The Educational Uses of Deep Learning to Tackle Music Information Retrieval Problems

Presented by: Kittitat Bamrung

Student Number: N00201327

Date: 05/01/2024

Supervisor: John Montayne

Second Reader: Mohammed Cherbatji

CA 2 Research & Analytics

Year 4 2023-24

DL836 BSc (Hons) in Creative Computing

Declaration of Authorship

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Programme Chair.

WARNING: Take care when discarding program listings lest they be copied by some- one else, which may well bring you under suspicion. Do not to leave copies of your own files on a hard disk where they can be accessed by others. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Computing (Hons) course handbook. Please read carefully and sign the declaration below.

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.

Declaration

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

Signed:

Recoverable Signature

X Kittitat Bamrung

Kittitat Bamrung
Student
Signed by: 3c9baa37-328c-4188-9ddb-e5763b507cc3

Date: 03/05/2024

Failure to complete and submit this form may lead to an investigation into your work.

Abstract

This research will explore the possibility of using artificial intelligence to tackle the music information problem of audio classification model. It will comprehensively conclude the research, design, and development of the mentioned application, where the application aims to provide the user an easy drag and drop solution to quickly transcribe audio into guitar tablature. Additionally, the user could further edit the tablature to manually improve its accuracy. The purpose of these functionalities were to conduct a platform for guitarists and music students to have a place to promptly practice different songs or replicate the song without the need to create the guitar track from scratch. The user will have the option to separate guitar audio from its mixture for prediction, favourite the song for later practice, copy the song for edit, and listen to the predicted tablature via MIDI player.

Furthermore, the improvement of the work could be carried out to provide a better accuracy to the prediction model, where it would correctly detect more complex sounds. An improvement of utilising URLs to transcribe song from other streaming platforms would allow the users to quickly paste the URL from their desired platform and have it quickly transcribe to the user without the need to download it separately before feeding it into the application. Finally, through all these possible improvements, it would definitely widen the scope and complexity of the application and leverage the user experience in surfing the site.

Acknowledgements

I would like to express my unfeigned gratitude to my supervisor, John Montayne for various words of advice and encouraging guidance throughout the course of this project. As well as to give praises to my second reader, Mohammed Cherbatji for a great guidance to the development of the Front-end and the valuable perspectives in the early stages of the development. And last but not least, my fellow classmates, especially Stephen Gordon and Tariq Horan for providing me with feedbacks on the continuous integration and deployment of DevOps and the overall development.

Contents

1 Introduction

The research aimed to implement and design an AI-assisted guitar transcription application incorporated for amateur guitarists and instructors. The application offered a platform for musicians to quickly transcribed guitar tablatures as a form of practices. Most of the tablature accessible online today was available for popular songs and artists. However, advancements in artificial intelligence and present new opportunities to create a way to enable musicians to quickly transcribe guitar tablature and encourage a wider audience to pick up guitar. The user would be able to quickly drag and drop audio file into an input field, where the application would process the audio in the background and navigated to the tablature once it was completed. Additionally, the user would have an option to also separate the guitar track from the original mixture and ease the difficulties of finding the track with purely guitar sounds. A deep-learning model that was capable of separating a guitar from its mixture and detecting each note being played would be implemented into the model, adding additional tools for learning. The application would be hosted entirely in the web browser to allow easy accessibility for its users.

To summarise, this research would go into the analysis of the technologies and similar applications to study their strengths and weaknesses as a sort of inspirations, along with researching components needed to create the application. Subsequently, the designs and development of the application would be discussed to specify its requirements and the overall architecture such as the technologies used, as well as the reasoning behind choosing the said technologies, and how each component would be integrated together. Ultimately, the functionalities of the application would then be tested to diagnose possible bugs and errors through Unit testing, alongside the User testing to receive feedback from the outsiders about the state of the application.

2 Research

2.1 Introduction

The line between technologies and education has become significantly intertwined, such that nowadays, in the process of learning musical instruments, whether chordophone or percussion instruments, there was a learning curve that one needs to climb to conquer such skills. Among these instruments was guitar, "a very popular instruments worldly used by many musicians." (III, Mallari, & Pelayo, 2015)*. Such that, a*ccording to the Music Trades Association (2023), guitar sales were expected to reach $19.9 billion by 2025 from $17.2 billion previously in 2020.

Essentially, playing guitar and strumming a chord could be generalised as a form of decision-making (Sandberg, L., 2020). Part of strumming was an attempt to solve a problem, taking one step after the next, which ultimately, was an attempt to come up with a chord sequence. It was a learning process and could take some degree to advance in the craft. Though, due to the introduction of tablatures (tabs) shown in Figure 1, it became easier to get into guitars. This was due to the simple nature of tabs as no formal training, nor complex instructions were required to comprehend the format of music notation.

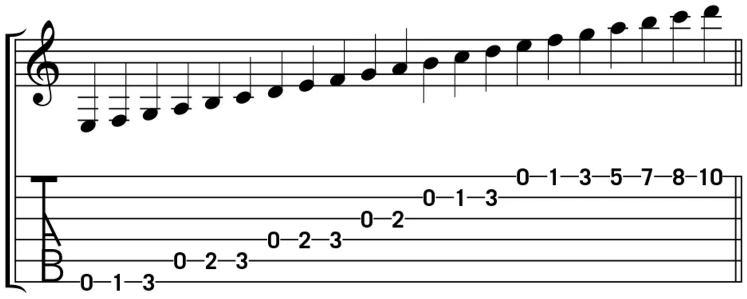


Figure 1 shows a visual representation of a tablature (Editors, 2013)

The study of this research was conducted to benefit new learners and encourage beginners' habit of practicing the guitar and eliminate the frustration and common issues with most people who had just begun to play guitar. The study proposed an alternative to inspire and strengthen guitar skills. This section of the paper expanded on the available technology to create an appropriate application to suggest an alternative and quicker method of learning guitar using a software to transcribe an audio waveform, understand about the audio signals, and transcribe the musical notation to be displayed on an online application. This led us to necessity of understanding the machine learning and the method of feeding audio and what format was best for data processing, which contributed to finalising the lowest computation cost.

2.2 Music Information Retrieval

2.2.1 What is Music Information Retrieval?

Music Information Retrieval (MIR) revolved around audio processing, which had been created as an effort to bridge the gap between modern technology and the performing arts (Kasak, Jarina, & Chmulik, Music Infomation Retrieval For Educational Purposes, 2022). In a broader sense, the study of MIR was concerned with extraction of meaningful features from music either from sources like audio signal, or symbolic representation. According to Lidy et al. (2009), MIR allowed for the creation of applications to efficiently perform an advanced search through a song to extract their features such as the rhythm, tempo, and beat tracking. This was done through the computation of audio analysis algorithms where the original audio source was extracted and the (numerical) features could be captured. These features were also called “music descriptors”. The result of these music descriptors could then be integrated into modern teaching tools for music study and learning musical instruments. The research from Kasak et al. (2022) found that methodologies of technology and musicology can be efficiently combined to create a break-through in the development of educational tools for musical student. For instance, a bunch of MIR algorithms such as tempo estimation and pitch detection could be used to identify chords that had been played in a song, which ultimately, made it possible for applications to automatically transpose audio signal of musical instruments into musical scores.

2.2.2 The Fundamental of MIR

The understanding of audio/signal processing using MIR was crucial in order to proceed with the research. The domain of MIR consisted of seven facets, where each took part in defining the nature of music. In the book written by Downie (2003), “these facets are the pitch, temporal, harmonic, timbral, editorial, textual, and bibliographic facets”. Nevertheless, the aspects of the so-called ‘multi-facets challenge’ that required the most attention and would majorly influence the research are:

a) Pitch Facet

Pitch was best described in Randel's paper (1986) as "the perceived quality of a sound that is chiefly a function of its fundamental frequency in the number of oscillations per second". This meant that the perceptual property of audio could be computed and graphed as a frequency-related scale. Thus, pitches could also be represented in numerous ways, such as note names (i.e. A#, B♭, C), scale degrees (i.e. I, II, III … VI) (Downie, 2003).

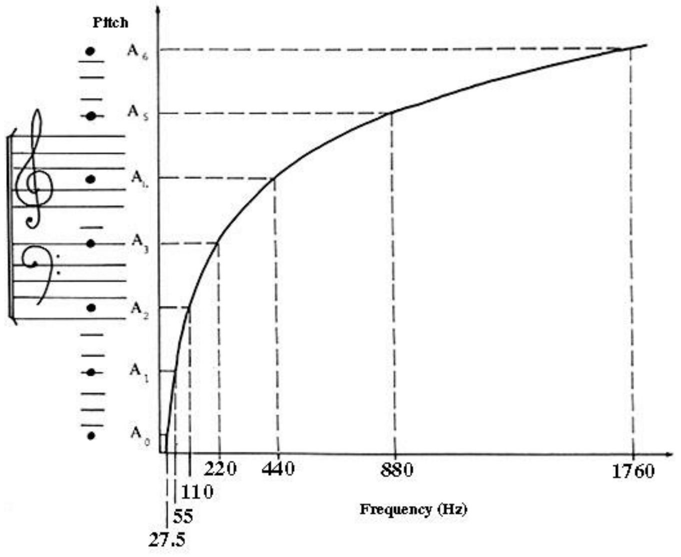


Figure 2 shows doubling frequency correspond to an increase in pitch (University, Perceptual attributes of acoustic waves, n.d.).

b) Harmonic Facet

The property of harmonic was the nature of polyphonic sounds. This occurred when multiple instruments played the same pitch at the same time, often referred to as the harmony. An example of a polyphonic sound was a chord strummed by a guitar, such that a chord of C Major, which consisted of C, F, G, and C, produced a harmonic sequence.

Symbolic representations tend to draw upon fewer computational and bandwidth resources than how audio representations do. For instance, a 10-second sample of music represented in CD-quality digital audio required approximately 14 megabits of data to be processed, transmitted, or stored. In the digestible symbolic representations, the same musical event could be represented in as little as about 8 to 16-bits. Thus, the processes of acquiring, recording, transcribing, and encoding music were all time-consuming and expensive activities. For some music, the whole new encoding schemes would also have to be developed. Thus, it was pragmatically more expedient to build systems based upon easier-to-obtain, easier-to-manipulate, music (Downie, 2003). Upon understanding the fundamental and the nature of sound allowed the adaptation of MIR in the research as it had the base for computing sounds into a digital format due to its logarithm nature and mathematical relation.

2.2.3 Audio Representation

The science behind audio signal was its root which lies in the vibration that occurred from objects such as the vocal cords of humans, the string or sound board vibration of a guitar, or the resonance from the head shakes of a drum. All of these created a vibration of air, which in turn transported the compressed waves from its origin through the destination i.e. the listener's ears, a microphone, or anything that could perceive sounds or convert these waveforms into electrical signals. The visual representation of the previous research was shown in Figure 3, where the waveform was represented by a pressure-time graph.

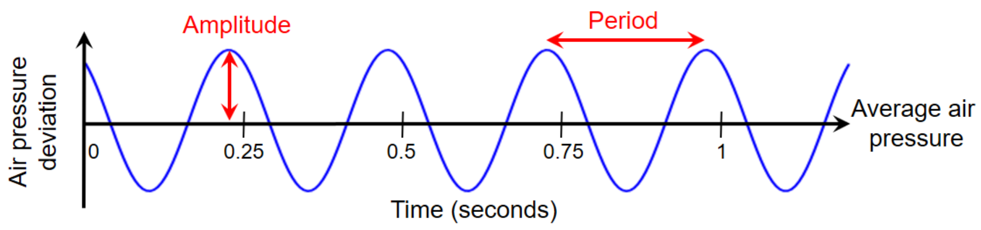


Figure 3 shows a sinusoidal waveform with the frequency of 4Hz (Müller, 2007).

In the Figure 3, the distance between two successive, either low or high, air pressure can be called a 'period'. This periodic value could be implemented to calculate the frequency of 4 hertz (Hz), due to the reciprocal nature of the period, the frequency formula was 1 divided by value of period (in this case, the period was a quarter of a second, thus 1/0.25 was 4Hz). This very same principal correlated to the acoustic representation of a musical note, where the musical terminology of this property was often referred to as the 'pitch'. Thus, the 'A' note in a guitar could be represented on the high E string as 440Hz or 440 vibrations per second (Müller, 2007).

Evidently in the paper Downie (2003) researched, the waveform representation of a sound could propose a challenging equation in correlating all the information needed to replicate its acoustic realisation i.e. multi-faceted challenge. Subsequently, even a single note of a guitar proved convoluted in contrast to playing a chord in harmony. As stated by Müller (2007), the representation of audio to date had been introduced in the manner of analogy where waveform was expressed as a continuous wave in both time and amplitude. This led to an infinite number of waves for computation. To oppose this problem, one must process the waveform into a discrete or digital representation – so called 'digitalisation', where the waveform was sampled at an equal time intervals (or periods). The value of the waveform at each period was then compressed to a discrete number of values; that's why, for instance, a compact disc (CD) usually stored sound information at a sampling rate of 44.1 kHz. The resulting discrete values were then mapped to a set of 65,536 possible values, and the end results were then encoded by a given bit depth (usually from 8-bit up to 24-bit). This process is shown in the Figure () below.

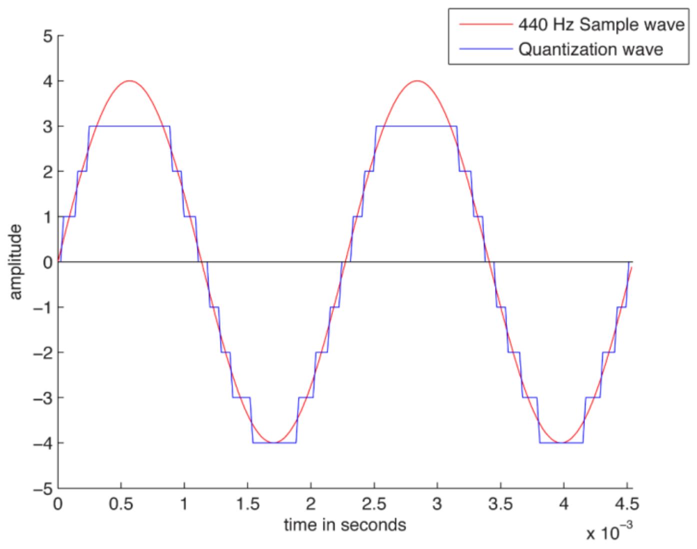


Figure 4 shows the digitalised waveform sampled at 44.1kHz with 8-bit depth. (Digitization)

2.2.4 MIDI

The digitalised waveform would often get re-sampled in such a way that it became popular amongst artists to use samples to create virtual instruments, often called Musical Instrument Digital Interface (MIDI). According to Lehrman P.D. et al. (2017), the information of MIDI could be stored in as binary values, where these information be grouped into bytes of 8 or 10 bits to combined into commands, or messages. The commands were first converted from binary (8-bit MIDI value) into a hex code i.e. 90H, 45H and 65H. Lehrman P.D. et al. further stated that these hex codes could be interpreted as the different musical notes like an *A* note, or the standard middle *C.*

As the structure may have suggested, the MIDI mainly provided a way to play notes through the information retrieved from the command, where no data of the position of the string nor the note would be supplied. This would be deemed irrelevant to the current project, but the technology could be of use in the next section.

2.2.5 MusicXML

The MusicXML was a type of notation information that designed to represent solely musical notation in a structured and standardised manner. The file type was originated from the eXtensible Markup Language (XML), which was created to ease both human and machine readabilities. It provided a means to store musical elements such that notes, keys, time signature, and even the position of strings and frets. This was a great alternatives to other language such GuitarPro, where the readability of the file was not human friendly. Due to its aims to provide the interchange of musical data between different software programmes, it had been incorporated into various applications and numerous of library were created to ease the development of applications that utilised the markup.

To conclude, the most efficient proposal for capturing sound was through the digital means due to the ability of acquiring acoustic characteristics from music. The digitalised format could be further extracted to get the necessary facets, in order to compute and extract new features from the audio.

2.3 Integration of MIR in Deep Learning

2.3.1 Deep Learning

From Bengio et al. (2017), the word Deep Learning (DL) can be summed up as an “intelligent software” that able to perform routine tasks such as comprehending speeches, images, or even diagnose medical conditions. It tackled the problems “that [were] intellectually difficult for human” due to a huge computation that required a list of formulae, or a subset of mathematical rules, which can relatively be rather straight-forward for computers. Though, as a result of extensive research on DL in the recent years, the Graphical Processing Units (GPU) had been tremendously improved to keep up with such a large computation cost that came with large amount of data being processed through multiple layers of the neural network. This rapid growth allowed for recent state-of-the-art architectures that could be performed on multiple tasks and solved problems mentioned above (Gu, et al., 2017).

2.3.2 Learning Methodology

DL resolved a given task using one of the two learning methodology called supervised and unsupervised learning. In relation to K. O’ Shea et al. (2015), **supervised learning** used pre-labelled inputs, where in each training iterations, the input values accompanied by the designated output value (also known as the ground truth) were supplied to the model to encourage their learning. Comparable to **unsupervised learning,** it did not supply any ground truths alongside the inputs. The cost function represented the rate of success whether the model was able to opportunistically lower its cost. However, in the matter of solving MIR problems, the classification majorly depended on supervised learning to detect patterns in the data. commonly utilised in most MIR problems.

2.3.3 Convolutional Neural Networks

According to K. O’Shea et al. (2015) and J. Gu et al. (2017), they stated that Convolutional Neural Networks (CNN) were traditionally a deviation of Artificial Neural Networks (ANN). The creation of ANN was strongly inspired from the concept of the way human nervous systems worked. They were designed to contain numerous interlinked ‘nodes’ (often referred to as neurons) as shown in Figure 5, where an input (usually of multi-dimensional vectors) could be taken in and distributed across the hidden layers. These hidden layers would determine and produce an estimated output. The estimation is subsequently compared with the ground truth to weigh the stochastic change and re-calculated its hidden values or ‘weight’. This was referred to as the process of learning. The product of the inputs and the weights was later distributed to the final layer called the ‘fully-connected’ layer to computed and output an optimised final value(s).

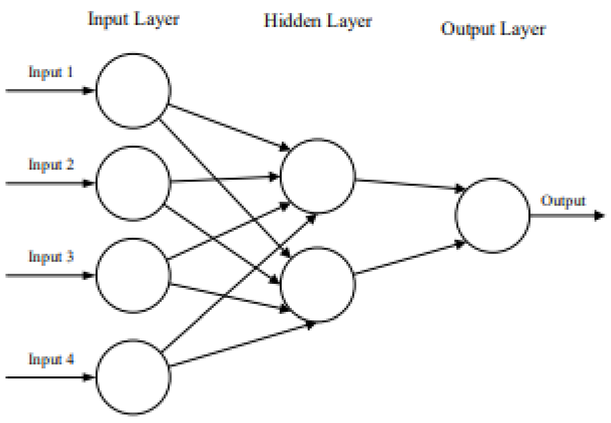


Figure 5 A representation of the way CNN works, which replicated the nervous system of human. (O'Shea & Nash, 2015)

Furthermore, in J. Gu et al. (2017)’s research, the authors stated that CNN usually consists of three main layers: **convolutional (input and hidden)** layer, **pooling (hidden)** layer, and **fully connected (output)** layers.

Convolutional layer

The purpose of the convolutional or hidden layers, as mentioned by K. O’ Shea et al. (2015), was to extract the features from the input. Additionally in Gu et al. (2017)’s paper, the layer (or neuron) could be stacked and connected to a group of neighbouring neurons to obtained new feature maps. Each layer also contained an activation function. This function determined whether to activate the function to re-compute the layers’ weight. The common activation function for CNN was a Rectified Linear Unit (ReLU) function which allowed for backpropagation algorithm to pass the error rate backward through the network to fine-tune the model’s weights. The catch was that this function only activated the neuron if the output was more than zero which can lead to weights not being updated across the board. Fortunately, there were alternatives to counter such issue such as Leaky ReLU, and Parametric ReLU functions. These efficiently combatted the model’s ability to properly fit and train the data by commemorate the negative values to allowed for neuron activations (O'Shea & Nash, 2015).

Pooling layer

The pooling layer contributed to reducing the representation of the input, as well as, supplementary lowering the number of parameters and the cost of computation of the architecture. According to K. O’Shea et al. (2015), the layer operated using an activation function called “Max” function, where instead of computing for weights, the usual kernel of a two-by-two grid and a stride window of two were ran along the spatial inputs from left to right as showing in Figure 6. The largest value in the grid was selected and became the output of the layer. This scaled the former inputs to a quarter of its original size, and thus tremendously lowered the computational complexity of the model. J. Gu et al. (2017) also added that the convolutional layer with the addition of max-pooling layer, the higher level of abstract features, as shown in Figure 7, could be extracted from the input rather than low-level features such as edges and curves.

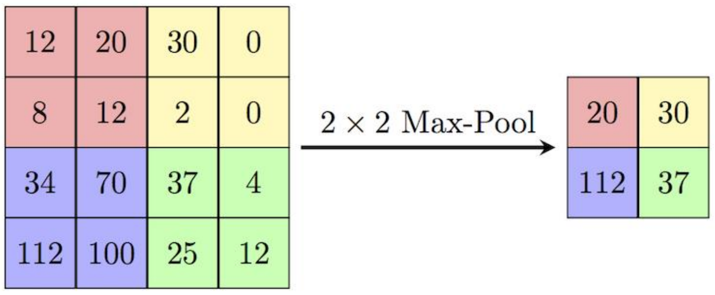


Figure 6 shows a max-pooling layer of size 2x2 with a stride of 2. This outputs the largest number in the 2x2 grid. (Britz, 2015)

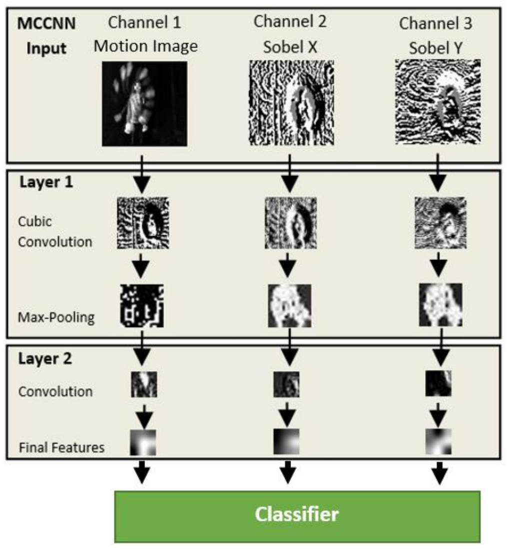


Figure 7 shows the output of max-pooling layer in Layer 1 (Parisi, Barros, Jirak, & Wermter, 2015).

Fully-connected layer

In K. O'Shea et al. (2015)'s paper, the fully-connected layer was briefly described as the classifier layer which predict the output value(s). The layer either classify a single or multiple outputs depending on its activation function i.e. SoftMax or Sigmoid functions. The concept of the fully-connected layer was further elaborated by J. Gu et al. (2017), whereby the layer takes all the neurons in the previous layers and then get the dot product of every neuron to come up with a prediction.

These layers were the essential in constructing a CNN model, which the purpose of focusing on CNNs rather than other types of ANN was due to the author's interests in exploiting its simpler structure architecture and the common choice used by various MIR problem that the author aimed to look into which is its beneficial ability to sourcing out audio sources from a mixture and transcribe the musical notation from the characteristics.

2.3.4 The uses of CNN architecture in MIR

In recent decades, MIR had shed new light to various music technologies as a result of greater computation power, and larger amount of research and information. Due to this, the automation of MIR in machine learning (ML) that required a demanding computing power of personal computers and dataset to increase its performance and accuracy became achievable ever than before (Inskip, 2011). The development has allowed for tools such as Music Source Separation (MSS) and Automatic Music Transcription (AMT) to be created.

The progress of DL has been heavily expressed in the research paper from Kasak et al. (2022), where MSS had made it possible for musical students to appropriately educated themselves with the right material to practice their instrument and create a play-along environment. This was done by retrieving individual instruments from the original song where the present of noises are low and controlled as possible to yield the ideal result as shown in figure 5. Through MSS, students could remove the instrumental stem from the mixture to practice along the company of a given song without an expensive recreation of the source material.

