Set cues, fades and shift multiple tracks in time. Record audio tracks or provide audio annotations. Export your mix to AudioBuffer or WAV! Project inspired by Audacity.* URL: https://github.com/naomiaro/waveform-playlist. (accessed 12 Jan 2021).

[4] *Homepage - Material Design.* URL: https://material.io. (accessed 12 Jan 2021).

[5] *HTMLAudioElement - Web APIs - MDN.* URL: https://developer.mozilla.org/en-US/docs/Web/API/HTMLAudioElement. (accessed 12 Jan 2021).

[6] *JavaScript.com.* URL: https://www.javascript.com. (accessed 12 Jan 2021).

[7] *Material-UI: A popular React UI framework.* URL: https://material-ui.com. (accessed 12 Jan 2021).

[8] *React – A JavaScript library for building user interfaces.* URL: https://reactjs.org. (accessed 12 Jan 2021).

[9] *react-virtualized - npm.* URL: https://www.npmjs.com/package/wavesurfer.js. (accessed 12 Jan 2021).

[10] *react-virtualized - npm.* URL: https://www.npmjs.com/package/react-virtualized. (accessed 12 Jan 2021).

[11] *Redux - A predictable state container for JavaScript apps. - Redux.* URL: https://redux.js.org. (accessed 12 Jan 2021).

[12] Pedro Serrador and Jeffrey K. Pinto. "Does Agile work? — A quantitative analysis of agile project success". In: *International Journal of Project Management* 33.5 (2015), pp. 1040–1051. ISSN: 0263-7863. DOI: https://doi.org/10.1016/j.ijproman.2015.01.006. URL: https://www.sciencedirect.com/science/article/pii/S0263786315000071.

[13] *Speech to Text - Microsoft Azure.* URL: https://azure.microsoft.com/en-us/services/cognitive-services/speech-to-text/#features. (accessed 12 Jan 2021).

[14] *Speech-to-Text: Automatic Speech Recognition - Google Cloud.* URL: https://cloud.google.com/speech-to-text. (accessed 12 Jan 2021).

[15] *SubPlayer - An online subtitle editor.* URL: https://subplayer.js.org. (accessed 12 Jan 2021).

[16] *Subtitle Video — Add Subtitles Online — Kapwing.* URL: https://www.kapwing.com/subtitles. (accessed 12 Jan 2021).

[17] *The most popular database for modern apps - MongoDB.* URL: https://www.mongodb.com/. (accessed 12 Jan 2021).

[18] *Transcribe - Transcription Software to Convert Audio to Text.* URL: https://transcribe.wreally.com. (accessed 12 Jan 2021).

[19] *Transcription Services - Audio & Video Transcriptions.* URL: https://www.rev.com/transcription. (accessed 12 Jan 2021).

[20] *Transcription Services Singapore - Affordable Transcription Singapore.* URL: https://www.translationservice.sg/transcription-services. (accessed 12 Jan 2021).

[21] Vinugayathri. *MVC vs Flux vs Redux – The Real Differences.* URL: https://www.clariontech.com/blog/mvc-vs-flux-vs-redux-the-real-differences. (accessed 12 Jan 2021).

[22] *Wavesurfer.js.* URL: https://wavesurfer-js.org. (accessed 12 Jan 2021).

Appendices

Function to decode an audio file into multiple channels data

```
export const getChannelAudioBuffersFromABlob = async blob => {
  const audioBuffers = []

  const audioBuffer = await blob.arrayBuffer()

  const ac = new AudioContext()

  const data = await ac.decodeAudioData(audioBuffer)

  const numOfChannels = data.numberOfChannels

  for (let i = 0; i < numOfChannels; i++) {
    const currentChannel = data.getChannelData(i)

    const newChannelArrayBuffer = ac.createBuffer(1, data.length, 44100)

    newChannelArrayBuffer.copyToChannel(currentChannel, 0)

    audioBuffers.push(newChannelArrayBuffer)
  }

  return audioBuffers
}
```

Function to get an audio section to display on the waveform

```
export const getSectionAudioBuffer = (audioCtx, audioBuffer, start, end) => {
  const { numberOfChannels, sampleRate } = audioBuffer
  const newArrayBuffer = audioCtx.createBuffer(
    numberOfChannels,
    (end - start) * sampleRate,
    sampleRate,
  )

  const startIndex = clamp(start * sampleRate, 0, Infinity)
  const endIndex = clamp(end * sampleRate, startIndex, Infinity)

  for (let i = 0; i < numberOfChannels; i++) {
    const dataToCopy = audioBuffer.getChannelData(i)
    const newChannelData = newArrayBuffer.getChannelData(i)

    for (let j = startIndex; j < endIndex; j++) {
      newChannelData[j - startIndex] = dataToCopy[j]
    }
  }

  return newArrayBuffer
}
```

Socket implementation for Transcriptor

```
const socket = io(`${process.env.REACT_APP_API_HOST}`, {
  reconnection: true,
  reconnectionDelay: 1000,
  reconnectionDelayMax: 5000,
  reconnectionAttempts: Infinity,
})

socket.on('authentication failed', data => {
  store.dispatch(setIsAuthenticatingSocket(false))
  console.log('Socket authentication failed!')
})

socket.on('authenticated', user => {
  console.log('Socket authenticated successfully!')

  store.dispatch(socketConnectionAuthenticated())
  store.dispatch(setUserAuthed(user))
  store.dispatch(setIsAuthenticatingSocket(false))

  socket.emit('join room')
})

socket.on('status updated', data => {
  /*
              Only send the important info from data
          */
  const { _id, content } = { ...data.speech, ...data.status }
  store.dispatch(socketDataUpdated({ _id, content }))
  store.dispatch(updateTranscriptStatus({ _id, content }))
})

const authenticateSocket = _token => {
  const token = _token || localStorage.getItem('token')
  if (token) {
    store.dispatch(setIsAuthenticatingSocket(true))
    socket.emit('authenticate', {
      token,
    })
  }
}

const logoutSocket = () => {
  socket.emit('logout')
```

Private route implementation to prevent unauthorized access

```
const PrivateRoute = () => {
  const { user } = useSelector(state => ({ ...state.USER }))

  return !user ? (
    <Redirect to="login" />
  ) : (
    <Switch>
      <Route path="/dashboard" exact={true} component={Dashboard} />
      <Redirect from="/" to="/dashboard" />
    </Switch>
  )
}
```