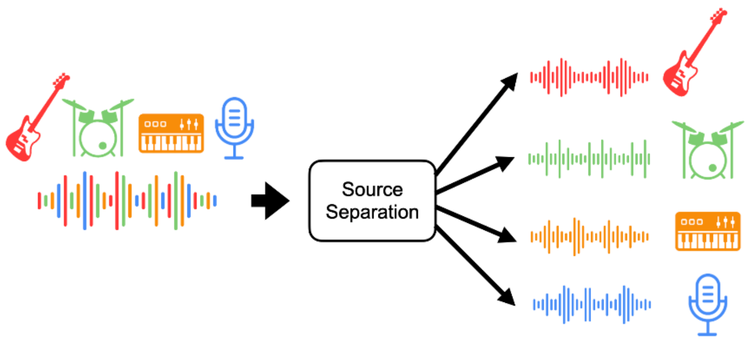


Figure 8 Simplified version of how MSS works (Manilow, Seetharaman, & Salamon, 2020)

Similarly, the AMT was created to seek the possible capability of converting musical signals into a variant of musical notation. The procedure included a huge range of tasks such as pitch estimation, onset and offset detection, instrument recognition, and beat and rhythm tracking etc. (Benetos, Dixon, Duan, & Ewert, 2019). This meant the allocation of music sheet may prove impossible for numerous of streaming platforms to supplement such a large number of musical contents due to the time-consuming process of music notation. As a result, many musical students may find interests in the existence of such educational tools to further educate and allows them to practice to a song of choice without spending an enormous amount of time to recreate the source material.

Although, the purpose of CNN was intentionally resided in addressing the property of images, the method of MIR allows audio to transform into an audio representable format i.e. Spectrogram as shown in Figure 9. These spectrograms were able to be fed into the model as an input to taught it to recognise shapes and output an estimation of what we wanted to predict. This was due to the nature of CNN, which reduced the number of parameters and improved generalisation by sharing parameters to captured local patterns in the input. The CNN was exceptionally great at analysing images and extracting common patterns. Thus, it has been the go-to method in the field of MIR to execute MSS and AMT.

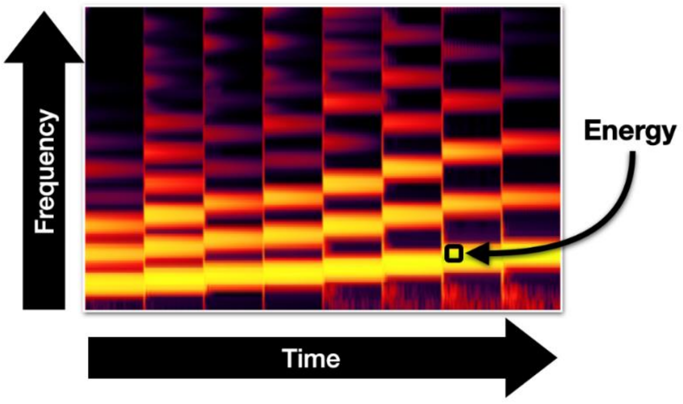


Figure 9 show a representation of a spectrogram.

2.3.5 CNN Applications

AMT

Automatic Music Transcription, or AMT, benefited individuals who shared an interest in music and wanted to casually play musical instruments, even without prior knowledge of musical theory. In this research in particular, the author aimed to focus on the extraction of tablature from guitars. Regardless of the beneficial factor of tablatures in encouraging beginner guitarists to practice, the availability of such convenience was not always applicable. The suggested tool aimed to contribute to creating tablatures from the guitar audio. Though the article written by Y. Jadhav et al. (2022) showed that the most recent annotated dataset only supplied guitars with six strings and only consisted of 19 frets. With that in mind, the nature of each guitar string which slightly differed from one another, such that on a guitar, the combinations of strings and frets could produce the pitch of the same note. Jadhav et al. showed the possibility of producing a tab of the string/fret combinations that may be utilised to play the combination of notes on guitars with six strings.

Though transcription of guitar could be composed by means of a musical score or tab. Fortunately, the duration of time a note is played, the string fretting, and position of fingering were provided by these compositions. As previously mentioned in Downey's paper (2003), the notes with identical pitch may be played on different strings of the guitar. A tablature exploited the numbers to represent the strings and the placement of the frets. This allowed players to play the guitar naturally even without much understanding about theory that went behind music. Tablature eliminated the variety of fingerings and made it accessible for players to pick up guitar. Respectively, the reason for numerous novice guitarists, realised the benefits of tablature when playing the instrument.

According to an article written by Tio (2019) about the automation of guitar transcription, the most concerning aspect of a string-fret combination when it came to modelling a DL model was the representation of the strings and frets matrix, where strings were usually represented as the six strings in a classical guitar and the frets can usually be represented as eighteen frets, in addition to an extra place when no note is played, which concluded to a 6 x 19 matrix. In order to achieve such output from the model, an DL architecture called multitask learning was required.

2.3.6 What is Muti-task learning?

Multitask learning (MTL) had been often referenced as joint learning or learning with complementary tasks. The method of MTL was purpose to resolve issues, where the model needed to yield and optimise more than one loss. In this particular case, an CNN architecture of six branches, where each produces an output vector of size 19. In each iteration of training, the model would be expected to become an expert at differentiate sound from certain strings. A representation of such architecture was given in Figure 11 (Jadhav, Patel, Jhaveri, & Raut, 2022).

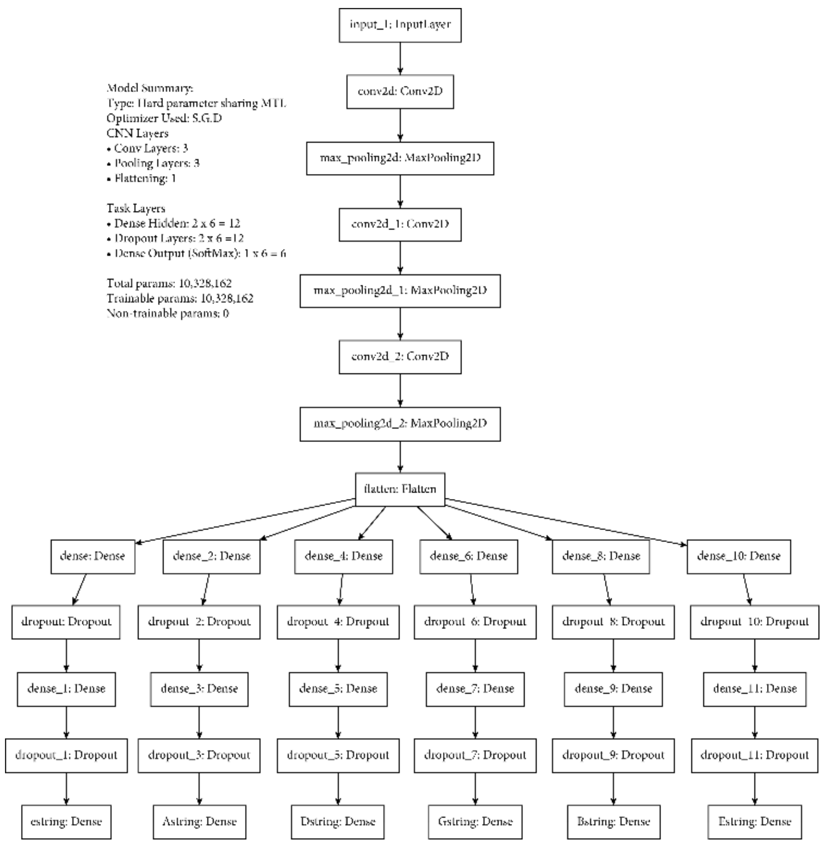


Figure 10 represented the CNN architecture with MTL that branched out into six endpoints in correspond to the output of the six strings (2022).

The SoftMax activation function was also used to classify categorical values of the given vector and turn converted them into values that summed up to one. The value with the highest correlation would be assigned the value of one while the others would be assigned to zero. By means of gathering these vectors of size 19 from six branches, one would be able to achieve a string-fret combination of the entire fretboard of the guitar presented by the input of the spectrograms. Y. Jadhav et al. (2022) utilised the existing architecture of the LeNet architecture due to its simplicity and low computational cost to use as a base for the architecture.

The resulting model could distinguish chords and notes of an audio source that only has an audio of the guitar without any external noises of other instruments. Due to this, the end result may not be appealing to other users, especially for users who required tablatures with other instruments. In order to resolve such issue, the tool such as ASS could be utilised to preprocess the audio files before handing over to AMT. This tool would allow for the extraction of specific sources from the audio mixture such as the guitar, bass, keyboard, drums, and vocals. This could tremendously save the students time as they may not have time to completely recreate each source from scratch (Kasak, Jarina, & Chmulik, 2020).

Audio Source Separation

Audio Source Separation (ASS) is the process of isolating one or multiple audio sources from a mixture. Due to the expressive nature of this MIR problem, research papers from numerous authors such as (Stoller, Ewert, & Dixon, 2018) and (Takahashi, Goswami, & Mitsufuji, 2018) were written where they attempted to separate musical instrument/vocal from a musical mixture. Music was used as their research topic due to its distinct problem from other types of source separation due to the multiple factors that made it uniquely challenging i.e. the multi-facet challenge mentioned in the previous section.

Nowadays, recent state-of-the-art systems for music separation used both spectrogram and waveforms as input. However, the go-to method for ASS mostly opted for spectrogram representations of the audio signals rather than waveform representations. This is due to the ability to access audio signals as time-frequency domain, where the necessity of the presence of frequency ranges were of importance in order to separate musical instrument(s) from its original mixture. The conversion from waveform to spectrograms often employed Short-Time Fourier transform (STFT) and applied to the input mixture signal. The resulting spectrogram would then be divided into the magnitude and phase elements, in which the magnitudes were subsequently fed through as inputs to a CNN model, and in turns compute the estimated spectrogram magnitudes for individual sound sources. These magnitudes of individual audio sources were later coupled with the mixture phase and converted with an inverse STFT to the time domain (a waveform) to generate corresponding audio signals (Open Source Tools & Data for Music Source Separation, n.d.).

To further elaborate, the time-frequency (TF) domain was a 2-dimensional matrix that represents the frequency contents of an audio signal over a period of a given time. The imagery representation of a TF was a heatmap (shown in Fig 9), where the time was presented along axis of x and frequency along the y-axis. Any individual entry of TF matrix is called a TF ‘bin’. The TF bin represented the amplitude of the audio signal at a particular time and frequency. The intensity of the energy in Figure 9, signified that brighter colours indicate high amplitudes, vice and versa. There were various methods to represent TF domain, though this research only aimed to cover the most frequently used TF representation (Stoller, Ewert, & Dixon, 2018).

Mel-scale Spectrograms

In relation to frequencies, the perception of human hearing was logarithmic. This meant that loudness of sound is perceived as proportional to the logarithm of the power transmitted through the air. Thus, the Mel spectrograms was often used to reduce the cost of computation on DL-based approaches, due to its smaller number of Mel-spaced frequency bins as compared to the number of linearly-spaced frequency bins. A visual comparison is shown in Figure 11. Notice the how the y-axis in the Mel-spaced spectrogram was squishing higher frequencies and leaving more space for lower frequencies.

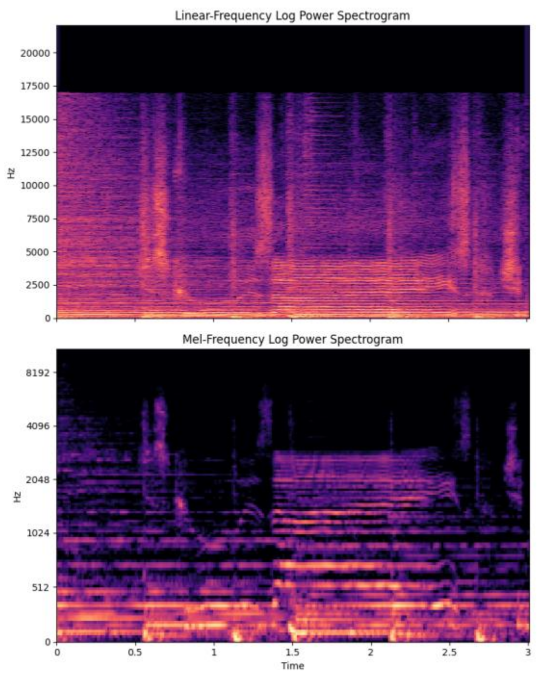


Figure 11 A visual comparison of linear-scaled vs Mel-scaled y axis. Lower frequencies have a larger representation in a Mel spectrogram.

Masking

The method of masking was widely known in DL for its uses in language modelling and computer vision. It is also played a crucial part in a lot of modern ASS approaches to estimate sources from a mixture. In order to separate an individual source, masks for existing sources in the mixture must be created.

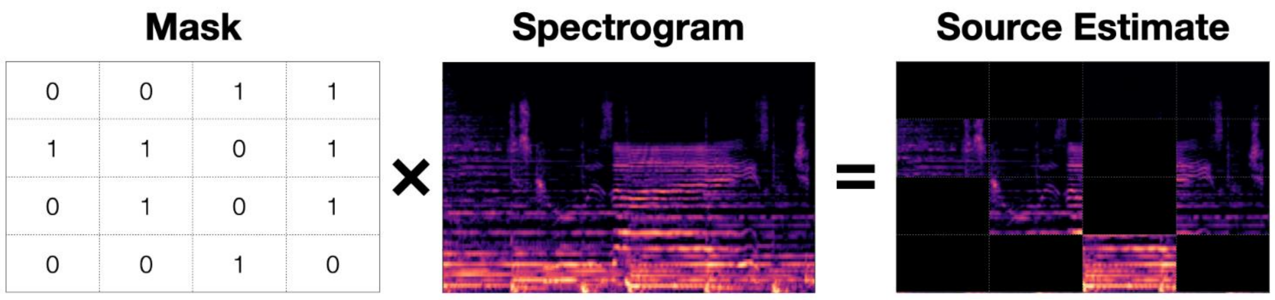


Figure 11 shows how mask estimation worked

According to (Manilow, Seetharaman, & Salamon, 2020), the characteristic of a mask is similar to a matrix, where each value in the entries would be assigned to a float of zero to one. This probability of a value reflected the amount of energy of the original mixture that a source contributes, wherein the case of the author's research would be musical instruments i.e. guitar, piano, or drums. This meant that in a particular TF bin, a float of one would allow all of the sound from the mixture through and vice versa. The ASS has proven a useful tool to further added on to the AMT system as it allowed for the separation of guitar and, eliminate the problem that Y. Jadhav et al. encountered.

2.4 Conclusion

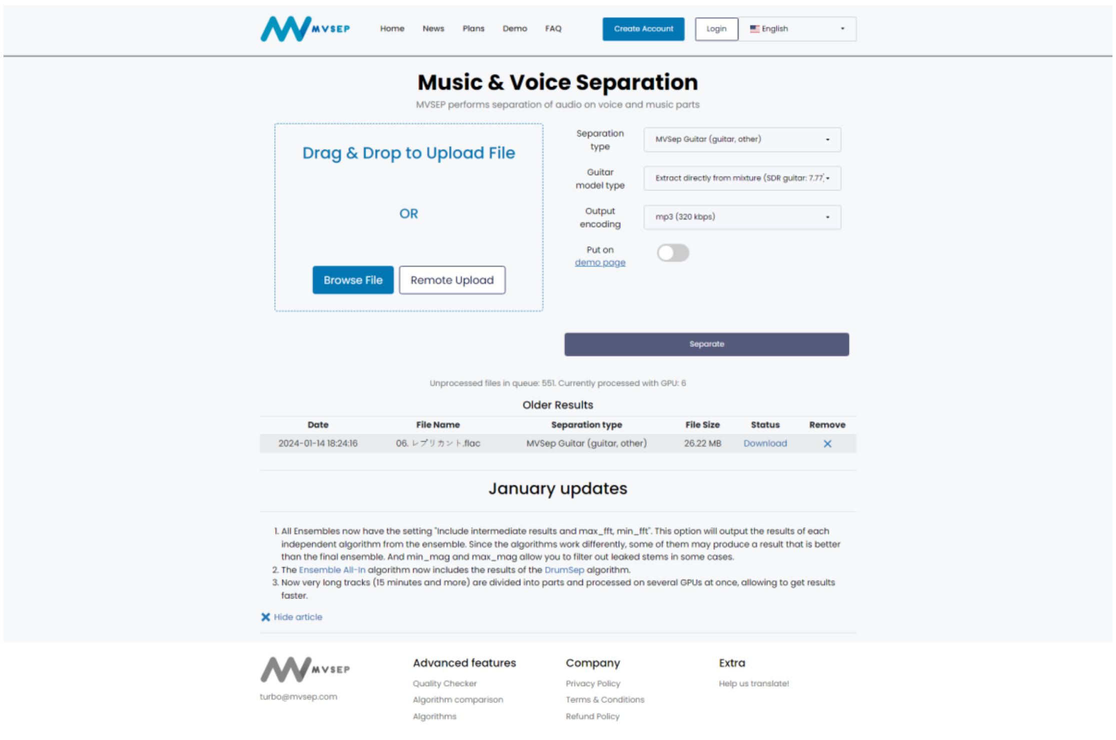
To conclude, there are a great opportunity to research in this rather growing topic, as it proven to be useful for the author's own research in their final project in constructing automatic guitar transcription. The benefit of creating a CNN model that could pipelined an audio file to separate its mixture into individual source, and extracted the guitar source to transcribe tab automatically was very interesting and challenging area of research to get into. As it would beneficially allow the development of educational tools for musical student.

3 Requirements

3.1 Similar Applications

The existence of the applications similar to the research proposal were minute and could be considered quite novel in the conceptual aspect of open-sourced MIR. Though, there might be applications developed in a proprietary environment such as automatic caption system created by YouTube, where vocal audios were individually extracted to be captioned by an algorithm. Regardless of the different algorithms used, there were similarities between the proposal and the automatic captioning system in its procedures to separate audio source(s) and transcribe the waveform, whether for the purpose of subtitling a video or transcribe music notation into a tab format.

3.1.1 MVSEP - Audio Source Separation



Music & Voice Separation

MVSEP performs separation of audio on voice and music parts

Separation type: MVSep Guitar (guitar, other)
Guitar model type: Extract directly from mixture (SDR guitar: 7.77,
Output encoding: mp3 (320 kbps)
Put on demo page

Unprocessed files in queue: 551. Currently processed with GPU: 6

Older Results

Date	File Name	Separation type	File Size	Status	Remove
2024-01-14 18:24:16	06. レプリカント.flac	MVSep Guitar (guitar, other)	26.22 MB	Download	✕

January updates

1. All Ensembles now have the setting "Include intermediate results and max_fft, min_fft". This option will output the results of each independent algorithm from the ensemble. Since the algorithms work differently, some of them may produce a result that is better than the final ensemble. And min_mag and max_mag allow you to filter out leaked stems in some cases.
2. The Ensemble All-In algorithm now includes the results of the DrumSep algorithm.
3. Now very long tracks (15 minutes and more) are divided into parts and processed on several GPUs at once, allowing to get results faster.

Figure 10 shows the web application that could separate audio sources from audio mixture using existing Neural network models.

The above Figure illustrated the web application that utilized variety of ASS models to separate sources from a mixture. This included a model that separates vocals from background noises, vocals, drums, guitars, and basses from a mixture, or even removing reverbs from audio.

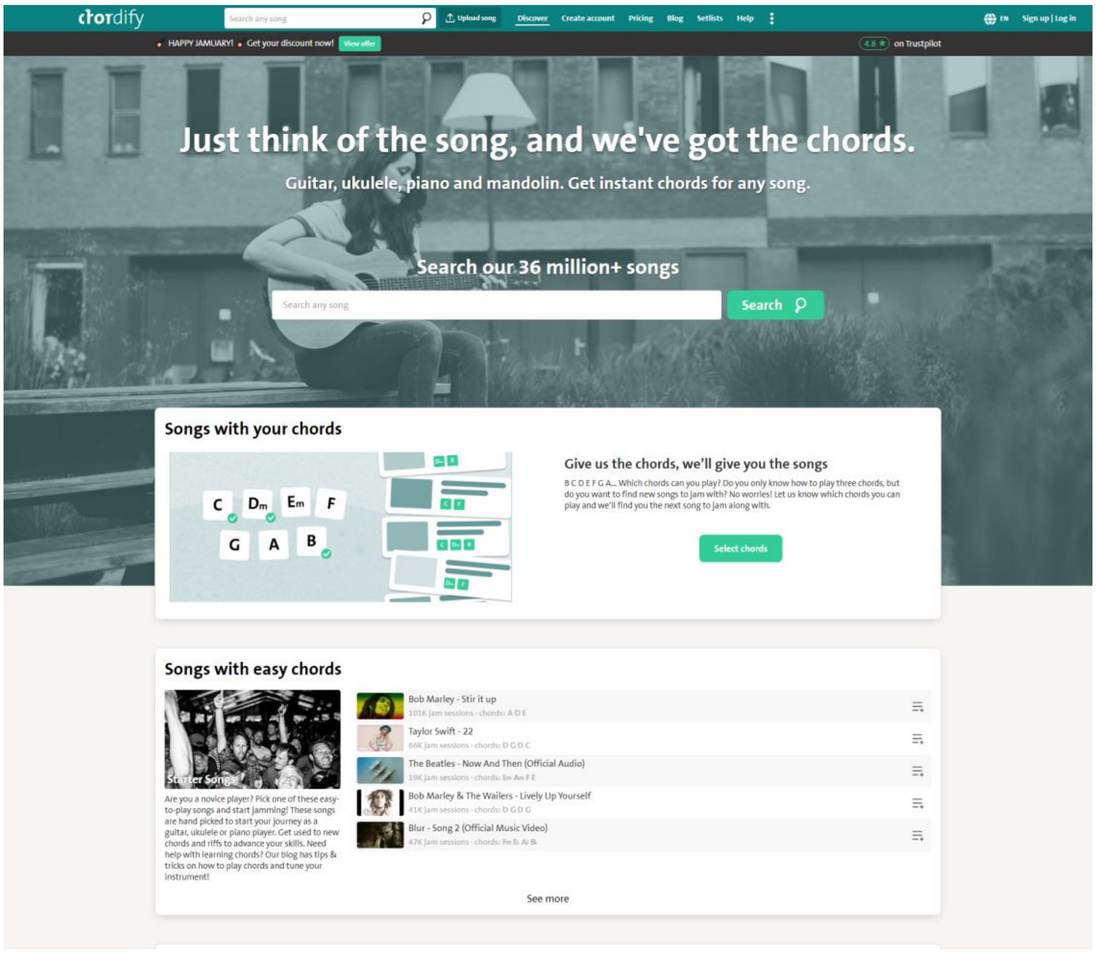
Advantages

- Collection of ASS model to select from; this included the state-of-the-art models of recent years such as Demucs4 HT, MVSEP, and LarsNet.
- Point and click – easy to use without the knowledge of coding.
- The separated sources were stored in a database allowing playback ability.
- No requirement for signing up to the platform.

Disadvantages

- Needed to pay for premium service – allowed fast tracking and skip long queues.
- Higher resolution models were locked behind subscriptions.
- Long waiting time in queues.

3.1.2 Chordify.net – Automatic Chord Transcription



Just think of the song, and we've got the chords.

Guitar, ukulele, piano and mandolin. Get instant chords for any song.

Search our 36 million+ songs

Songs with your chords

Give us the chords, we'll give you the songs

B C D E F G A... Which chords can you play? Do you only know how to play three chords, but do you want to find new songs to jam with? No worries! Let us know which chords you can play and we'll find you the next song to jam along with.

Songs with easy chords

Are you a novice player? Pick one of these easy-to-play songs and start jamming! These songs are hand picked to start your journey as a guitar, ukulele or piano player. Get used to new chords and riffs to advance your skills. Need help with learning chords? Our blog has tips & tricks on how to play chords and tune your instrument!

Bob Marley - Stir it up
101K jam sessions · chords: A D E

Taylor Swift - 22
66K jam sessions · chords: D G D C

The Beatles - Now And Then (Official Audio)
19K jam sessions · chords: Em Am F E

Bob Marley & The Wailers - Lively Up Yourself
41K jam sessions · chords: D G D G

Blur - Song 2 (Official Music Video)
47K jam sessions · chords: F♯m E♭ A♭ B♭

Figure 11 shows the web application that allows users to transcribe chords from YouTube videos, uploaded files, and audios.

Chordify was a platform that allowed users to search through a library of songs to see the list of chords that were played in the specific song. This platform also used an algorithm which extracted chords from songs on the surface without any details of guitar riffs.

Advantages

- Available for all platforms i.e. web, iOS, and android.
- Chords were available for piano, mandolin, ukulele, and guitar.
- 36 million of songs that were already transcribed.
- Users were able to upload their own audio to transcribe chords.
- The website showed songs that were easy for beginners to play.

Disadvantages

- Premium subscriptions were required to upload your own audios.

- Only general chords of the song were transcribed from the song and not solos, or guitar riffs.

3.2 Requirement Modelling

3.2.1 Personas

Cassandra

Cassandra is a 17-year-old Highschooler who has been exposed to the world of music by her peers. This happened when she was invited to attend an underground type of concert by a couple of her peers. The energy of the crowds and the cheer noises of the music had greatly inspired her to pick up an instrument. She decided to pick up a guitar due to its wide range of practical resources. However, the existence of music sheet nor the tab were scarce due to the greater difficulty in discovering artists with lower popularity.

The above persona perfectly described all the attributes proposed in the research – an ideal user to utilise the application. They were particularly exposed to the benefits of this application where the platform could provide the stepping stone into learning guitar via the tab. This allowed the user to get the grasp of the properties such as the position of the frets in each string along the guitar, the shape of guitar chords, and the introduction of guitar scales.

Peter

Peter was an intermediate guitarist studying a bachelor's degree in music production. During his time in the university, he was often tasked with assignments to mix and master a song of his choice. The process of producing individual instrumental sources were doable, though could be time consuming. Consequently, Peter searched for an optimal solution to quickly automate this process to allow him to centre his attention towards improving his mixing and mastering skills.

The above scenario allowed a better insight into the needs of these kind of applications in other type of scenarios.

3.2.2 Functional Requirements

The purpose of functional requirements was to identify the upmost required features that an application needs to meet in order to satisfy the users. Such requirements only opt to feature functionalities that an application should be able to perform at a base level. For this application, the required functionalities were:

1. **Upload audio file**: a user needs to be able to upload an audio file to be transcribed.

2**. Be able to transcribe guitar tab from a file**: An audio needs to be converted and return a transcribe of the guitar tab.

3. **Be able to display the guitar tab from storage**: Once a file has been transcribed, the user's must be able to see and access the tab(s).

4. **Use AI to detect a chord and riff from the guitar source of a mixture**: An algorithm needs to be put in place to transcribe the chords/notes from the guitar.

3.2.3 Non-functional Requirements

The non-functional requirements, on the other hand, identified the additional features that an application that were complementary to the existing features. For this particular application, these non-functional requirements were:

- Be Compatible with all web browsers.
- Sync the result of the tab to a video player or an audio player to seek forward/backward in the mixture.
- Allow users to adjust audio volume.
- The ability to transcribe not only guitar source, but all sort of instrumental sources in a mixture.
- The ability to edit the working tablature.
- Be able to favourite songs for later.

3.2.4 Use Case Diagram

In the Figure shown below, an actor (the user) would be able to utilise the application, where they could upload an audio to transcript their guitar into tablatures without the need to login/register.

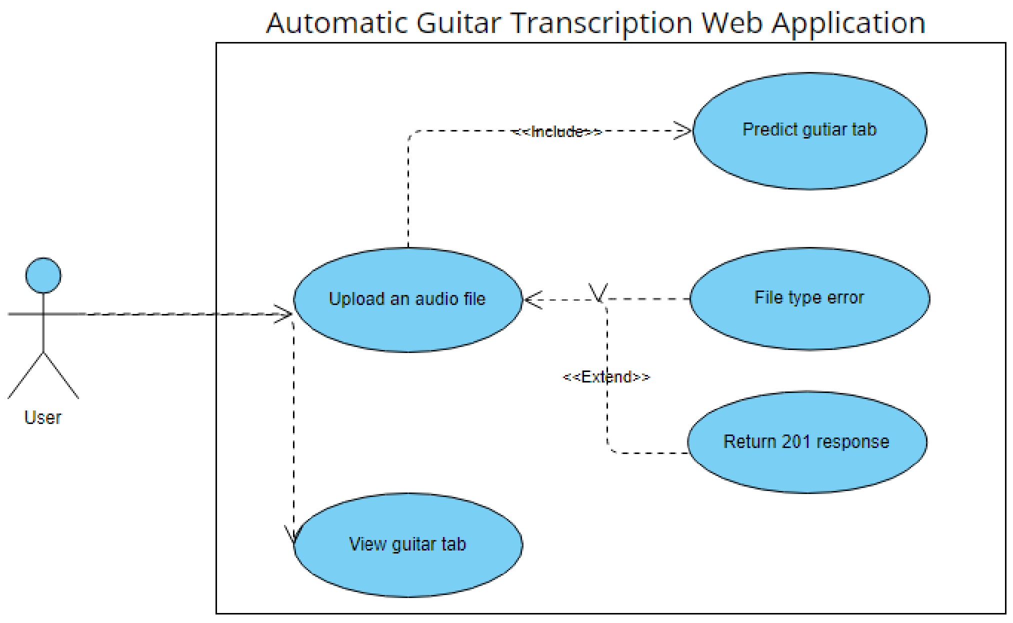


Figure 12 shows the use-case diagram of the web application.

3.3 Feasibility

Due to the nature of the full-stack web application, there were a definite number of components that would need to be utilised to build the final product. These components were:

The Tab Visualiser Component:

The component summed up the architecture of the front-end development, which were required to display a tab of a given audio. The user interface (UI) of the tab were developed in React to take advantage of its Single Page Application (SPA) and virtual DOM. The data of each tab would be stored on a server on the cloud provider where the author was able to use the AWS S3 to freely store tab in XML file. Although, the cloud provider would not be able to establish a connection between the client, and the back-end system on its own. The file needed to be store in a database of sort, in order to store its file path. This was handled by MongoDB, as the author deemed the ACID transaction, and other features were not required for this particular scenario. The final iteration of the web application would be deployed on a Firebase project, as it offered free servers for small-scale projects, and were suitable for the initial testing phases of the project.

The ML component:

The ML development was the crucial aspect of the application due to the part it played in detecting chords/notes and realising the predictions from spectral images. In order to accomplish this, a model must first be trained. The process included finding existing dataset that allowed the user to detect each string in a guitar, and labels of which notes and frets were being played. Once the model was trained and tested, actual real-world audio needed to be altered in such a way that it would understand by the model. The final procedure was to host the model as an API to take in file and predict the solution as an XML file. This process was accomplished using Google CoLab and later re-compiled to a Flask app which would be invoke by an API endpoint in order to execute the prediction. The pricing of Azure was free and included in the student starter pack, hence the free tiers would cover the expenses of usage due to the free credits available for the first 12 months of signing up.

3.4 Conclusion

To conclude, the application aimed to ease the process of learning guitar for new beginners using AI-integrated ML model. The gathered requirements included an analysis of similar applications such as MVSEP, and Chordify, highlighted their advantages and disadvantages. Additionally, realising the components and complexity of the application allowed the author to visualise the potential to provide an innovative and useful tool for guitarists, and guitar instructors.

4 Design

4.1 Introduction

The requirements of the design section were briefly elaborated in the Feasibility section. It stated the importance of the components needed to successfully execute the application i.e. the tab visualiser component and the ML component. In prioritising the personas and the objections, the application aimed to provide a platform that quickly transcribe guitar audio into tablature and allow the ability to edit and correct the chords, which ultimately save time from creating the resources from scratch.

The main component that required the most attention was the ML component. The user needed to be able to upload their audio and let the back end completely handled the painstaking task of transcribing the tab. This required the model to understand the polyphonic nature of guitar sounds and output the predicted notes that were played by the guitarist. Once the prediction was complete, the output needed to be compiled in such a way that the result could be imported and understood by the tab visualiser – which is the front end of the application.

4.2 Program Design

4.2.1 Technologies

TensorFlow

TensorFlow was a DL library that would be utilised to train/test the model. It was often the go-to choice for creating models for classification problems similar to another ML library called PyTorch. The library would be utilised to train a DL model to detect note(s) in a song in each consecutive time frame (in milliseconds). The model would be used to detect notes and chords, which would output the prediction to the Front-end.

Google CoLab

Google CoLab was a virtual machine service for users with Google accounts to freely utilise to write Python codes on the cloud. This eliminated the needs the computation power needed for training a large dataset model as it had been provided by the service itself.

Librosa

A python package for music and audio analysis. It provides the building blocks necessary to create music information retrieval systems. The package would be utilised to extract features such as the tempo, onsets, musical key, and beat tracking. The package served as the base for the application as it provided powerful analysis of MIR.

React JS

React was a JavaScript framework commonly utilised to implement a Single-page application. This easily updates to easily changes the DOM without the need to refresh the page. The framework was decided based on the familiarity and the hands-on experiences as the project required short development deadlines and research heavy features that were needed to implement.

AlphaTab

AlphaTab was a crucial JS library that allowed the author to fulfil the major features of the frontend component. The library initiated a visualiser to import a MusicXML file and display a guitar tablature as a musical sheet. In addition, the functionalities such as MIDI player, file exports, playbacks, and note selections were supplied out of the box, which efficiently saved the author’s time.

Azure Web Services

Azure was a Software-As-A-Service (SAAS) provider which benefit the research in deploying hosted services such as WebApp, Container App, and Blob Storage. The cloud provider offered the point-and-click ability to easily create a web application, store data, and deploy containers all in one site.

MongoDB

MongoDB was a cross-platform, document-oriented NoSQL database system that allows for flexible and scalable data storage. Data is stored in flexible, dynamic documents that can vary in structure from one to another. This allows for a more natural representation of data and makes it easier to store and query complex data structures.

Terraform

Terraform was an infrastructure-as-code (IaC) tool that allowed user to declaratively configure infrastructure through code. This software would be utilised to automatically create instances for former storage service in AWS S3, and Azure Blob Storage where the author would only need to take care of the code which share similar structures across each service provider, rather than creating the service through the website that cloud providers had created where the UI and naming conventions were different across each platform.

4.2.2 Design Patterns

A design pattern was a solution that provide repeatability to common issues in the development of various architecture such as web development. In particular, the design patterns in React web development, where each component would generally be used to resolve issues corresponding to rendering pages, passing down props and context managements. The following design patterns below would be utilised for constructing the Front-end application:

Structure of React

The folders in Figure 13 illustrated the common file structure for React application when they were first initialised. These were known as the standard of React and their design pattern that the developers would follow:

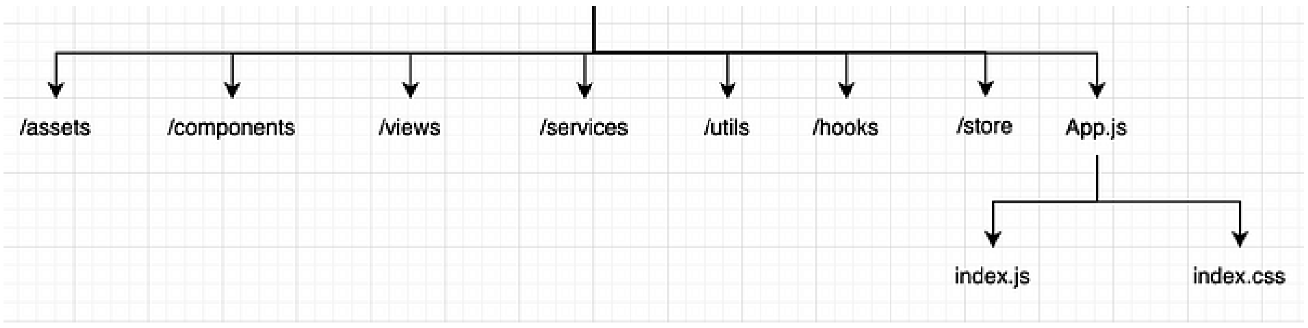


Figure 13 illustrated the directory of React files.

Assets

The folder in the assets often contained resources of images, and CSS files, where sub-folders may be present to group and organise the assets as needed by its group names.

Components

The components folder would house the React components i.e. navigational bar, inputs, buttons, and so on. These components would typically be created to be used multiple times throughout the application and stored in this folder for utilisation in pages that required it.

Views/Pages

The views/pages folder would typically act as the parent component to store its child component shown in Figure 14 i.e. a list component that would use its data props to display information for each data.

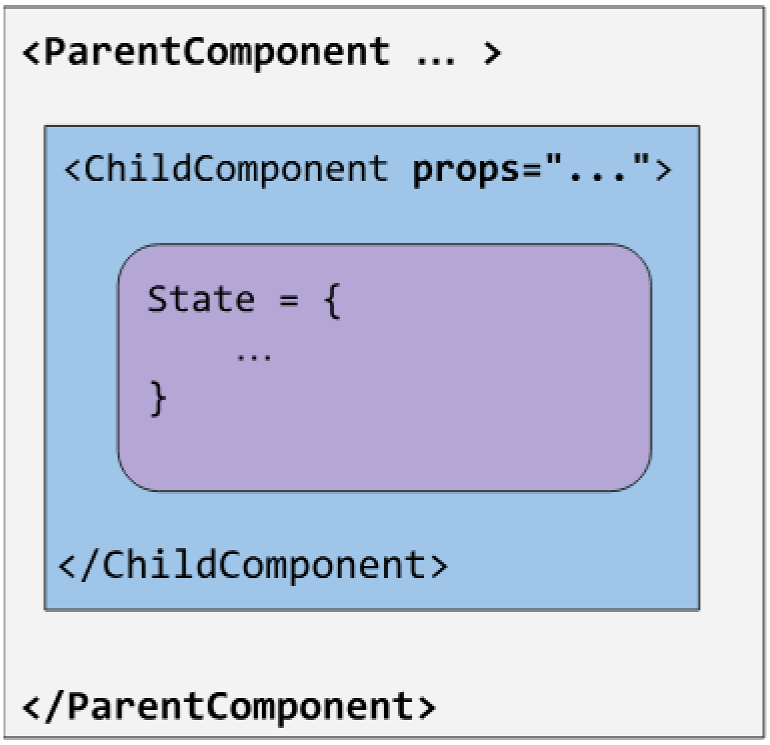


Figure 14 illustrated the structure of the parent component.

Hooks

This folder contained custom hooks created to satisfied or ease the issue, for instance, fetching data through the standard Fetch API of JavaScript could be cumbersome and repetitive due to the amount of time a developer would have to convert the retrieved request to JSON format before passing the final data and assign it to a useState variable.

4.2.3 Application Structures

The front end of the application was created using the MERN stack, which was a common web development stack to build web applications. A stack typically consisted of a front-end framework to handle the UI and UX, a back end to handle the database and create paths to access data, and the server to direct the traffics to the web. The stack utilised MongoDB, Flask, ReactJS, and NodeJS to execute the stack.

Development Stack

MongoDB
MongoDB was a NoSQL service that stored data as a JSON format. The utilisation of the service let the data of users, songs, roles, and favourites to be stored and easily retrieved in the front end. Due to the free credits given by the GitHub Educational Starter Pack, MongoDB Atlas could be utilised to store data using their Serverless package to automatically scales to meet the current workload of the Flask application.

Flask
Flask was a Python framework that provides the ability to write server on Python. The framework had the same functionality as ExpressJS, which enabled the author to serve Application Protocol Interface (API) endpoints to the client with the compatibility of Librosa, a package that irreplaceable in attain musical information.

ReactJS
As mentioned in the previous section, ReactJS hindered the static process of updating the DOM without the user needing to refresh the page. The framework would house the entirety of the front end and fetch data from the Back-end (Flask).

NodeJS
NodeJS was a runtime that execute JavaScript code on the client and were often utilised to develop a web application.

The Application Architecture
The diagram shown in Figure 15 illustrated the initial overview of the architecture and the interactions of between each component in the subnet. Essentially, the user utilising the application would upload an audio to through the AWS API Gateway, where the request would trigger a Lambda function on AWS Lambda to extract music information and predict the tablature from a Docker container hosting a Flask app on an EC2 instance. Once the prediction was completed, the user would be navigated to the tab visualiser to view and edit the music score through the use of AlphaTab library.

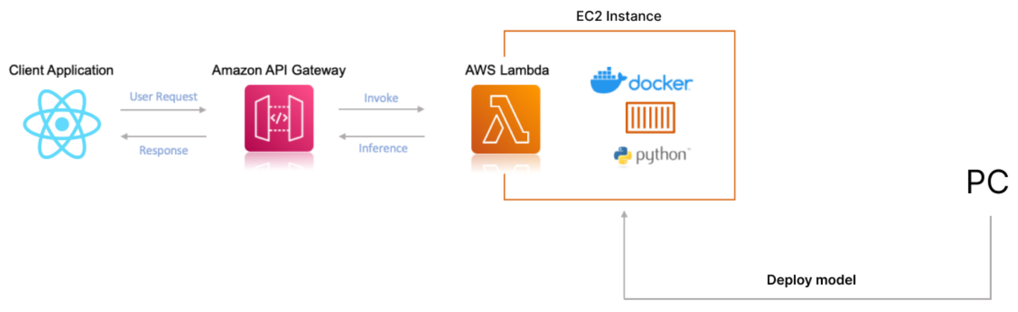


Figure 15 illustrated the initial architecture of the application.

Despite the designated architecture, AWS Lambda forced the author to redesign the architecture due to the complication caused by hosting a Flask app with Librosa library installed, where the library required another C++ library for the library to function as intended. A library, of which, the serverless function or AWS Lambda was not supplied with according to Yvesonline (2021). Therefore, the deployment of the hosted AI model was impossible to be implemented using the current structure.

A new architecture in Figure 16 was developed to better reconcile with the previous issue. Albeit the interactions between each component remain the same, the cloud service provider was changed to Azure due to the free credits supplied by GitHub educational pack. The difference between the two architectures were the change in the server, where instead of using an equivalent of AWS Lambda to handle API requests, the Flask app would handle these endpoints for the application.

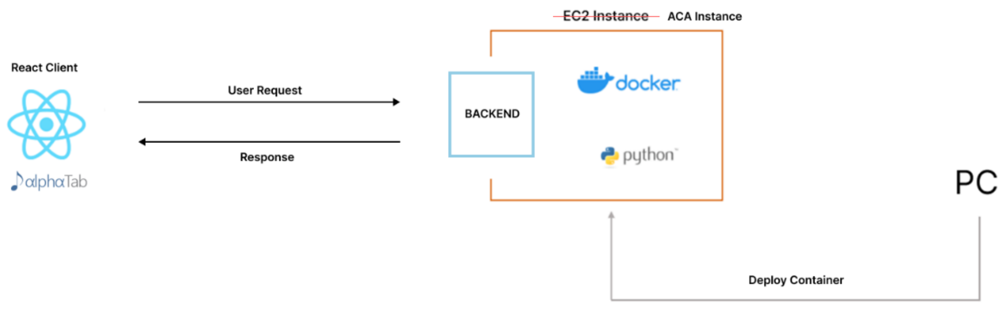


Figure 16 illustrated the architecture of the application.

4.2.4 Database Design

The Figure 17 illustrated the Entity Relationship Diagram (ERD) where the tables were constructed with the syntax of the relational database – an Structured Query Language (SQL), though the actual application would be utilising the architecture of NoSQL ,i.e. MongoDB, to warehouse the data but the ERD was used to provide a better visualisation of the idea.

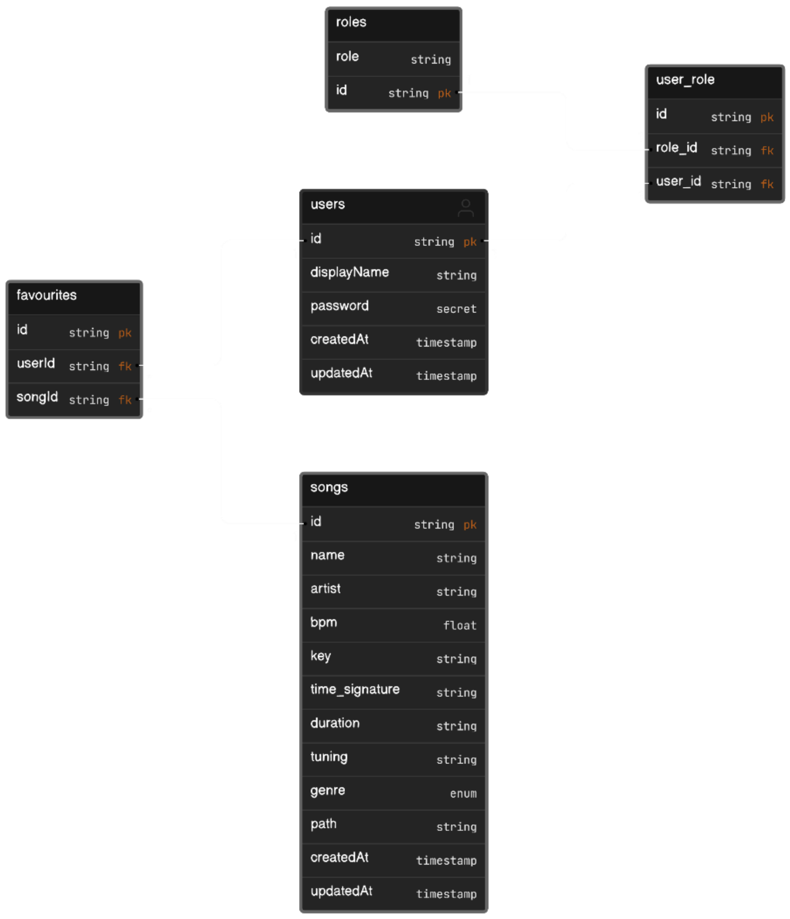


Figure 17 shows the ERD of the web app.

4.3 Process Design

4.3.1 Pseudocode

Pseudocode was a method to provide a simple, yet efficient way to describe computer programs and/or algorithms at a high-level. This allowed for the expression of programs in plain and simple words that were uncomplicated to follow and comprehend. The implementation of pseudocode was usually written in an early stage of development to visually examine the important functional requirements that needed to be prioritised. It allowed developers to directly explain the process of a program to someone who might not be proficient in the area. The requirements required for this application were:

1. The user would have an option to create an account or continue as a guest.
2. The user’s login details will be stored on a database, they will now be able to log in.
3. The user would be able to search for songs.
4. The resulting song would be shown, and the user would be able to visualise the guitar tab.
5. If the song did not exist, the option to upload the song would appear, which would upload the song to the database and transcribe them.
6. The audio of the song would be sent to a DL model through an API endpoint to predict the chords and riffs.
7. The API would response with a file (of XML extension) that could be readable by the front-end application.
8. The procedure in step 4 would occur.
9. The logged-in user would have the option to favourite the song for later practices.

4.3.2 Flowchart

The flowchart in Figure 18 had shown the visual representation of the pseudocode illustrated in the previous section.

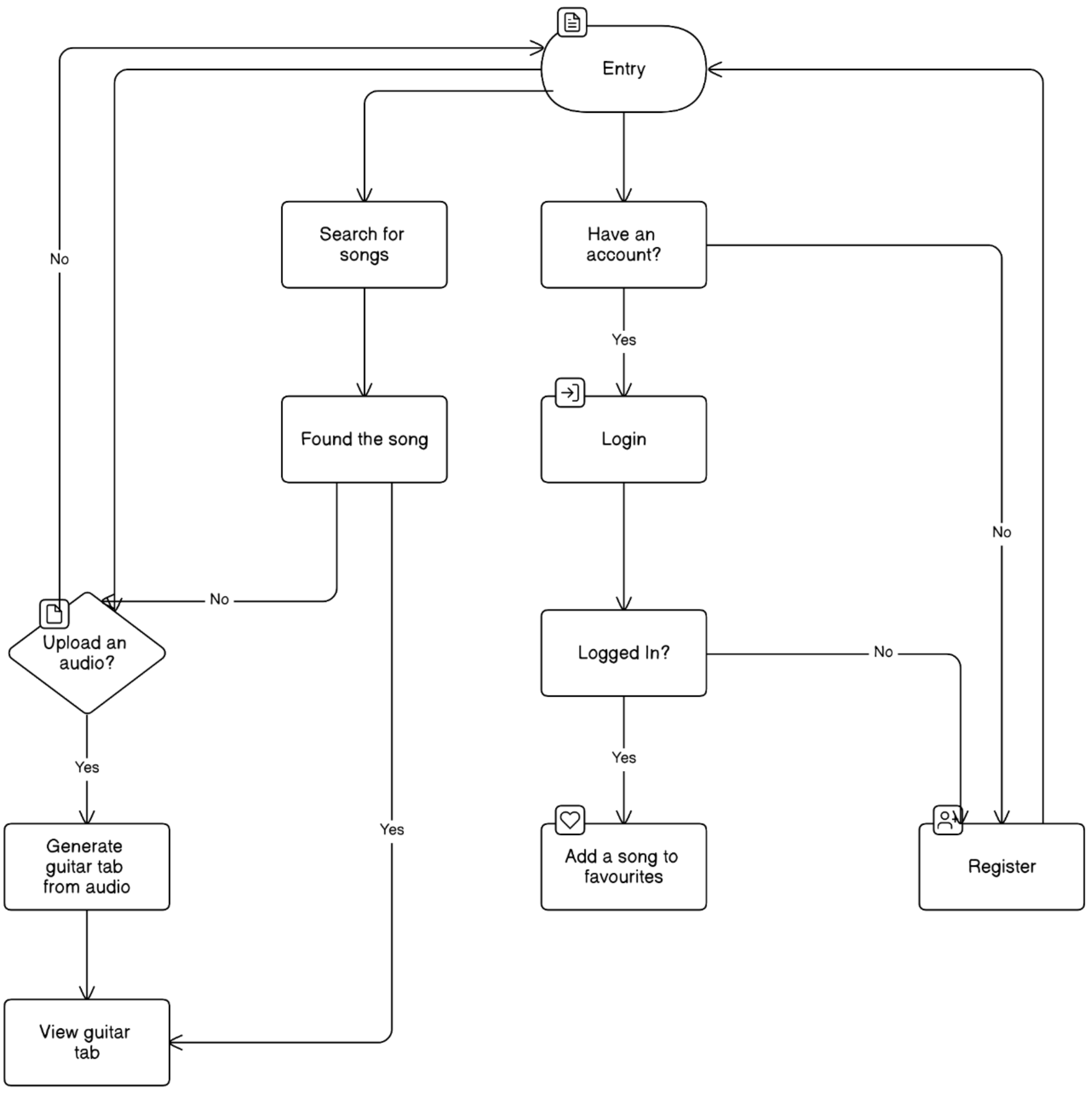


Figure 18 shows the flowchart which exercised the possible combination a user could execute.

4.4 User Interface (UI)

4.4.1 Wireframes

The first iteration of the wireframe was done through analysing potential competitors' strengths and weaknesses in order to extract the most important features and implement any features the author felt were missing in the existing applications. The results of the analysis could be seen in Figure 10 and 11.

Paper Prototype

The paper prototype had been developed to realise the functionalities that were needed to ease the user experience in browsing the web application. The Figure shown in 19, illustrated two possible iterations of web structure, which included the home page that a user could drag and drop audio for guitar transcription and search through recent uploaded songs, the result page that let the user visualised guitar transcription, the login page, and the favourite page for users to favourite their songs for later practice.

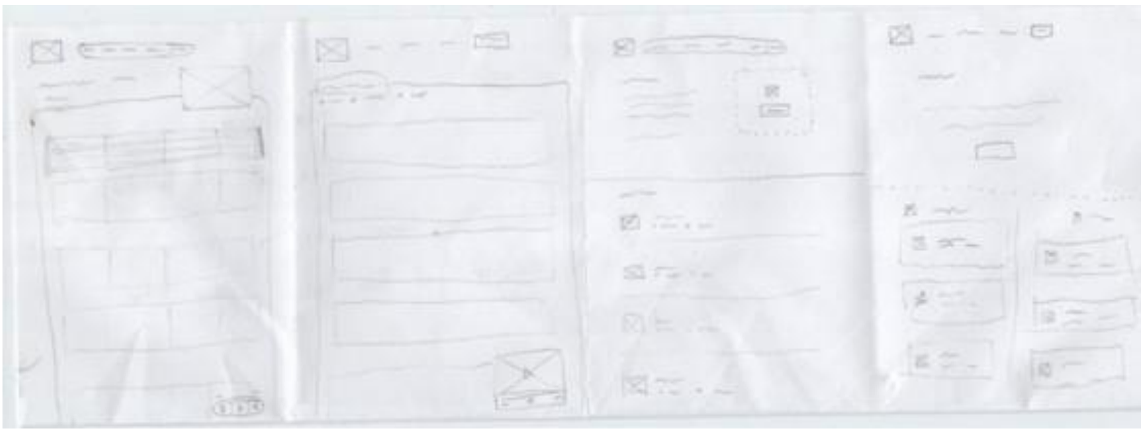


Figure 19 illustrated the initial paper prototype.

Lo-Fi Prototype

The first iteration of the application was chosen due to its simplicity and easy-to-navigate layout. The outline of the lo-fi prototypes was to check the usefulness and the flow of the whole application. The wireframes illustrated in Figure 20-23 showed the proposed functionality of the paper prototypes.

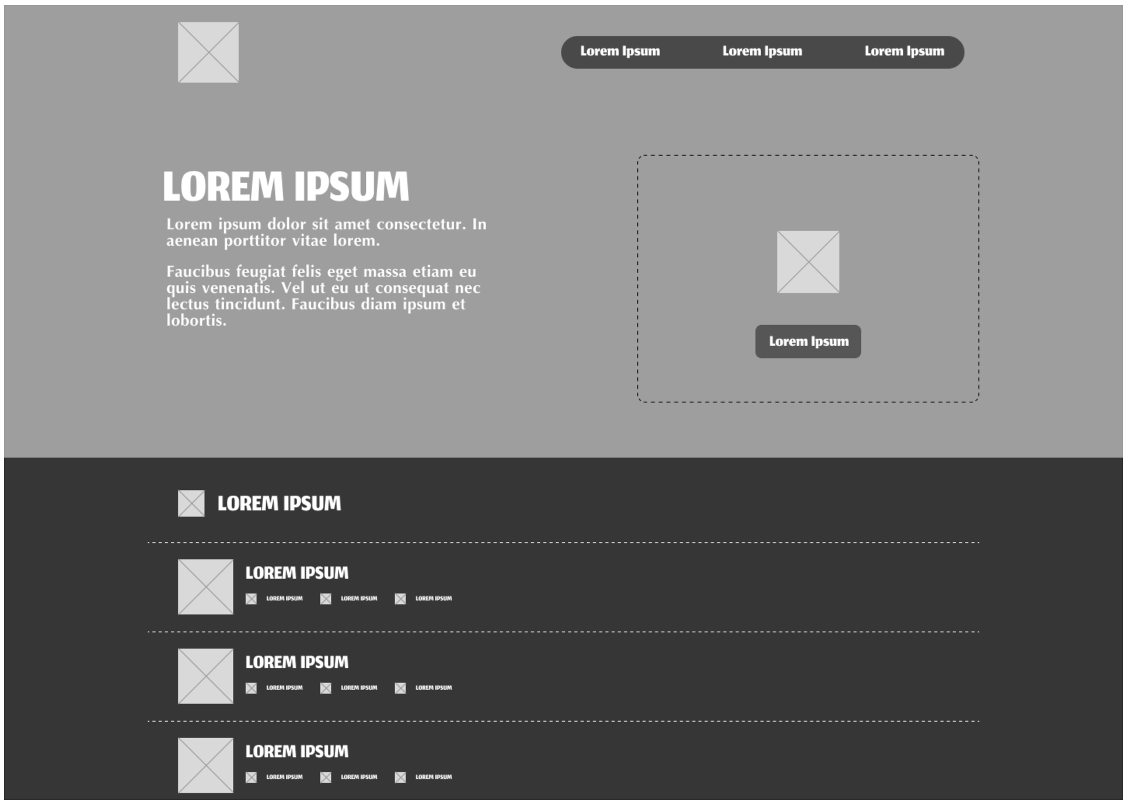

Lorem Ipsum
Lorem Ipsum
Lorem Ipsum
LOREM IPSUM
Lorem ipsum dolor sit amet consectetur. In aenean porttitor vitae lorem.
Faucibus feugiat felis eget massa etiam eu quis venenatis. Vel ut eu ut consequat nec lectus tincidunt. Faucibus diam ipsum et lobortis.
Lorem Ipsum
LOREM IPSUM
LOREM IPSUM
LOREM IPSUM
LOREM IPSUM

Figure 20 shows the Lo-Fi prototype of homepage.

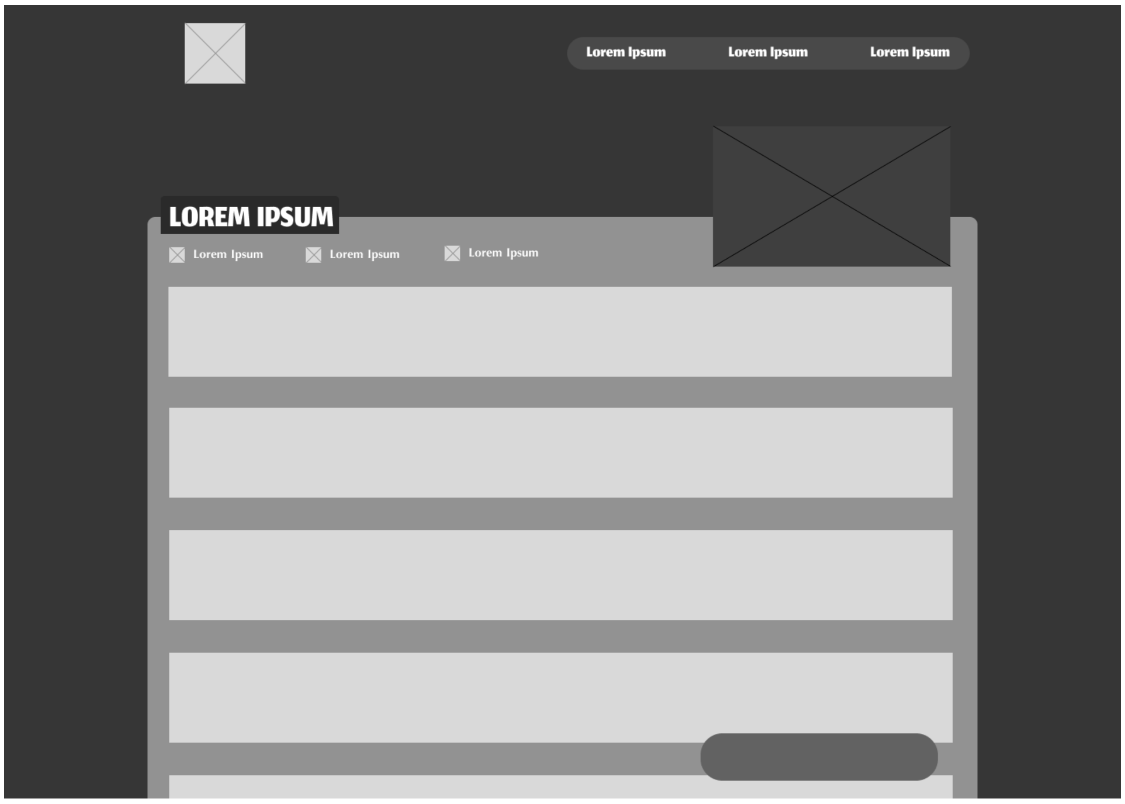


Figure 21 shows the Lo-Fi prototype of result page.

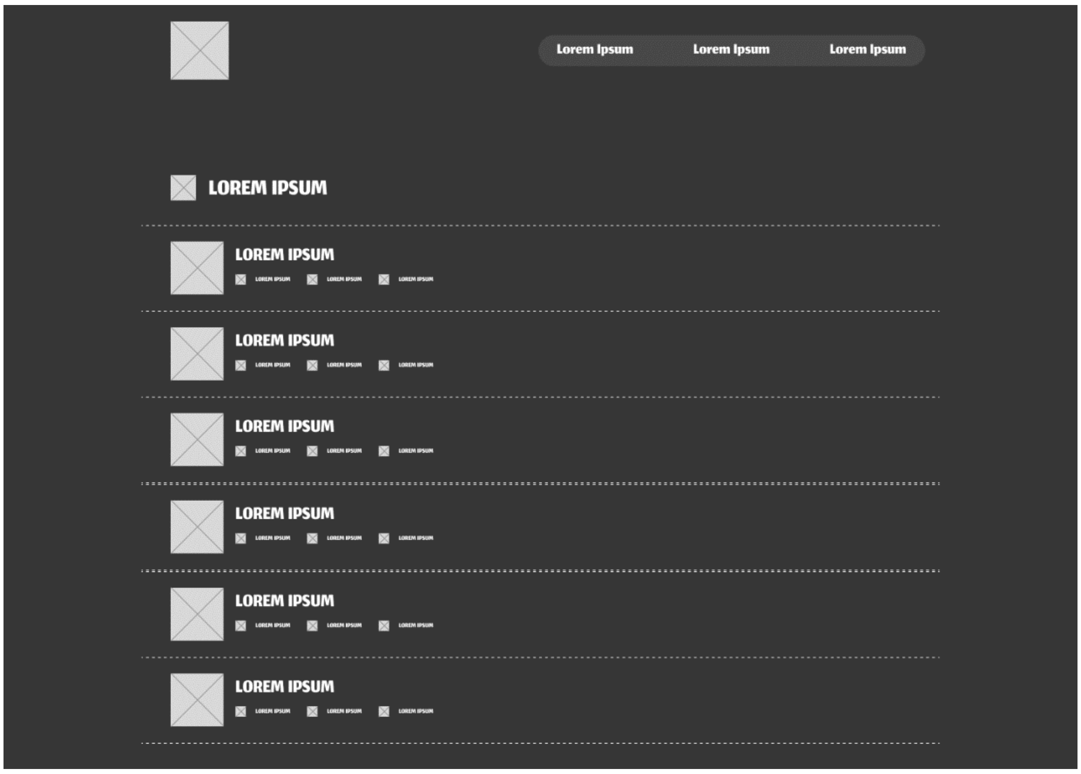


Figure 22 shows the Favourited page.

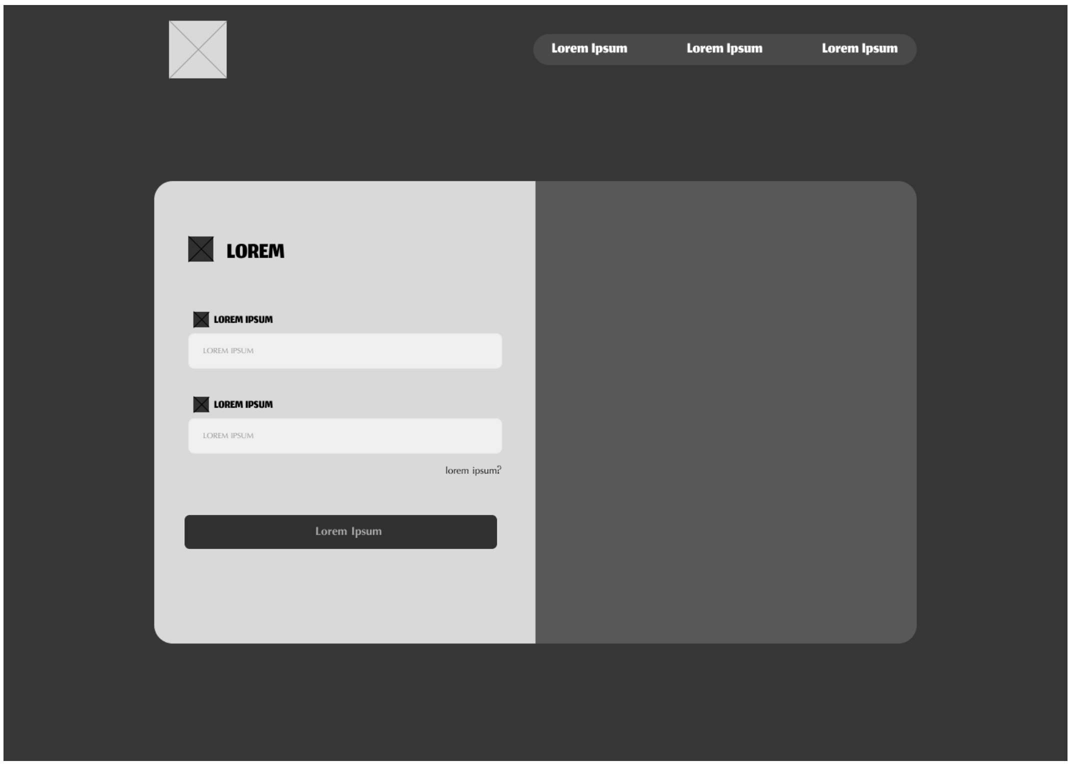
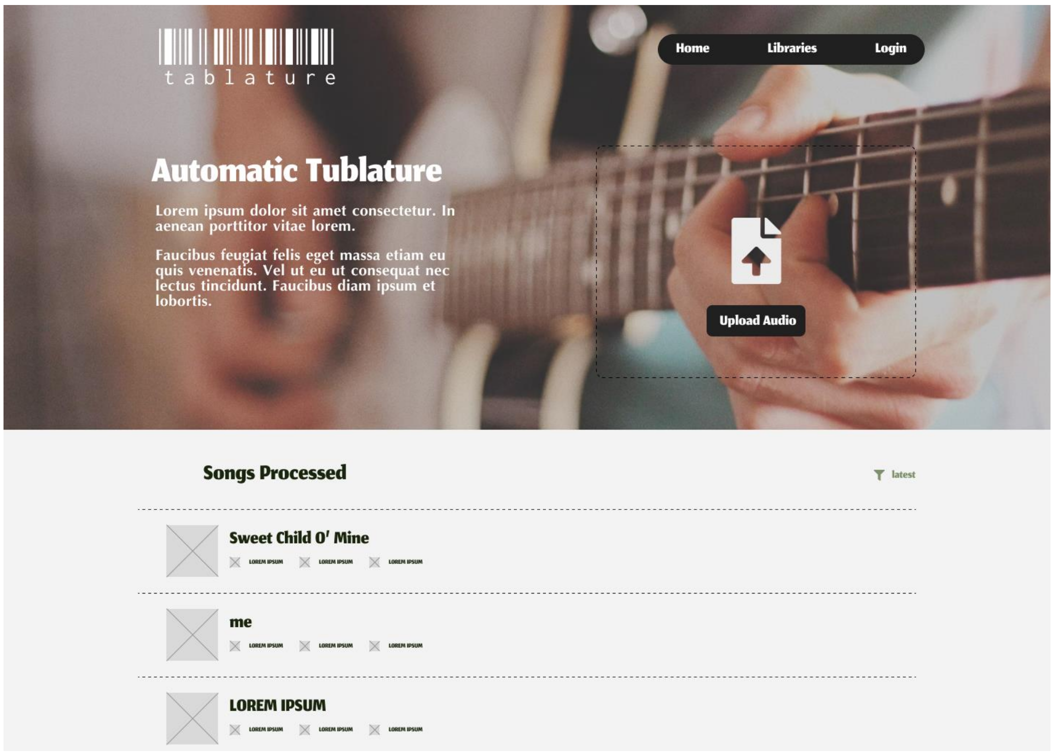


Figure 23 shows the Login page.

Hi-Fi Prototype

From prototyping a simple concept of the application to quickly gained an initial insight into the final product had efficiently contributed to fine-tuning the final version. The Hi-Fi prototype illustrated in the Figure 24-27 where the author had settled with the mood and tone of the application, finalising the layout, and adding the login feature to allow for the replay ability of the saved songs.



Lorem ipsum dolor sit amet consectetur. In aenean porttitor vitae lorem.

Faucibus feugiat felis eget massa etiam eu quis venenatis. Vel ut eu ut consequat nec lectus tincidunt. Faucibus diam ipsum et lobortis.

Figure 24 shows the Hi-Fi Prototype of the homepage.

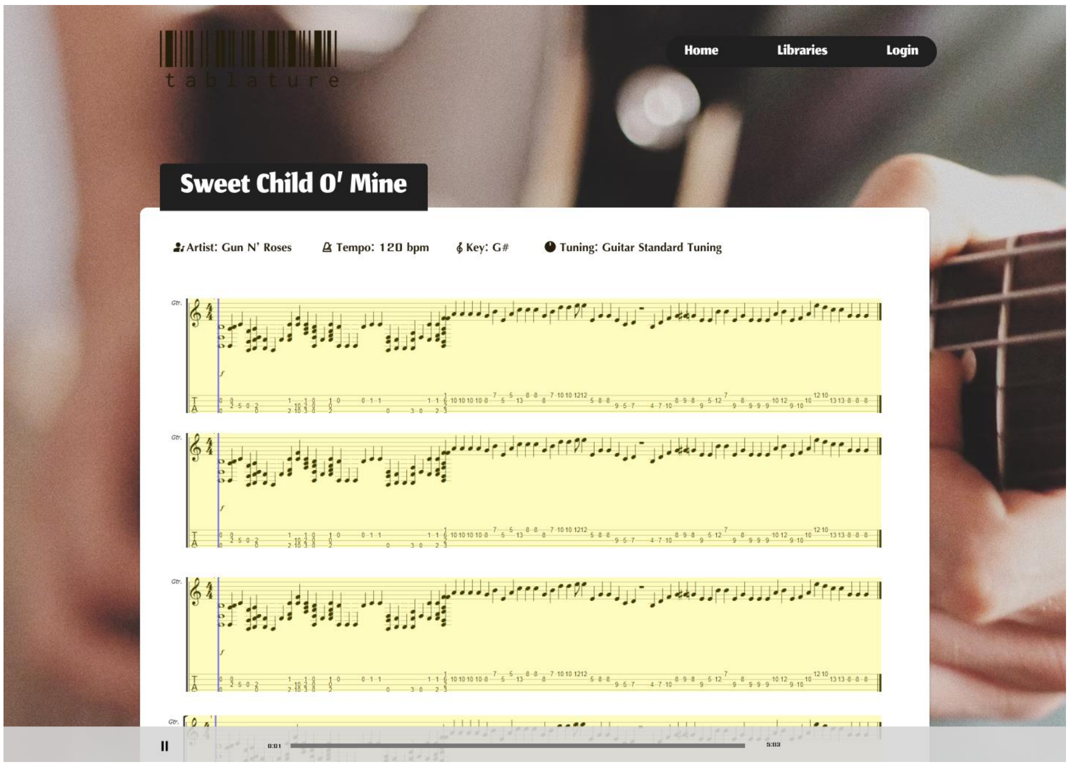


Figure 25 shows the Hi-Fi prototype of the result page.

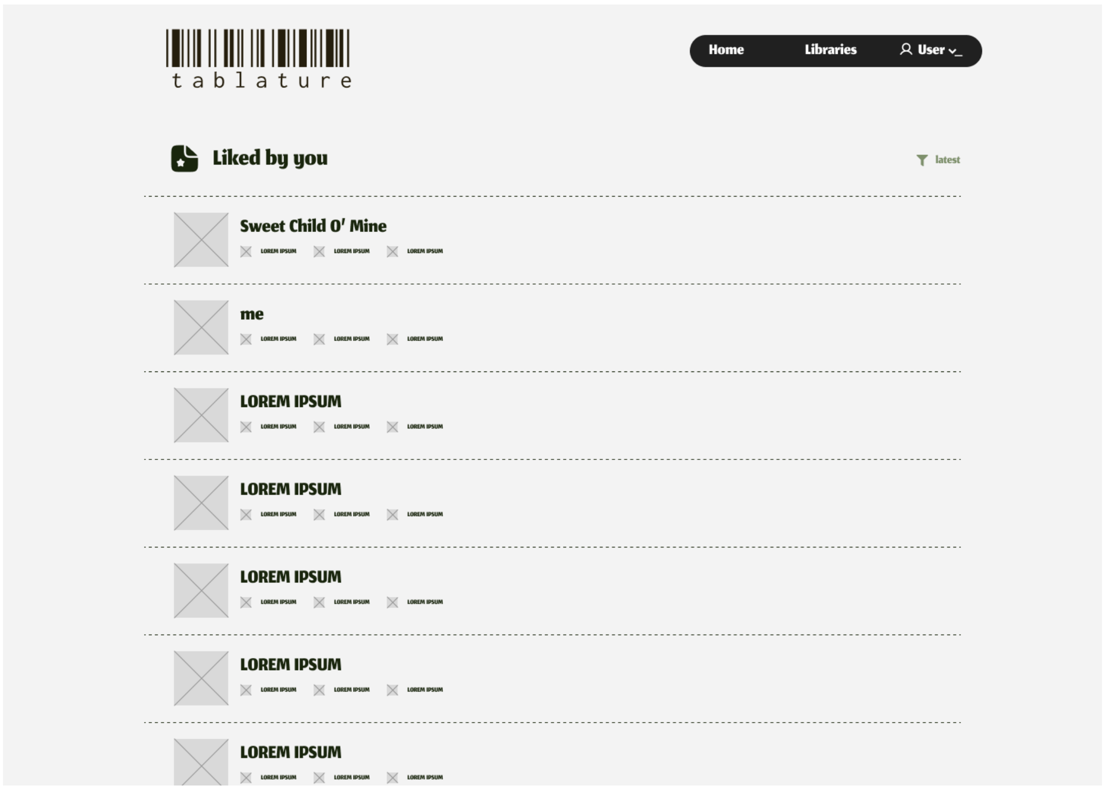


Figure 26 shows the Hi-Fi prototype of the favourites page.

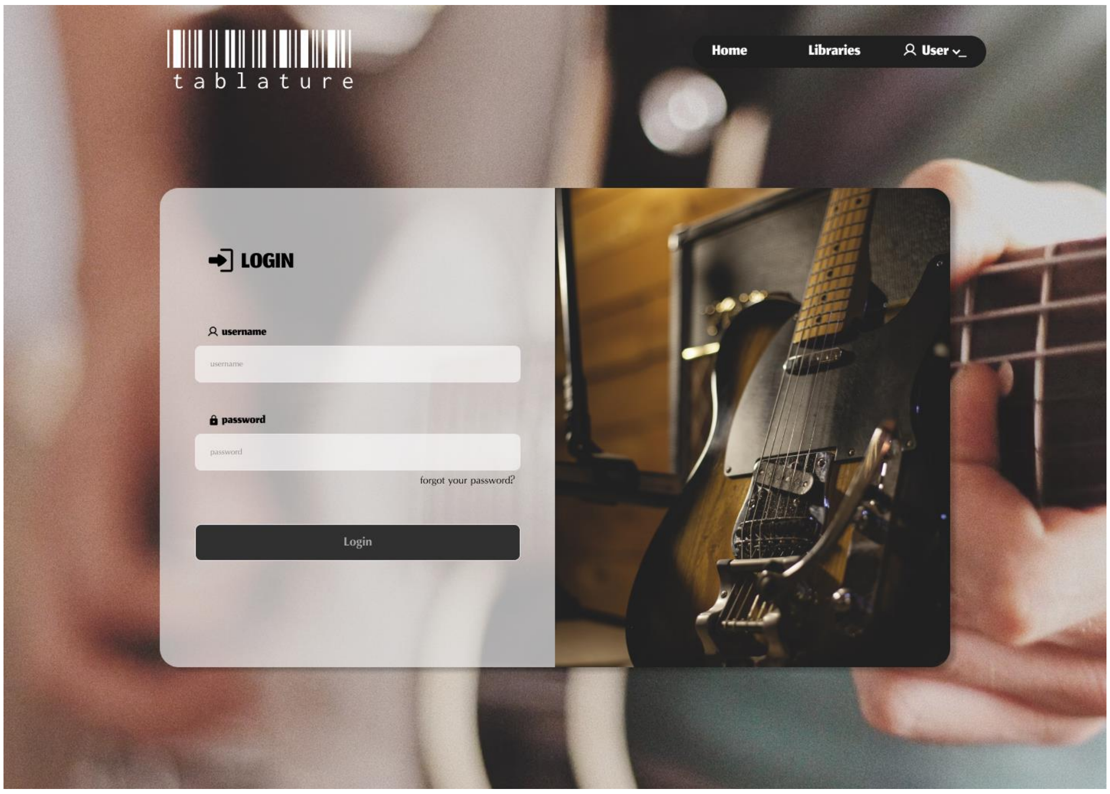


Figure 27 shows the Hi-Fi prototype of the login page.

4.5 Conclusion

Through each iteration from paper to Hi-Fi prototyping had allowed the author to identify issues that could be avoided during development process. For instance, the application was designed with prior inspiration from existing websites i.e. Spotify, YouTube Music, Chordify, and MVSEP. The features that existed on these applications were thoroughly studied to integrated into the author's own application.

5 Implementation

5.1 Front-end Development

5.1.1 React Router Dom

React Router Dom handled routing for SPA, as the integration with React framework allowed the component pages to mount/unmount seamlessly without actually having to refresh the page, instead the content would be dynamically fetched based on the URL. This was the reason React Router was the most appropriate decision to utilised in creating React applications.

Additionally, a rather handy feature supplied by the library was incredibly helpful due to the implementation of the prototype, where a response needed to be passed to the adjacent page once the audio has been processed on the Backend such as the name of the file, its tempo, key signature, duration, time signature, and file locations. Fortunately, the states could be sent through both the Navigate function and Link component by passing the state(s) in the state object. This very feature would be implemented in the homepage, login/sign up, and register pages to cache the state to the next/previous page once a response had been received to either show retrieved data or errors from the API endpoint(s).

5.1.2 AlphaTab

AlphaTab was a crucial JS library that allowed the author to fulfil the major features of the frontend component. The library contributed to initiating a visualiser to import a MusicXML file to display a guitar tablature. In addition, the functionalities such as MIDI player, file exports, playbacks, and note selections were supplied out of the box, which efficiently saved the author's time.

5.1.3 Tailwind

Tailwind was a utility-first CSS framework which provide fast and straightforward method to build unique UIs. The framework offered the ability for developers to create their very own components using classes in the markups. Instead of utilising pre-existing UI frameworks, such as Bootstraps or Material UI, which had already designed components and restricting the flexibility of the design system (colours, spacing, typography etc.).

5.1.4 Typescript

Typescript was essentially JavaScript with strongly typed syntax. This meant that the language allowed the specifications of types for the data that were passed around within the code, and report errors when the types didn't match. Although, additional typing could increment development time, the language allowed for typed safe environments and avoid potential bugs that could occur at runtime.

5.1.5 Azure Web App

The entirety of the front-end would be hosted online using Microsoft's cloud provider called 'Azure', where a service named 'Web App' could be utilised to host the static React app that has been firstly compiled using GitHub actions. Through the use of the actions, the application could be set up for automations such as compilation to static files, and unit testing. This would save a tremendous amount of time where check-ups could be tested prior to launch and the static app could be automatically deployed to the web without the developer having to manually add files to the cloud.

5.2 Back-end Development

5.2.1 Flask

Flask was a framework for Python, a server-side Python runtime environment. Using Express, a backend server for the application was created. This server would handle functionality such as creating and storing a user's profile, allowing users to send and receive friend invitations as well as handling the functionality of chat rooms and direct messages. The backend server also handles user authentication, restricting users from using the application if they were not registered or currently logged in.

5.2.2 **M**ongoDB

MongoDB was a cross-platform, document-oriented NoSQL database system that allows for flexible and scalable data storage. Data is stored in flexible, dynamic documents that can vary in structure from one to another. This allows for a more natural representation of data and makes it easier to store and query complex data structures.

5.2.3 HashLib

HashLib was a Python package that allowed a user to convert plain text to an encrypted format. The package would be utilised to store a user's password, as sensitive information within a database should not be shown or seen by the public.

5.2.4 JWT

JSON Web Token (JWT) was a Python package used for generating web tokens used for authentication. When a user signed in to the application, they would be provided a token, of which could be used to access otherwise restricted sections of the application.

5.2.5 Docker

Docker was a software that allowed applications to be delivered in packages called containers. The containers were especially useful for running different applications on the same machine without the need to worry about clashing dependencies or conflicts. Due to its ability to simplify configuration, both the environment and configuration could be encapsulated into code, wherein the time and effort needed to set up and synchronise environments across different machines could be significantly reduced.

5.2.6 Azure Container App

Azure Container App was a service that would be utilised to house the Docker container. This allowed easy configuration and deployment automation once the container was pushed to Azure Container Registry.

5.2.7 Azure Blob Storage

Azure Blob Storage was a service that allow files to be stored on it, essentially act as a file storage for audio and MusicXML files.

5.3 Functionality of Application

5.3.1 Login/Register

The login and register functionalities were needed to keep track of changes made to the music sheet by the users. The pages, shown in Figure 28, would allow them to enter their credentials to login or click a link to navigate to the register page. Both pages had forms with validations to check for correct username/passwords, and syntax. These validations are specified both in the Front and Back end of the application to ensure security and order for both the users and the application itself. These validations verified the user against their username, email, and password to assert and maintain rules that standardise the data stored in the database. Once all validation had been verified, the page would be navigated to the home page with the login state attached for later usage such as token authentication for editing music sheet, favouriting songs, and listing user's owned music sheet(s).

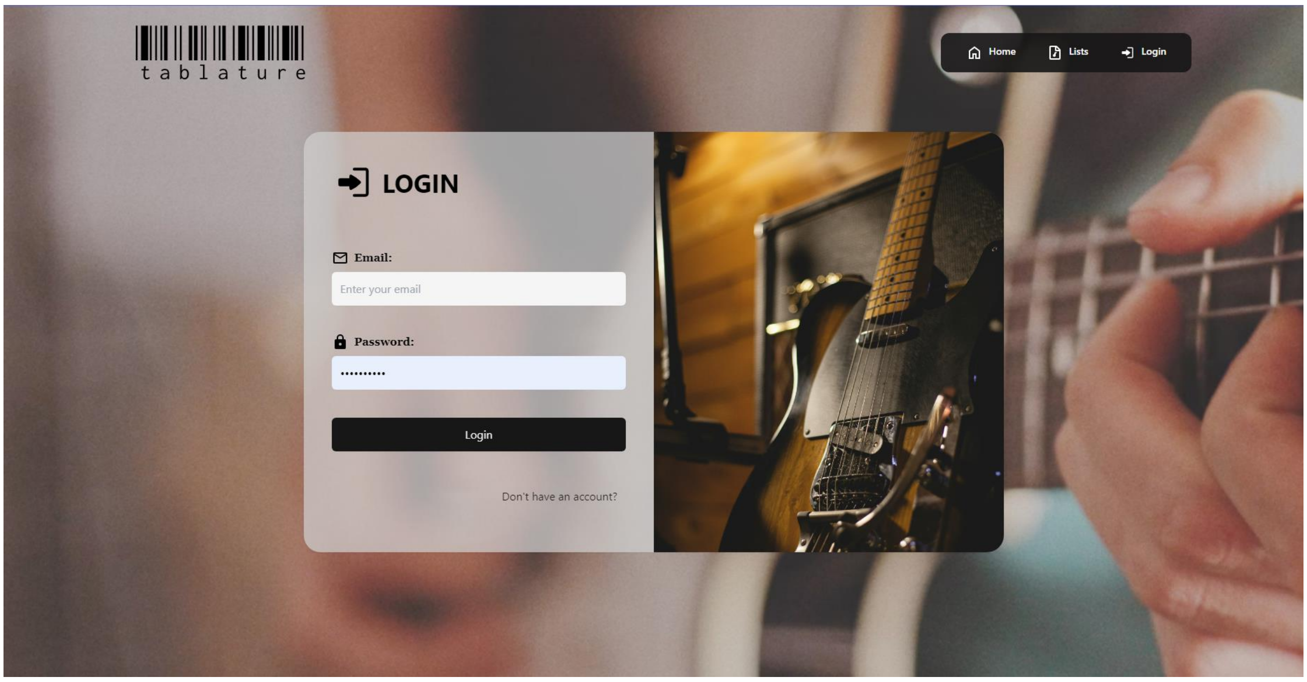
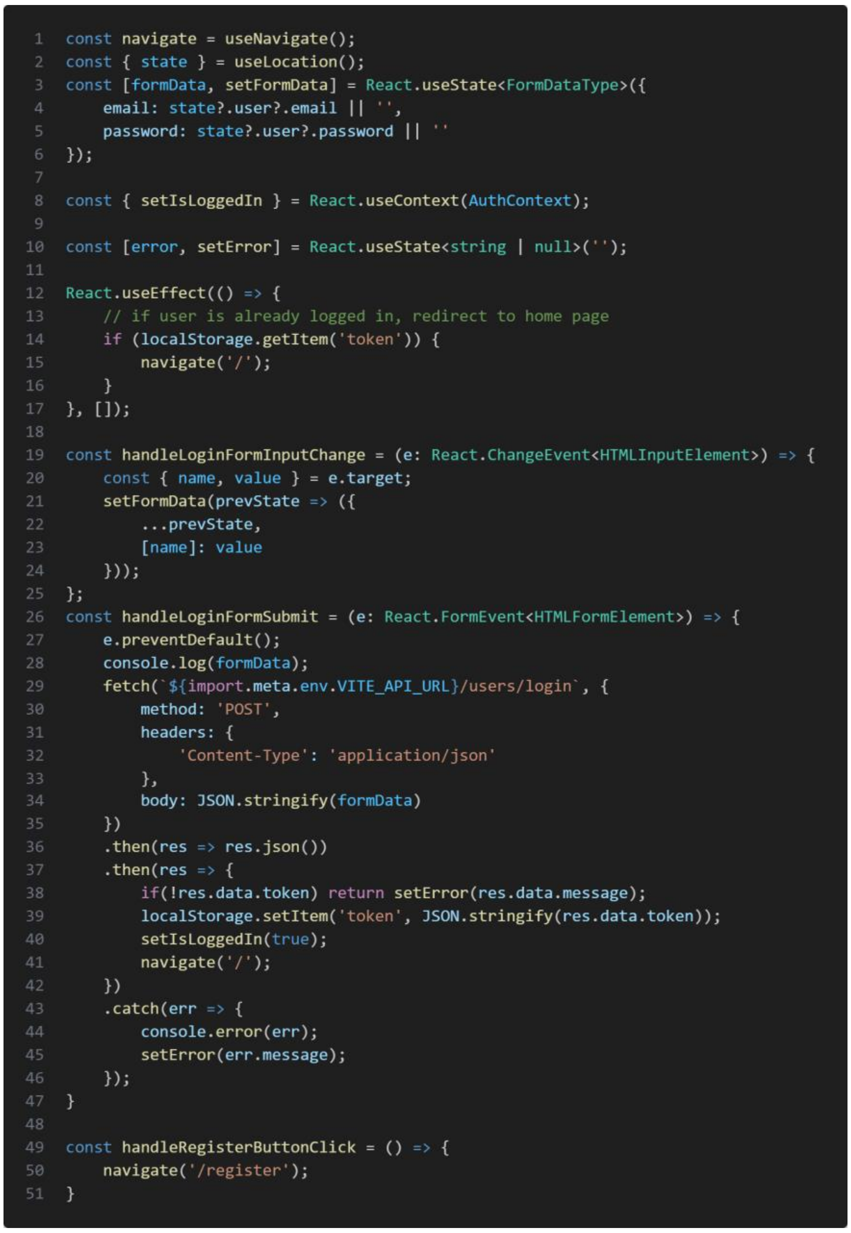


Figure 28 illustrated the Login page of the application.

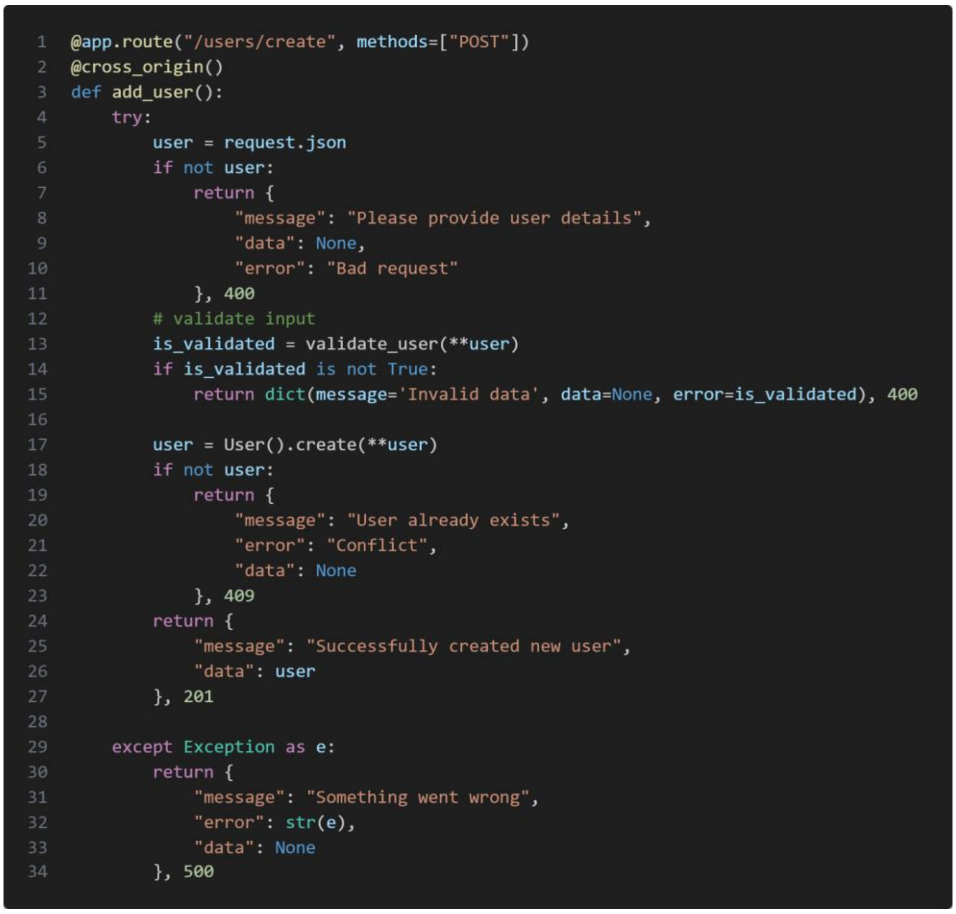
If the user attempted to log in to the application, a request would be sent to the back end of the application to validate their truthiness. This meant that the email input would be compared to the current email in the database to ensure they are the owner of the account, and vice versa for the password. In the situation where no match was found in the database, the server will return a corresponding HTTP code with the messages to provide the best possible user experience and allowed for the user to better debug the error(s). Otherwise, if the email and password matched, a bearer token would be created and returned as a response to the client alongside their credentials and stored as a state on UseContext for components to be easily accessible. The client would then utilise the token to access the server to execute and access user-privileged routes and functionalities illustrated in Figure 29.



```
const navigate = useNavigate();
const { state } = useLocation();
const [formData, setFormData] = React.useState<FormDataType>({
    email: state?.user?.email || '',
    password: state?.user?.password || ''
});

const { setIsLoggedIn } = React.useContext(AuthContext);

const [error, setError] = React.useState<string | null>('');

React.useEffect(() => {
    // if user is already logged in, redirect to home page
    if (localStorage.getItem('token')) {
        navigate('/');
    }
}, []);

const handleLoginFormInputChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    const { name, value } = e.target;
    setFormData(prevState => ({
        ...prevState,
        [name]: value
    }));
};
const handleLoginFormSubmit = (e: React.FormEvent<HTMLFormElement>) => {
    e.preventDefault();
    console.log(formData);
    fetch(`${import.meta.env.VITE_API_URL}/users/login`, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(formData)
    })
    .then(res => res.json())
    .then(res => {
        if(!res.data.token) return setError(res.data.message);
        localStorage.setItem('token', JSON.stringify(res.data.token));
        setIsLoggedIn(true);
        navigate('/');
    })
    .catch(err => {
        console.error(err);
        setError(err.message);
    });
}

const handleRegisterButtonClick = () => {
    navigate('/register');
}
```

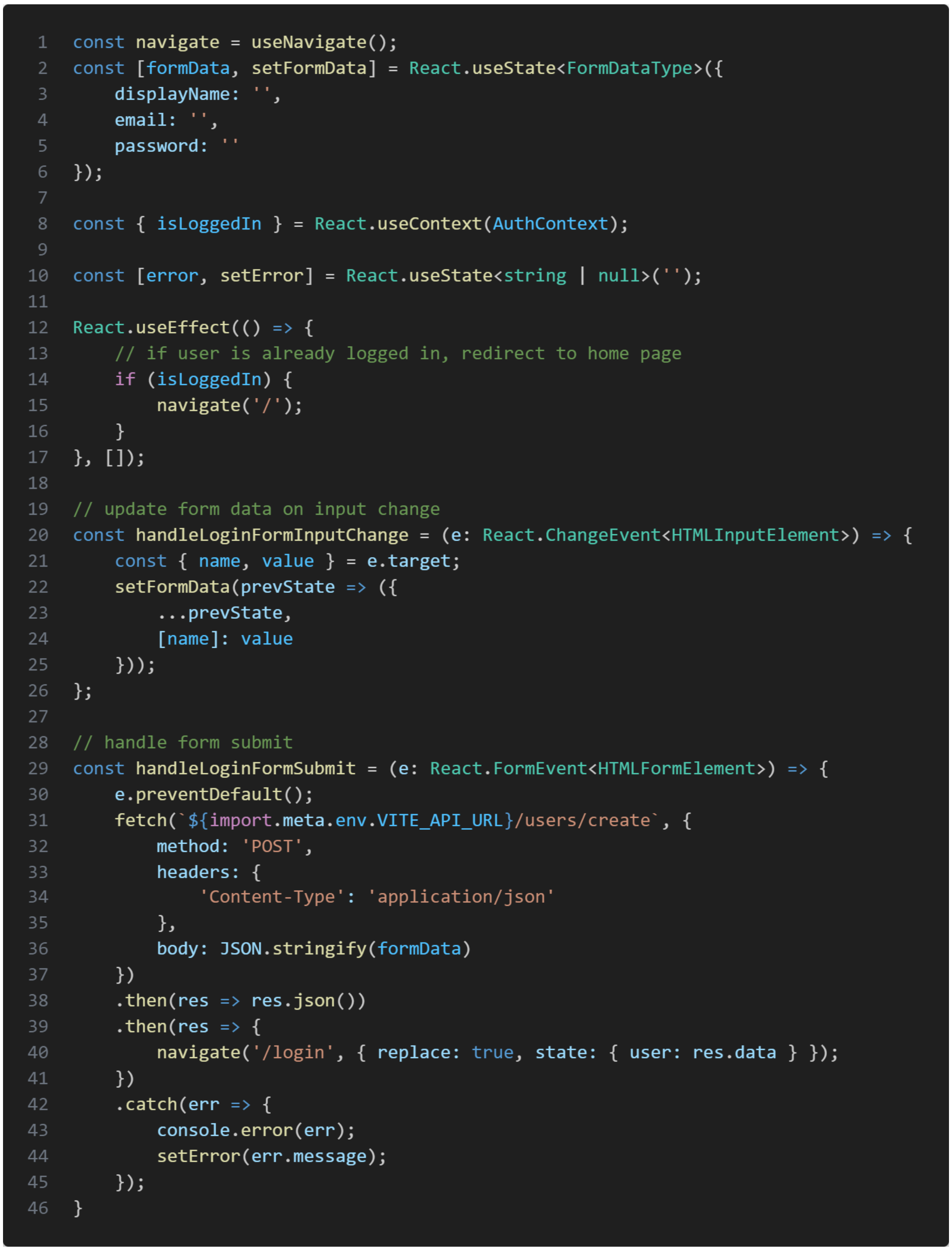
Figure 29 illustrated the Login Page On the login page, when the user presses the login button, the information will be sent to the backend server of the application to verify the information in the inputs.

Though, the registration page shared similarities with the login page, they differentiated in the way information were handled in the server side. Once information had been sent, the same validation from the Front-end would take place on the server side with the additional email comparison to look for possible existing email with the same value on the database to ensure that each email was unique shown in Figure 30. If the server found the email in the database, a HTTP 409 response would be attached to the error massage to elaborate to the user that the request could not be processed due to a possible conflict in the request. If the email did not exist, the user’s password would be encrypted using HashLib, then the user’s username, email, encrypted password, and their role would be stored in the database. As a final process, the token would be created and returned with the user's credentials. Similar to the Login page, the response would then be stored in a UseContext and the user would be navigated to the homepage as shown in Figure 31.



```python
@app.route("/users/create", methods=["POST"])
@cross_origin()
def add_user():
    try:
        user = request.json
        if not user:
            return {
                "message": "Please provide user details",
                "data": None,
                "error": "Bad request"
            }, 400
        # validate input
        is_validated = validate_user(**user)
        if is_validated is not True:
            return dict(message='Invalid data', data=None, error=is_validated), 400

        user = User().create(**user)
        if not user:
            return {
                "message": "User already exists",
                "error": "Conflict",
                "data": None
            }, 409
        return {
            "message": "Successfully created new user",
            "data": user
        }, 201

    except Exception as e:
        return {
            "message": "Something went wrong",
            "error": str(e),
            "data": None
        }, 500
```

Figure 30 illustrated the validation done in the server side when the user was signing up.



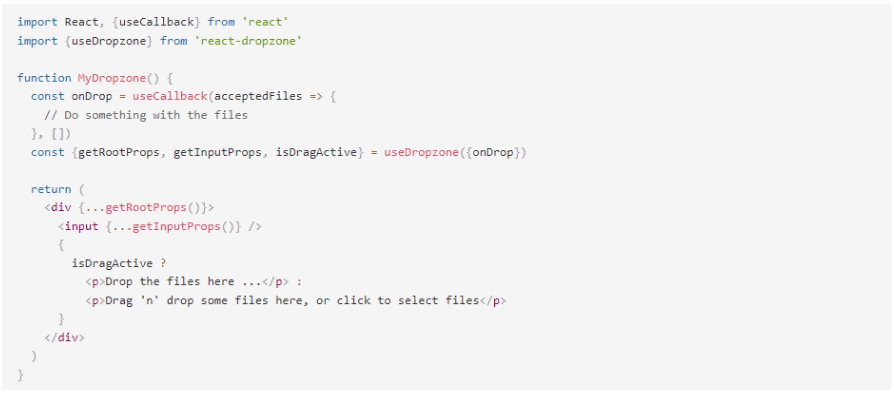
```
const navigate = useNavigate();
const [formData, setFormData] = React.useState<FormDataType>({
    displayName: '',
    email: '',
    password: ''
});

const { isLoggedIn } = React.useContext(AuthContext);

const [error, setError] = React.useState<string | null>('');

React.useEffect(() => {
    // if user is already logged in, redirect to home page
    if (isLoggedIn) {
        navigate('/');
    }
}, []);

// update form data on input change
const handleLoginFormInputChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    const { name, value } = e.target;
    setFormData(prevState => ({
        ...prevState,
        [name]: value
    }));
};

// handle form submit
const handleLoginFormSubmit = (e: React.FormEvent<HTMLFormElement>) => {
    e.preventDefault();
    fetch(`${import.meta.env.VITE_API_URL}/users/create`, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(formData)
    })
    .then(res => res.json())
    .then(res => {
        navigate('/login', { replace: true, state: { user: res.data } });
    })
    .catch(err => {
        console.error(err);
        setError(err.message);
    });
}
```

Figure 31 illustrated the redirect method that navigate the user to the home page after sign up.

5.3.2 File Upload

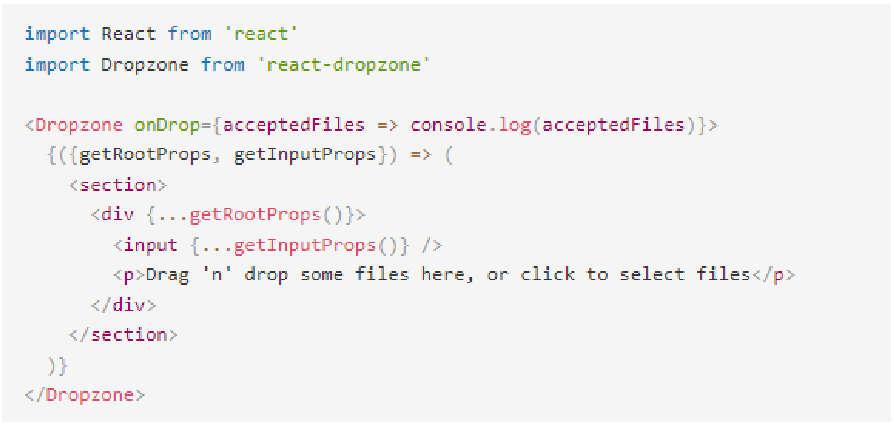
Front-end Implementation

The file upload functionality allowed the user to quickly drag and drop their audio file for automatic transcription. The feature utilised the React Dropzone to easily integrate into any application using their hook. The functionality could be initialised in a couple of ways; either initiated through the use of hook or a wrapper component shown in Figure 32. Albeit the similarity between the two were identical, the usage of hook, shown in Figure 33, was considerably cleaner in terms of its readability, where the wrapper clustered the content in one place as illustrated in Figure 32. Thus, the author decided to utilise the hook for easy readability and debugging.



```
import React, {useCallback} from 'react'
import {useDropzone} from 'react-dropzone'

function MyDropzone() {
  const onDrop = useCallback(acceptedFiles => {
    // Do something with the files
  }, [])
  const {getRootProps, getInputProps, isDragActive} = useDropzone({onDrop})

  return (
    <div {...getRootProps()}>
      <input {...getInputProps()} />
      {
        isDragActive ?
          <p>Drop the files here ...</p> :
          <p>Drag 'n' drop some files here, or click to select files</p>
      }
    </div>
  )
}
```

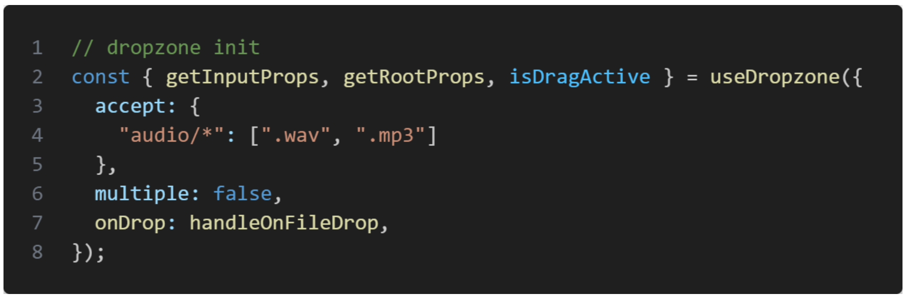
Figure 32 illustrated the usage of hook in Dropzone (n.d.).



```
import React from 'react'
import Dropzone from 'react-dropzone'

<Dropzone onDrop={acceptedFiles => console.log(acceptedFiles)}>
  {({getRootProps, getInputProps}) => (
    <section>
      <div {...getRootProps()}>
        <input {...getInputProps()} />
        <p>Drag 'n' drop some files here, or click to select files</p>
      </div>
    </section>
  )}
</Dropzone>
```

Figure 33 illustrated the usage of hook in a wrapper component. (n.d.)

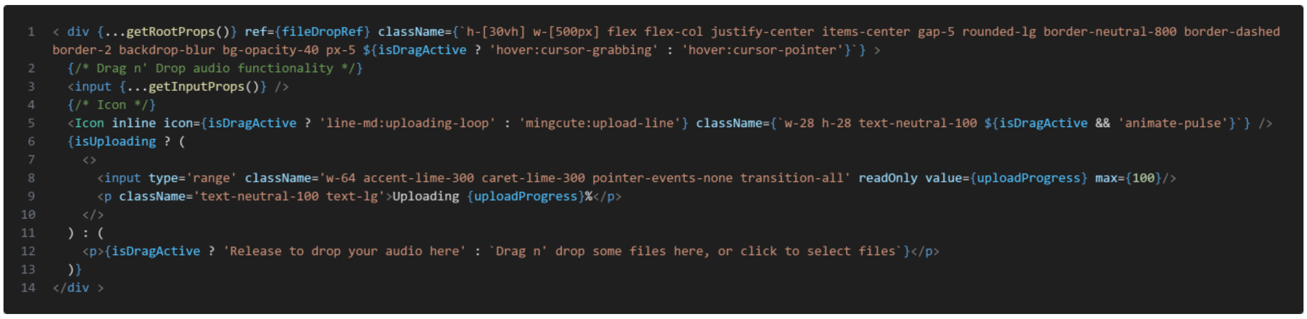
Firstly, the Dropzone hook, illustrated in Figure 34, would need to be initialised using useDropzone API, alongside the configuration object, specifying to accept only certain file extensions i.e. WAV and MP3 files, to disabled multiple files upload, and to provide a function to handle the file once it had been dropped into the input.



```
// dropzone init
const { getInputProps, getRootProps, isDragActive } = useDropzone({
  accept: {
    "audio/*": [".wav", ".mp3"]
  },
  multiple: false,
  onDrop: handleOnFileDrop,
});
```

Figure 34 illustrated the Dropzone hook that was used by the author in the application.

The Figure 35 illustrated how the hook was utilized in markups. The props getters were passed as functions to a parent wrapper, and a child input. This offered the parent and child elements the API to expose the Ref, as well as the ability to check for dragging activity to change CSS styling accordingly.



```
< div {...getRootProps()} ref={fileDropRef} className={`h-[30vh] w-[500px] flex flex-col justify-center items-center gap-5 rounded-lg border-neutral-800 border-dashed
border-2 backdrop-blur bg-opacity-40 px-5 ${isDragActive ? 'hover:cursor-grabbing' : 'hover:cursor-pointer'}`} >
  {/* Drag n' Drop audio functionality */}
  <input {...getInputProps()} />
  {/* Icon */}
  <Icon inline icon={isDragActive ? 'line-md:uploading-loop' : 'mingcute:upload-line'} className={`w-28 h-28 text-neutral-100 ${isDragActive && 'animate-pulse'}`} />
  {isUploading ? (
    <>
      <input type='range' className='w-64 accent-lime-300 caret-lime-300 pointer-events-none transition-all' readOnly value={uploadProgress} max={100}/>
      <p className='text-neutral-100 text-lg'>Uploading {uploadProgress}%</p>
    </>
  ) : (
    <p>{isDragActive ? 'Release to drop your audio here' : `Drag n' drop some files here, or click to select files`}</p>
  )}
</div >
```

Figure 35 illustrated the usage of Dropzone hook in the homepage.

Once the file has been dropped into the input, a function would be executed to handle the file upload as shown in Figure 36. This included attaching the audio file as a FormData, opening XML request to the server, sending the audio to be processed by the server, all the while showing the progress bar to prevent dull interactions. If the request was successful, the user would be navigated to the music sheet of the upload audio, otherwise an error response would be returned.

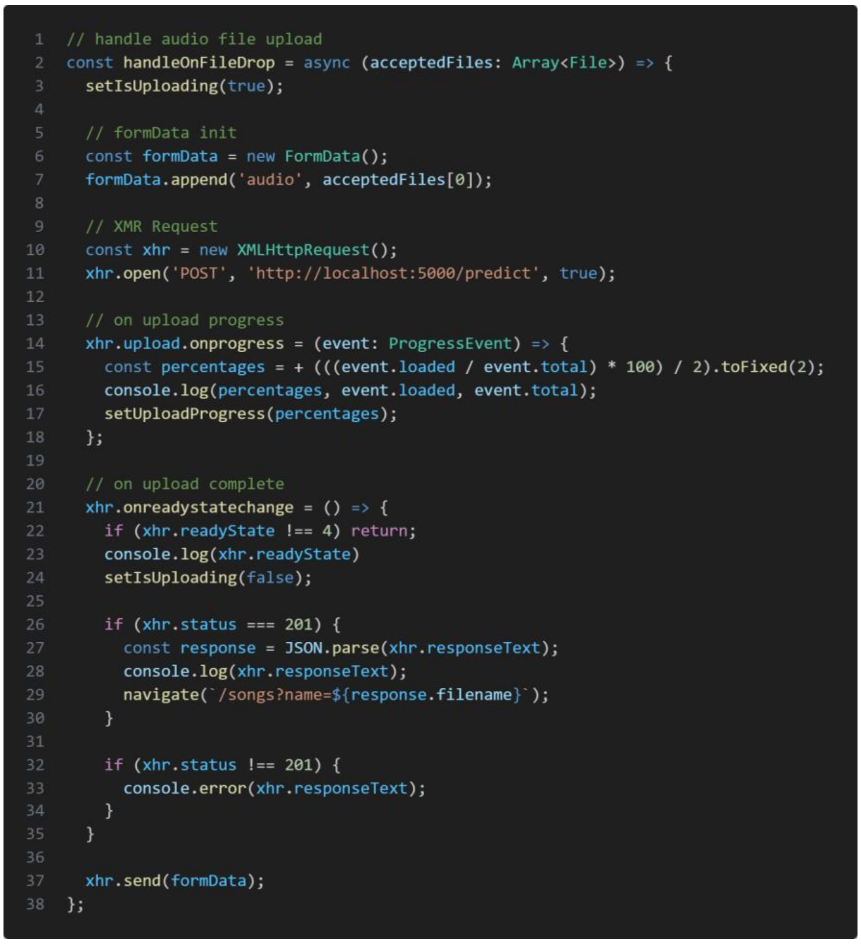
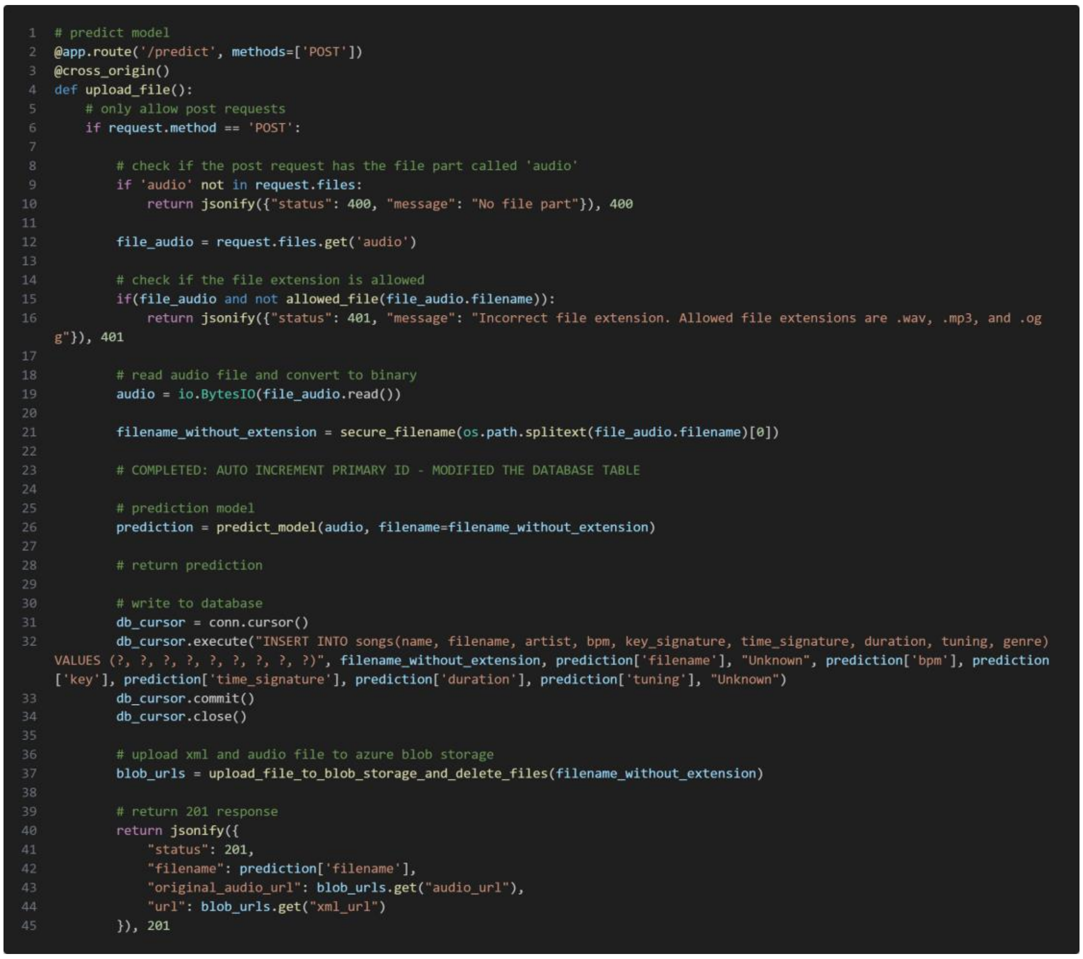

```
// handle audio file upload
const handleOnFileDrop = async (acceptedFiles: Array<File>) => {
  setIsUploading(true);

  // formData init
  const formData = new FormData();
  formData.append('audio', acceptedFiles[0]);

  // XMR Request
  const xhr = new XMLHttpRequest();
  xhr.open('POST', 'http://localhost:5000/predict', true);

  // on upload progress
  xhr.upload.onprogress = (event: ProgressEvent) => {
    const percentages = + (((event.loaded / event.total) * 100) / 2).toFixed(2);
    console.log(percentages, event.loaded, event.total);
    setUploadProgress(percentages);
  };

  // on upload complete
  xhr.onreadystatechange = () => {
    if (xhr.readyState !== 4) return;
    console.log(xhr.readyState)
    setIsUploading(false);

    if (xhr.status === 201) {
      const response = JSON.parse(xhr.responseText);
      console.log(xhr.responseText);
      navigate(`/songs?name=${response.filename}`);
    }

    if (xhr.status !== 201) {
      console.error(xhr.responseText);
    }
  }

  xhr.send(formData);
};
```

Figure 36 illustrated how the audio file was handled once it was dropped into the input.

Back-end Implementation

While the Front end handled the user's request, the Back end would need to handle the audio processing. The route called 'predict' was defined, as shown in Figure 37, to only take in a POST request and a multi-part file request named 'audio'. Otherwise, a 400 HTTP response would be returned to specify that no file existed on the request header. The validation to detect file extensions was also implemented in the Back end to gatekeep incorrect Mime types i.e. image file type such as PNG, JPEG, or GIF.

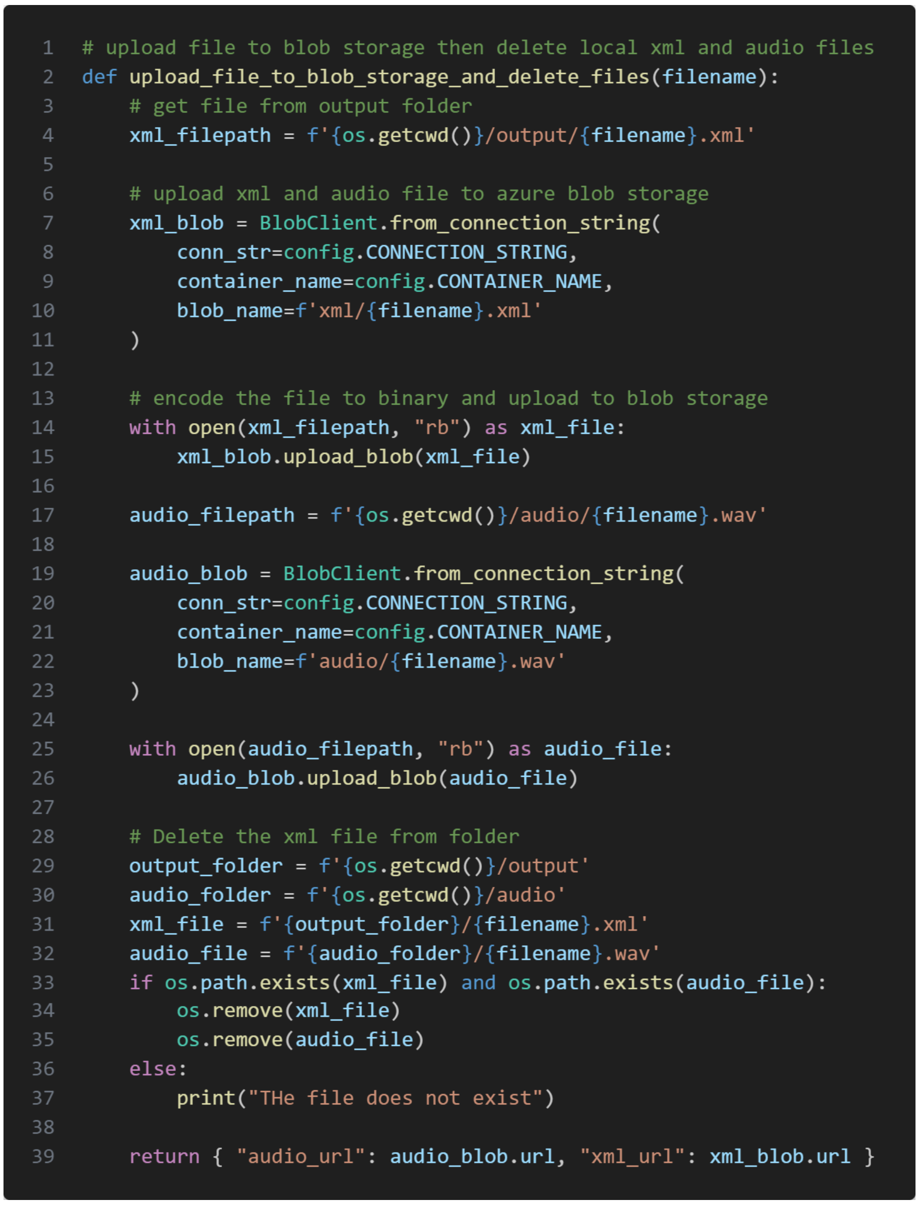


```python
# predict model
@app.route('/predict', methods=['POST'])
@cross_origin()
def upload_file():
    # only allow post requests
    if request.method == 'POST':

        # check if the post request has the file part called 'audio'
        if 'audio' not in request.files:
            return jsonify({"status": 400, "message": "No file part"}), 400

        file_audio = request.files.get('audio')

        # check if the file extension is allowed
        if(file_audio and not allowed_file(file_audio.filename)):
            return jsonify({"status": 401, "message": "Incorrect file extension. Allowed file extensions are .wav, .mp3, and .ogg"}), 401

        # read audio file and convert to binary
        audio = io.BytesIO(file_audio.read())

        filename_without_extension = secure_filename(os.path.splitext(file_audio.filename)[0])

        # COMPLETED: AUTO INCREMENT PRIMARY ID - MODIFIED THE DATABASE TABLE

        # prediction model
        prediction = predict_model(audio, filename=filename_without_extension)

        # return prediction

        # write to database
        db_cursor = conn.cursor()
        db_cursor.execute("INSERT INTO songs(name, filename, artist, bpm, key_signature, time_signature, duration, tuning, genre) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)", filename_without_extension, prediction['filename'], "Unknown", prediction['bpm'], prediction['key'], prediction['time_signature'], prediction['duration'], prediction['tuning'], "Unknown")
        db_cursor.commit()
        db_cursor.close()

        # upload xml and audio file to azure blob storage
        blob_urls = upload_file_to_blob_storage_and_delete_files(filename_without_extension)

        # return 201 response
        return jsonify({
            "status": 201,
            "filename": prediction['filename'],
            "original_audio_url": blob_urls.get("audio_url"),
            "url": blob_urls.get("xml_url")
        }), 201
```

Figure 37 illustrated the predict route that process the audio file and output XML file.

Once all validations were met, the audio file needed to be read to memory i.e. converted from binary data into streams of bytes. This was done using the IO module in Python to provide a file-like object interface that Librosa could understand and format. These streams would be fed into the predict_model function, then saved the MusicXML and audio file would be uploaded to Azure storage service called ‘Blob Storage’ and insert new column into the No-SQL database accordingly.

Finally, a 201 HTTP response would be returned alongside the name of the file, the audio file, and the URL of MusicXML that the Front end could utilized to fetch to retrieve the files necessary to display music sheet as illustrated by the Figure 38.



```python
# upload file to blob storage then delete local xml and audio files
def upload_file_to_blob_storage_and_delete_files(filename):
    # get file from output folder
    xml_filepath = f'{os.getcwd()}/output/{filename}.xml'

    # upload xml and audio file to azure blob storage
    xml_blob = BlobClient.from_connection_string(
        conn_str=config.CONNECTION_STRING,
        container_name=config.CONTAINER_NAME,
        blob_name=f'xml/{filename}.xml'
    )

    # encode the file to binary and upload to blob storage
    with open(xml_filepath, "rb") as xml_file:
        xml_blob.upload_blob(xml_file)

    audio_filepath = f'{os.getcwd()}/audio/{filename}.wav'

    audio_blob = BlobClient.from_connection_string(
        conn_str=config.CONNECTION_STRING,
        container_name=config.CONTAINER_NAME,
        blob_name=f'audio/{filename}.wav'
    )

    with open(audio_filepath, "rb") as audio_file:
        audio_blob.upload_blob(audio_file)

    # Delete the xml file from folder
    output_folder = f'{os.getcwd()}/output'
    audio_folder = f'{os.getcwd()}/audio'
    xml_file = f'{output_folder}/{filename}.xml'
    audio_file = f'{audio_folder}/{filename}.wav'
    if os.path.exists(xml_file) and os.path.exists(audio_file):
        os.remove(xml_file)
        os.remove(audio_file)
    else:
        print("THe file does not exist")

    return { "audio_url": audio_blob.url, "xml_url": xml_blob.url }
```

Figure 38 illustrated the function that handle file upload and store files on Azure Blob Storage.

5.3.3 Song Lists

While the homepage handled the file upload, users would be able to also access the list of songs available in the database. This was done to give the users an easy access to the whole library of songs, where they could potentially learn new genres.

The useEffect hook was utilised to fetch the list of songs using a custom hook called ‘useFetch’. This custom hook returned a promise of the response from a URL. The hook utilised TypeScript to pass in a generic response type, as shown in Figure 39 and 40, for a better developer experience and make it easy to debug in the long run.

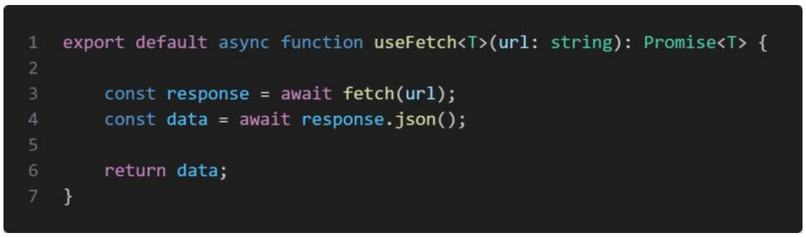

```
export default async function useFetch<T>(url: string): Promise<T> {

    const response = await fetch(url);
    const data = await response.json();

    return data;
}
```

Figure 39 illustrated the custom hook used to fetch the data from URL.

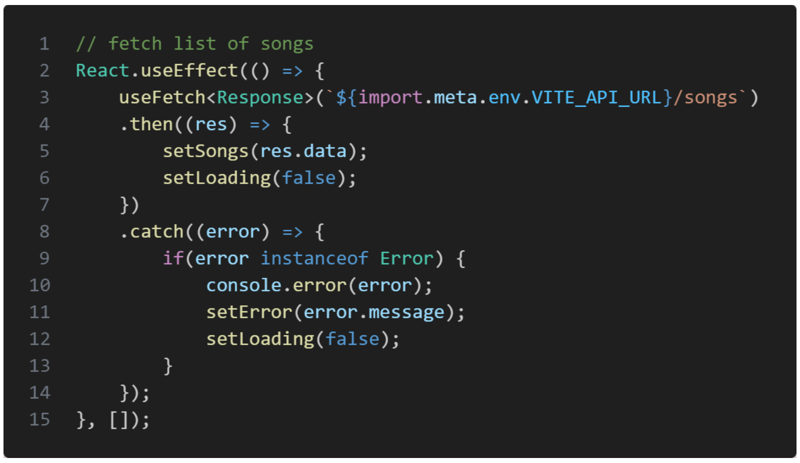

```
// fetch list of songs
React.useEffect(() => {
    useFetch<Response>(`${import.meta.env.VITE_API_URL}/songs`)
    .then((res) => {
        setSongs(res.data);
        setLoading(false);
    })
    .catch((error) => {
        if(error instanceof Error) {
            console.error(error);
            setError(error.message);
            setLoading(false);
        }
    });
}, []);
```

Figure 40 illustrated how the list of songs was fetched.

Once the response had been returned from the API endpoint, it could either returned a successful 200 HTTP response, or an error message if there was a possible malfunctioning on the server. In the scenario that the endpoint was successful, the returned data would be stored as an array in the useState called ‘Songs’, and the array would later be mapped into a list of components for displays as shown in Figure 41.

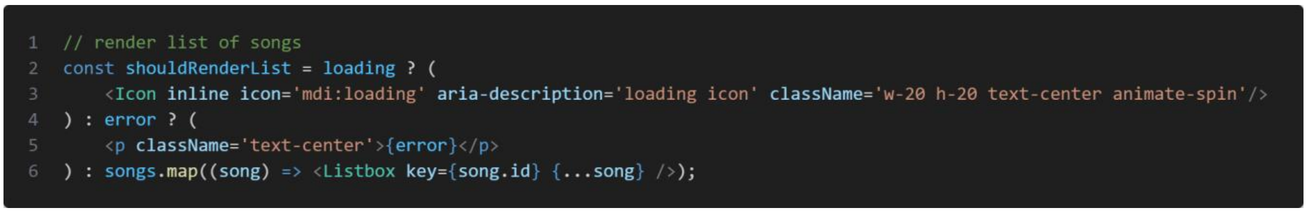

```
// render list of songs
const shouldRenderList = loading ? (
    <Icon inline icon='mdi:loading' aria-description='loading icon' className='w-20 h-20 text-center animate-spin'/>
) : error ? (
    <p className='text-center'>{error}</p>
) : songs.map((song) => <Listbox key={song.id} {...song} />);
```

Figure 41 illustrated how the list of songs was mapped into ListBox component.

5.3.4 Favourites

The users needed to able to favourite the song they want to re-visit later. To do this, the feature needed to be implemented on both the client and server, where the client would handle the visualisation of the list of favourites, while the server handled the query to Create, Read, Update, and Delete the favourite from the user, and song.

Favourite Page

Front-end Implementation

The feature had been quickly implemented as a Hi-Fi prototype, illustrated in Figure 42, to allow the author to visually characterise requirements needed for the page to function.

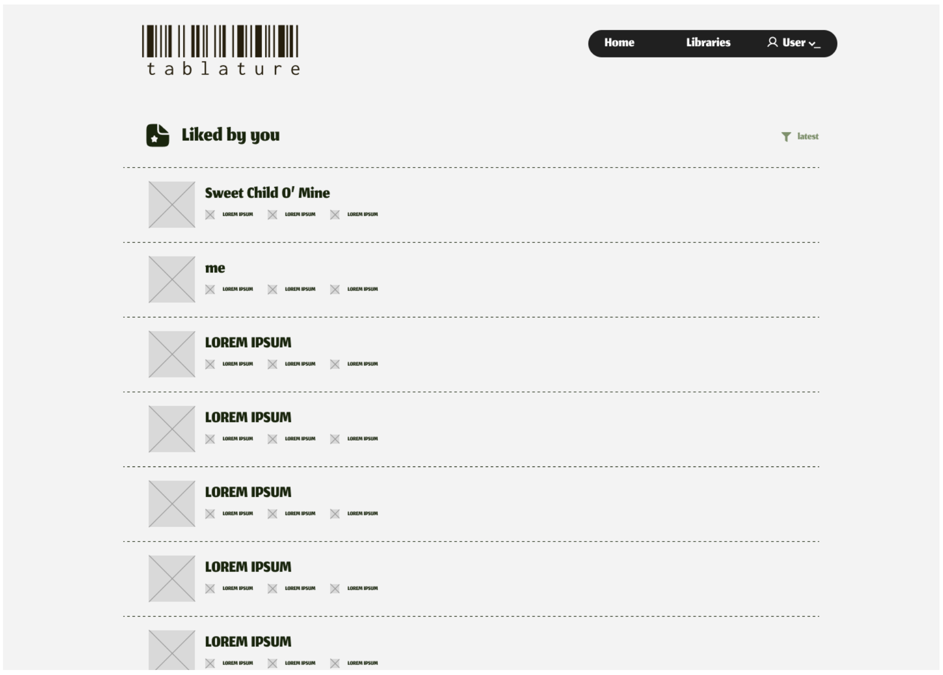
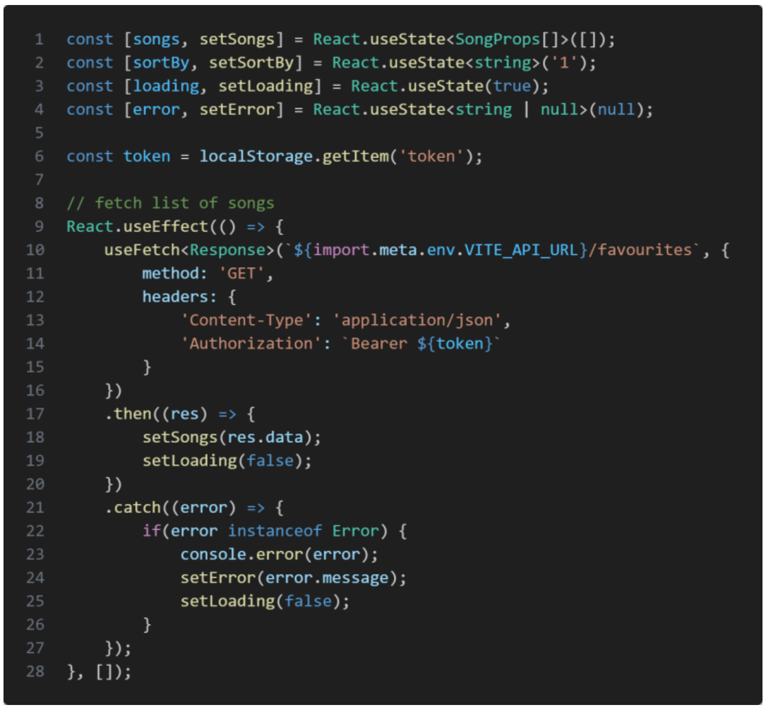


Figure 42 illustrated the Hi-Fi prototype of the Favourite page.

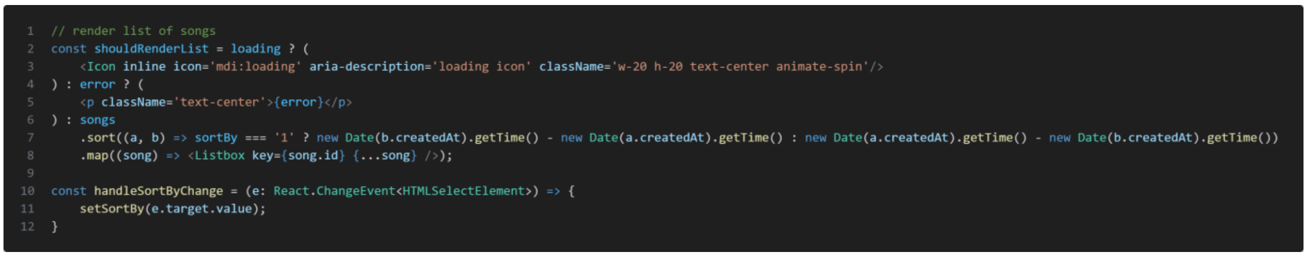
The component would fetch to the server once the page had been mounted via useEffect, where their JWT token would be passed along the headers to provide the server with the user information to retrieve their favourited songs as illustrated in Figure 43.



```
const [songs, setSongs] = React.useState<SongProps[]>([]);
const [sortBy, setSortBy] = React.useState<string>('1');
const [loading, setLoading] = React.useState(true);
const [error, setError] = React.useState<string | null>(null);

const token = localStorage.getItem('token');

// fetch list of songs
React.useEffect(() => {
    useFetch<Response>(`${import.meta.env.VITE_API_URL}/favourites`, {
        method: 'GET',
        headers: {
            'Content-Type': 'application/json',
            'Authorization': `Bearer ${token}`
        }
    })
    .then((res) => {
        setSongs(res.data);
        setLoading(false);
    })
    .catch((error) => {
        if(error instanceof Error) {
            console.error(error);
            setError(error.message);
            setLoading(false);
        }
    });
}, []);
```

Figure 43 illustrated the useFetch that request to the API for the list of songs.

These favourites would then be stored using a useState hook and the data would be mapped into a list of components for each favourite as shown in Figure 44.

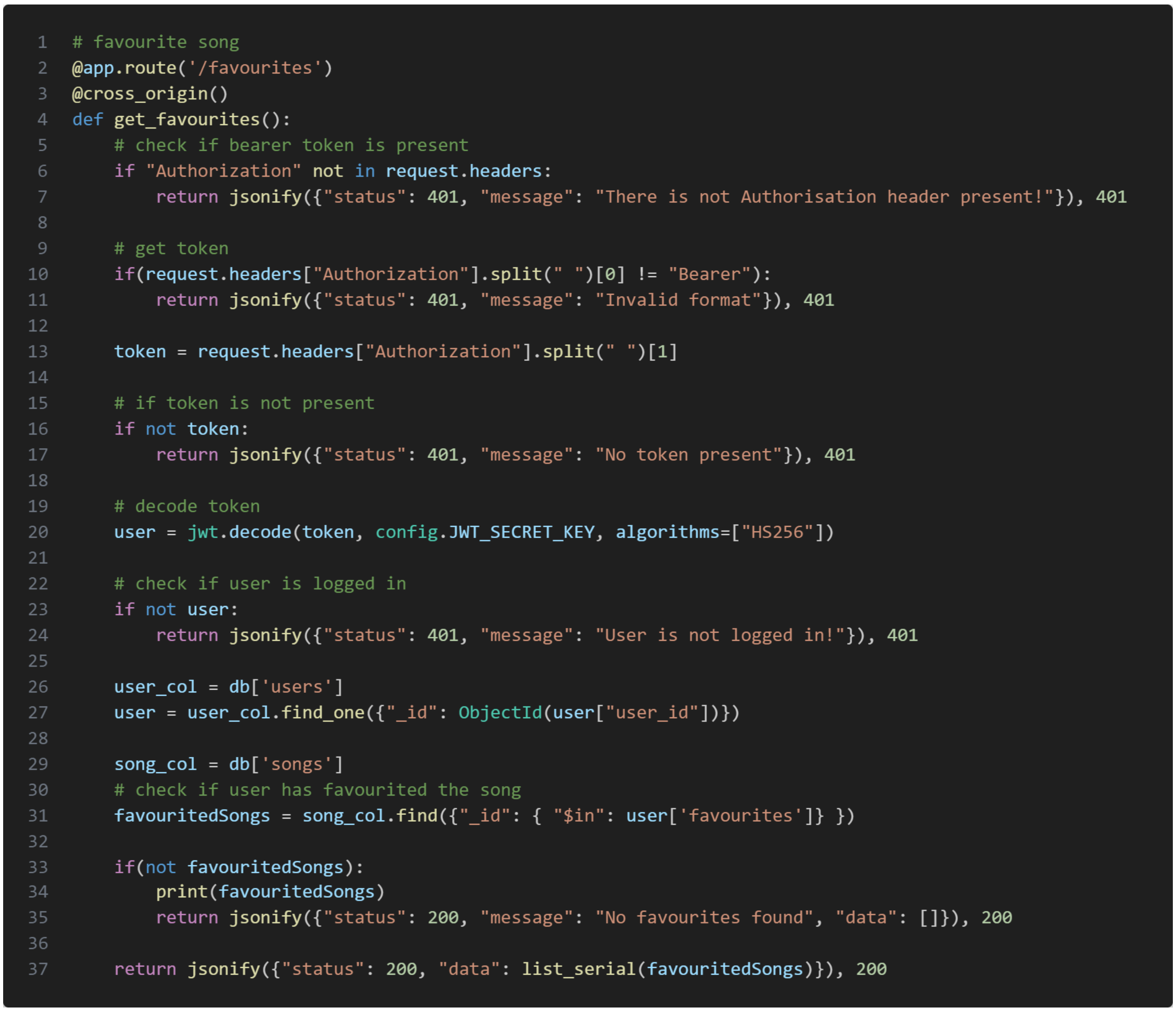


```
// render list of songs
const shouldRenderList = loading ? (
    <Icon inline icon='mdi:loading' aria-description='loading icon' className='w-20 h-20 text-center animate-spin'/>
) : error ? (
    <p className='text-center'>{error}</p>
) : songs
    .sort((a, b) => sortBy === '1' ? new Date(b.createdAt).getTime() - new Date(a.createdAt).getTime() : new Date(a.createdAt).getTime() - new Date(b.createdAt).getTime())
    .map((song) => <Listbox key={song.id} {...song} />);

const handleSortByChange = (e: React.ChangeEvent<HTMLSelectElement>) => {
    setSortBy(e.target.value);
}
```

Figure 44 illustrated the list of songs that would be mapped into components.

Back-end Implementation

The Figure 45 shown handled the retrieval of the song(s) favourited by the user. The API endpoint would need to acquire the token from the user, which would be passed from the client. This token would then be decoded through the JWT library to retrieve the ID of the user. Once the user ID had been retrieved the server would execute a search query to the database with the user ID and the favourited songs were able to be sent back to the client through the form of a JSON response.



```python
# favourite song
@app.route('/favourites')
@cross_origin()
def get_favourites():
    # check if bearer token is present
    if "Authorization" not in request.headers:
        return jsonify({"status": 401, "message": "There is not Authorisation header present!"}), 401

    # get token
    if(request.headers["Authorization"].split(" ")[0] != "Bearer"):
        return jsonify({"status": 401, "message": "Invalid format"}), 401

    token = request.headers["Authorization"].split(" ")[1]

    # if token is not present
    if not token:
        return jsonify({"status": 401, "message": "No token present"}), 401

    # decode token
    user = jwt.decode(token, config.JWT_SECRET_KEY, algorithms=["HS256"])

    # check if user is logged in
    if not user:
        return jsonify({"status": 401, "message": "User is not logged in!"}), 401

    user_col = db['users']
    user = user_col.find_one({"_id": ObjectId(user["user_id"])})

    song_col = db['songs']
    # check if user has favourited the song
    favouritedSongs = song_col.find({"_id": { "$in": user['favourites']} })

    if(not favouritedSongs):
        print(favouritedSongs)
        return jsonify({"status": 200, "message": "No favourites found", "data": []}), 200

    return jsonify({"status": 200, "data": list_serial(favouritedSongs)}), 200
```

Figure 45 illustrated the server side that handled the API request for the list of songs.

Tab Visualiser Page

Front-end Implementation

The Figure 46 below illustrated the read, create, and edit functionality of the favourite, where the heart icon was responsible for these API operations.

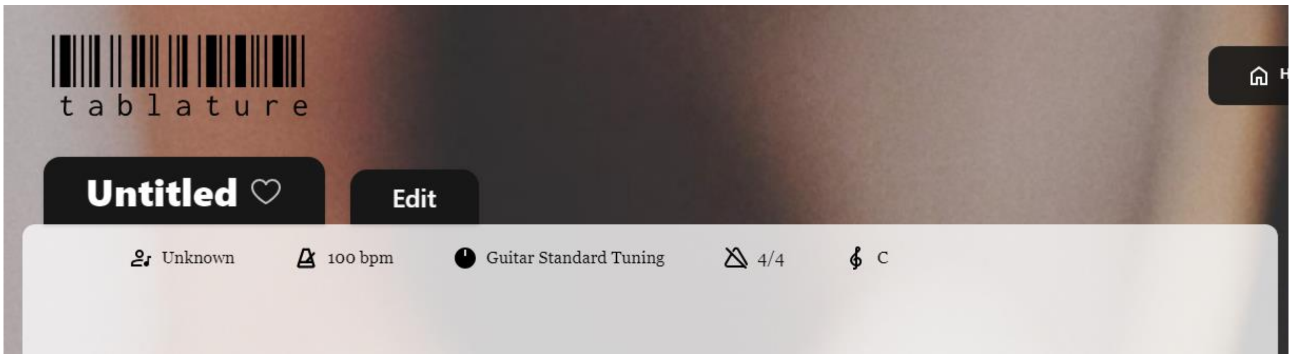
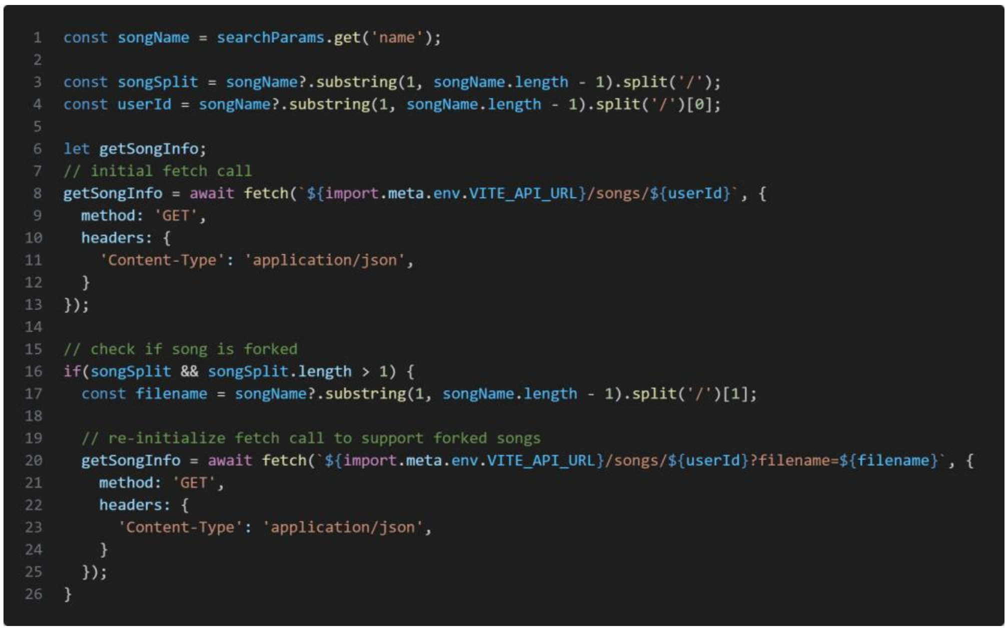


Figure 46 illustrated the Tab Visualiser page that perform the CRUD operation of the favourite functionality.

Once the component was mounted, the useEffect hook was utilised to execute a bunch of API calls to the server, which included:

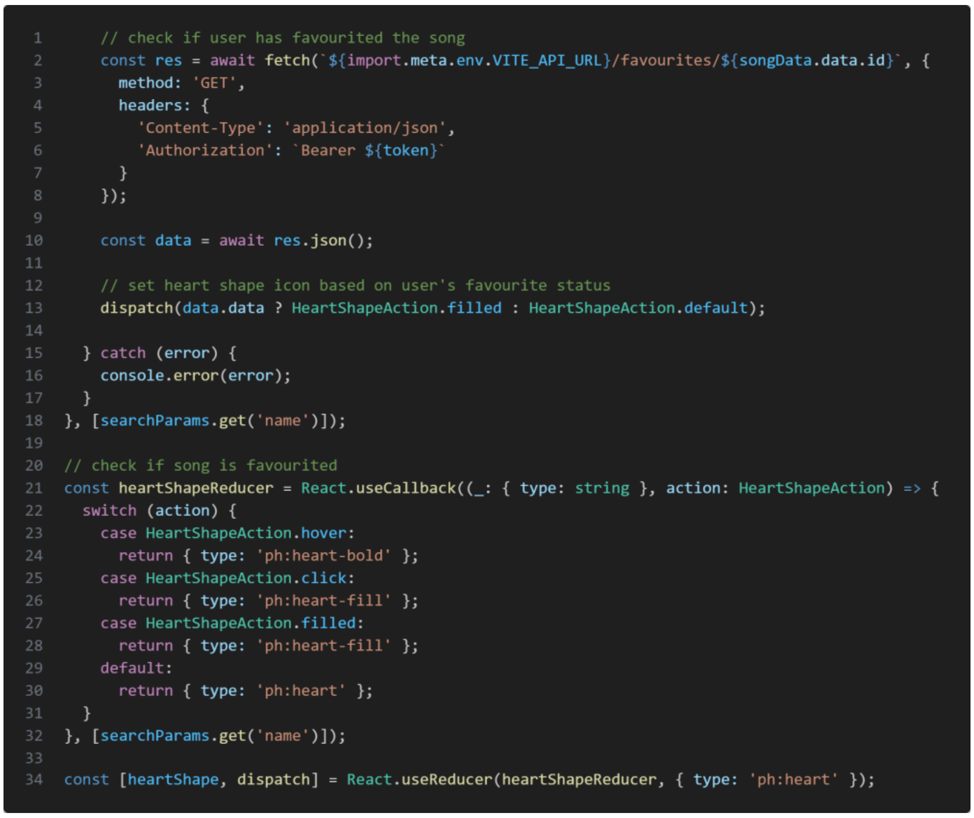
- Check if the user had already favourited the song – First, the component needed to fetch for the details of the song, where the favourite array was stored on MongoDB instance. Depending on the name of the query in the URL path, the API would be re-initialised to suit if the tab was forked from an existing song as shown in Figure 47 below.



```
const songName = searchParams.get('name');

const songSplit = songName?.substring(1, songName.length - 1).split('/');
const userId = songName?.substring(1, songName.length - 1).split('/')[0];

let getSongInfo;
// initial fetch call
getSongInfo = await fetch(`${import.meta.env.VITE_API_URL}/songs/${userId}`, {
  method: 'GET',
  headers: {
    'Content-Type': 'application/json',
  }
});

// check if song is forked
if(songSplit && songSplit.length > 1) {
  const filename = songName?.substring(1, songName.length - 1).split('/')[1];

  // re-initialize fetch call to support forked songs
  getSongInfo = await fetch(`${import.meta.env.VITE_API_URL}/songs/${userId}?filename=${filename}`, {
    method: 'GET',
    headers: {
      'Content-Type': 'application/json',
    }
  });
}
```

Figure 47 illustrated the functionality that check if the user had favourited the song.

The ID of the song would then be passed into another API call to retrieve the information of the favourite, which returned a Boolean depending on if the user had favourited the song. This Boolean was dispatched to a useReducer hook to change the heart icon to a hollowed heart or a filled heart as illustrated by the Figure 48 below.

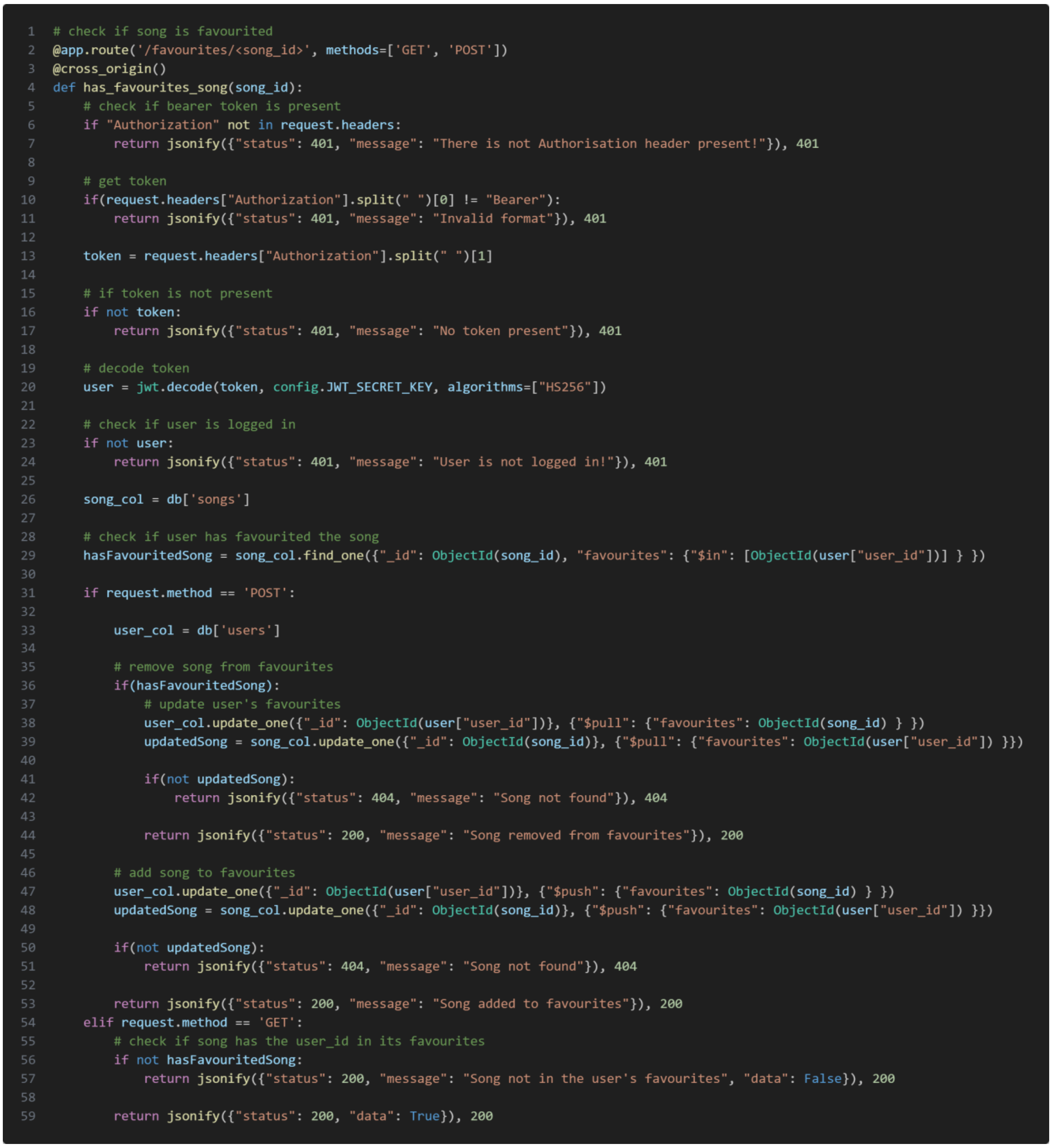


```
    // check if user has favourited the song
    const res = await fetch(`${import.meta.env.VITE_API_URL}/favourites/${songData.data.id}`, {
      method: 'GET',
      headers: {
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${token}`
      }
    });

    const data = await res.json();

    // set heart shape icon based on user's favourite status
    dispatch(data.data ? HeartShapeAction.filled : HeartShapeAction.default);

  } catch (error) {
    console.error(error);
  }
}, [searchParams.get('name')]);

// check if song is favourited
const heartShapeReducer = React.useCallback((_: { type: string }, action: HeartShapeAction) => {
  switch (action) {
    case HeartShapeAction.hover:
      return { type: 'ph:heart-bold' };
    case HeartShapeAction.click:
      return { type: 'ph:heart-fill' };
    case HeartShapeAction.filled:
      return { type: 'ph:heart-fill' };
    default:
      return { type: 'ph:heart' };
  }
}, [searchParams.get('name')]);

const [heartShape, dispatch] = React.useReducer(heartShapeReducer, { type: 'ph:heart' });
```

Figure 48 illustrated the useReducer that handle the Boolean of the favourite state.

Back-end Implementation

The server that handled these requests was specified as a function that took in both GET and POST method. The ID of the song would need to be passed in conjunction with the JWT token for the request to be accepted as valid. Once the token had been decoded to retrieve the user ID, a query would be sent to the database with both IDs as the filter. If the request was a GET request, the server would return a Boolean as the response to the client. While the POST request would perform another query to update the song and user documents, which either insert or remove the favourite shown in Figure 49.



```python
# check if song is favourited
@app.route('/favourites/<song_id>', methods=['GET', 'POST'])
@cross_origin()
def has_favourites_song(song_id):
    # check if bearer token is present
    if "Authorization" not in request.headers:
        return jsonify({"status": 401, "message": "There is not Authorisation header present!"}), 401

    # get token
    if(request.headers["Authorization"].split(" ")[0] != "Bearer"):
        return jsonify({"status": 401, "message": "Invalid format"}), 401

    token = request.headers["Authorization"].split(" ")[1]

    # if token is not present
    if not token:
        return jsonify({"status": 401, "message": "No token present"}), 401

    # decode token
    user = jwt.decode(token, config.JWT_SECRET_KEY, algorithms=["HS256"])

    # check if user is logged in
    if not user:
        return jsonify({"status": 401, "message": "User is not logged in!"}), 401

    song_col = db['songs']

    # check if user has favourited the song
    hasFavouritedSong = song_col.find_one({"_id": ObjectId(song_id), "favourites": {"$in": [ObjectId(user["user_id"])] } })

    if request.method == 'POST':

        user_col = db['users']

        # remove song from favourites
        if(hasFavouritedSong):
            # update user's favourites
            user_col.update_one({"_id": ObjectId(user["user_id"])}, {"$pull": {"favourites": ObjectId(song_id) } })
            updatedSong = song_col.update_one({"_id": ObjectId(song_id)}, {"$pull": {"favourites": ObjectId(user["user_id"]) }})

            if(not updatedSong):
                return jsonify({"status": 404, "message": "Song not found"}), 404

            return jsonify({"status": 200, "message": "Song removed from favourites"}), 200

        # add song to favourites
        user_col.update_one({"_id": ObjectId(user["user_id"])}, {"$push": {"favourites": ObjectId(song_id) } })
        updatedSong = song_col.update_one({"_id": ObjectId(song_id)}, {"$push": {"favourites": ObjectId(user["user_id"]) }})

        if(not updatedSong):
            return jsonify({"status": 404, "message": "Song not found"}), 404

        return jsonify({"status": 200, "message": "Song added to favourites"}), 200
    elif request.method == 'GET':
        # check if song has the user_id in its favourites
        if not hasFavouritedSong:
            return jsonify({"status": 200, "message": "Song not in the user's favourites", "data": False}), 200

        return jsonify({"status": 200, "data": True}), 200
```

Figure 49 illustrated the server that handled the CRUD functionality of favourite.

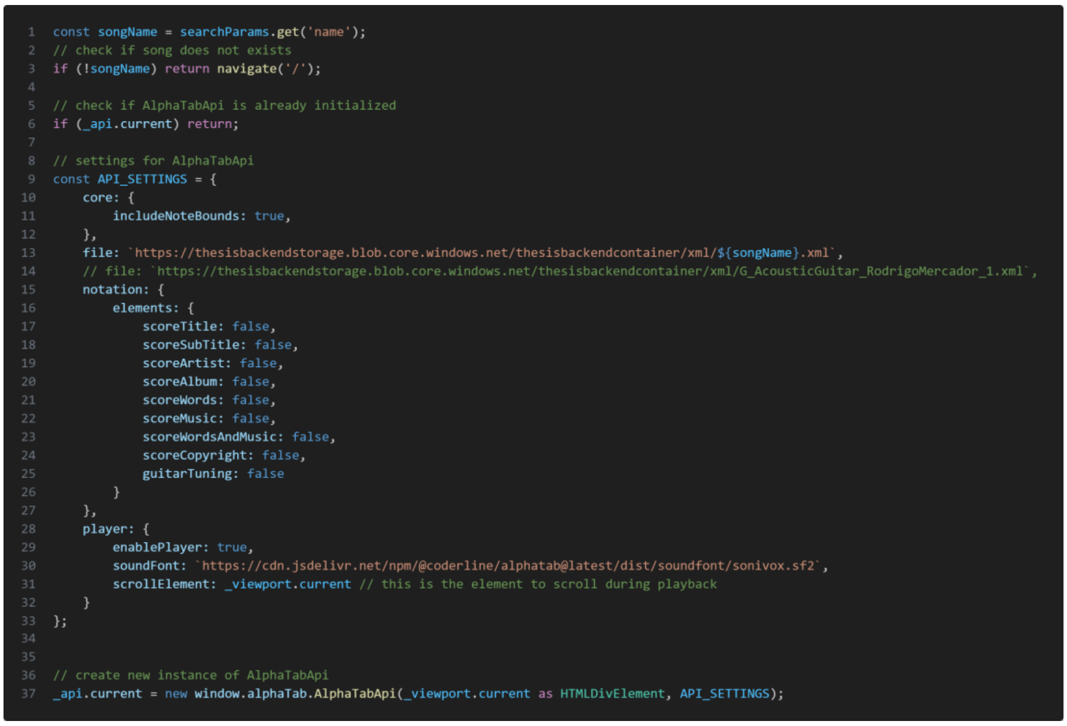
5.3.5 Tab Visualisation

Visualiser

Once the audio had been successfully predicted, the user would be navigated to the interactive music sheet, where they would have the ability to point and click to any section of the song. By utilising a JavaScript library called ‘AlphaTab’, the author was able to implement such feature with the API that provided ways to track the current beat, note, and bar in a music sheet.

In order to set up AlphaTab for utilisation, there was some complication that was worth mentioning, where the library would struggle to be integrate into a React application. After extensive research, the author had discovered that Create-React-App and Vite bundlers would not work with AlphaTab’s worker, both of which the author had tried to use with no avails. According to Fheyen (2023) and Matevsenlab (2023), whom discussed about these very issues with the developer of AlphaTab and found that either bundlers were supported by the library at the moment due to ‘rather terrible support of Web Workers and Audio Worklets’, of which the library relied heavily on. The developer of the library suggested that one should customise the Webpack configuration to change how Webpack bundled dependencies. Albeit the attempt, the author was not managed to solve the issue. Thus, the final resource was to utilise AlphaTab’s Content Delivery Network (CDN) to import the dependency or install the library as static files in the public path of the application. Though using this method, the developer experience could be less enjoyable since there won’t be autocompletes, but there was no other choice until the developer decided to increase compatibility to web bundlers.

To initalise the library, a setting must be first provided into the AlphaTab API to enable a couple of features such as IncludeNoteBounds for note editability, hide elements in the music sheet, load in MusicXML file, and MIDI player as shown in Figure 50.



```
const songName = searchParams.get('name');
// check if song does not exists
if (!songName) return navigate('/');

// check if AlphaTabApi is already initialized
if (_api.current) return;

// settings for AlphaTabApi
const API_SETTINGS = {
    core: {
        includeNoteBounds: true,
    },
    file: `https://thesisbackendstorage.blob.core.windows.net/thesisbackendcontainer/xml/${songName}.xml`,
    // file: `https://thesisbackendstorage.blob.core.windows.net/thesisbackendcontainer/xml/G_AcousticGuitar_RodrigoMercador_1.xml`,
    notation: {
        elements: {
            scoreTitle: false,
            scoreSubTitle: false,
            scoreArtist: false,
            scoreAlbum: false,
            scoreWords: false,
            scoreMusic: false,
            scoreWordsAndMusic: false,
            scoreCopyright: false,
            guitarTuning: false
        }
    },
    player: {
        enablePlayer: true,
        soundFont: `https://cdn.jsdelivr.net/npm/@coderline/alphatab@latest/dist/soundfont/sonivox.sf2`,
        scrollElement: _viewport.current // this is the element to scroll during playback
    }
};


// create new instance of AlphaTabApi
_api.current = new window.alphaTab.AlphaTabApi(_viewport.current as HTMLDivElement, API_SETTINGS);
```

Figure 50 illustrated the initialisation of AlphaTab.

Since the AlphaTab was utilised in React, the best practice to initialise the API would be in the useEffect to prevent the API from re-rendering once the component was updated or mounted on the page. The API would also take in event listeners to execute codes based on the conditions such as:

1. Toggle loading screen – the feature utilized the renderStarted and renderFinished event listeners to toggle the display property in CSS of the element to show/hide the loading screen.
2. Set song details – this utilized the scoreLoaded event listener to store the state of the loaded music sheet to display the song title in the header, as well as other information such as tempo, tuning, time signature, key signature, and artist name in the sub header.
3. Set the volume of the MIDI player – if the user had set the volume to another value prior, the cached volume would be fetched from the local storage to enhance the UX.

Once the music score had been loaded the interactive sheet would be displayed to the user, where they would be able to seek to the section would like to listen to and either choose to edit the sheet if they are logged in.

Audio Player

The audio player, shown in Figure 51, was implemented to greatly enhance the UX and UI when a user browses through the music sheet, making the sheet interactive to seek to any section of the score, play/pause the audio, and adjust the volume of the output audio.

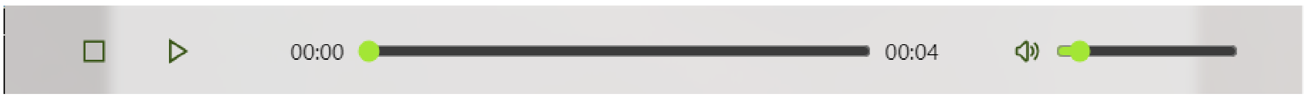


Figure 51 illustrated the interface of the audio player.

The player, itself, was a component that took in the API to use its play/pause, stop, seek, volume, current duration, and total duration to function. The API functions were easily integrated into the component and passed down to the child components to keep the parent component clean and manageable. Finally, the player was created as a portal to appended after the React DOM to styled it to fix on the bottom of the webpage.

Editing

The ability to edit the tablature had been developed as a conceptual implementation. This meant that the editability was met in terms of its functional requirement to select, edit, and add new note into the tablature. Though, the non-functional features that were not be able in the course of the project were the ability to add different beat types, adding new bar to the music sheet, and the ability to delete the note was not added.

Fork Edits

To be able to edit the existing tab, the edit button that had been added to the current tab would navigate to the edit page. Once the new page was mounted, it would fetch to the server using the token of the currently logged-in user, in order to assert the user id as the owner of the new edited tab and inserted it alongside the filename for the client to locate the MusicXML and audio files on Azure Blob Storage.

Update Edits

Once the user was logged in and able to use the edit page, the save button would be put in place of the edit button to send an API request to the server, where it would attempt to overwrite the existing MusicXML file saved on Blob Storage.

Delete Edits

In the case that the song needed to be removed, the delete button could be utilised which also send an API request to the server, where it would be checked against the token to validate the user’s ownership to the edit. If the validation was met, the MusicXML file would be removed from the Blob Storage along with the document in MongoDB. Otherwise, the message would be returned to the client stating that the user was not an authorised user and cannot perform the action.

5.3.6 Chord and Note Predictions

The automatic guitar transcription was designed to facilitate the Back end and utilised the audio sent in from the Front end to predict possible chords or notes in the consecutive time frame in terms of milliseconds.

Guitar Spectrograms

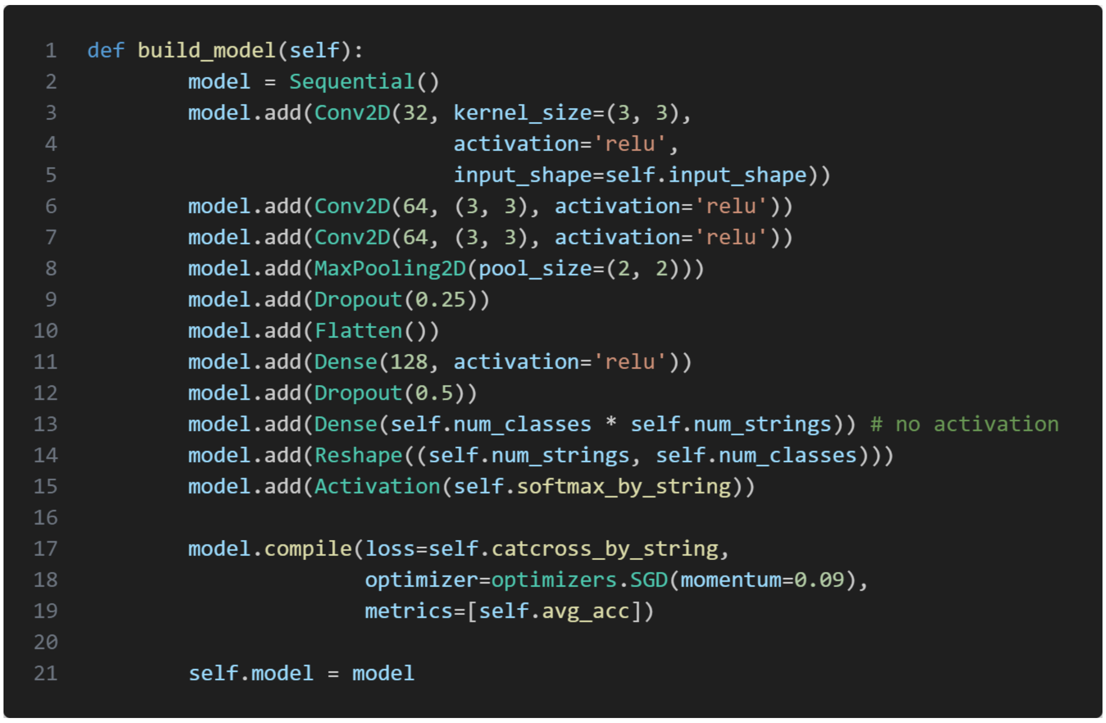
In order for the model to predict the result, the audio needed to be converted into information that could be interpreted by the prediction model. The author had experimented with a few methods to further choose the best implementation to utilised. Even though, the go-to choice was to convert the audio waveform into a constant-q spectrogram, there was a different method that this data of the spectrograms were used in MIR problems, such as:

- Utilising the Constant-Q data; through the use of Librosa to convert audio waveform into a time-frequency series, the output could be purely fed into the input of the model to learn.
- Utilising the image version of spectrograms to feed into the input where each image would be converted to grayscale, resized, and normalised into a float value between 0 and 1. This could gradually contribute to a better performance of the model, as it fixated some certainty in the range of weights.

But before either data was fed into the model for training, the loudness of each frequency in the data would need to be lower to prevent noises interfering with the predictions. This was done through putting threshold on frequencies that were below a certain decibel (-61 dB in this case), where any possible ringing note in a string could be neutralised and hindered it from bleeding into another onset.

Model

The model, shown in Figure 52, consisted of a layer which takes in an input (i.e. CQT matrices) and pass down through each Convolutional layer to extract possible features with an additional Max-pooling layer and Dropout layer to prevent over-fitting the model and hindered it from too tightly coupling with the training data, and may result in lower accuracy. Adjacently, the fully connected layer (Dense layer) was put in place to convert the extracted features into a data type that could be understood by the Activation layer to determine the prediction of the waveform using the SoftMax function due to its suitability with classifying group tasks.



```python
def build_model(self):
        model = Sequential()
        model.add(Conv2D(32, kernel_size=(3, 3),
                             activation='relu',
                             input_shape=self.input_shape))
        model.add(Conv2D(64, (3, 3), activation='relu'))
        model.add(Conv2D(64, (3, 3), activation='relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))
        model.add(Flatten())
        model.add(Dense(128, activation='relu'))
        model.add(Dropout(0.5))
        model.add(Dense(self.num_classes * self.num_strings)) # no activation
        model.add(Reshape((self.num_strings, self.num_classes)))
        model.add(Activation(self.softmax_by_string))

        model.compile(loss=self.catcross_by_string,
                      optimizer=optimizers.SGD(momentum=0.09),
                      metrics=[self.avg_acc])

        self.model = model
```

Figure 52 shows the CNN architecture behind TabCNN.

From investigating a similar application researched by Jadhav et. al (2022), where better prediction result was achieved through an optimiser called ‘SGD with momentum’, and the momentum rate was configured to 0.09. The change reflected in model to output overall accuracy of 89.8 percents from the original 80.3 percents.

Source Separation

The option to separate the guitar track from the mixture was also implemented to allow the user to transcribe guitar tab from audio that contained not only guitars but drums, basses, or vocals. This was accomplished via the use of a source separation model called Demucs-4HT, where this latest version could achieve the separation of guitar track from all its mixture, rather than its previous predecessors and other existing models that had lower accuracy or only separate guitar into the ‘other’ track with similar harmonic facet (or utilizing similar frequency range).

Front-end Implementation

The separation option would only be visible once the audio had been dropped into the input field as seen in Figure 53 below.

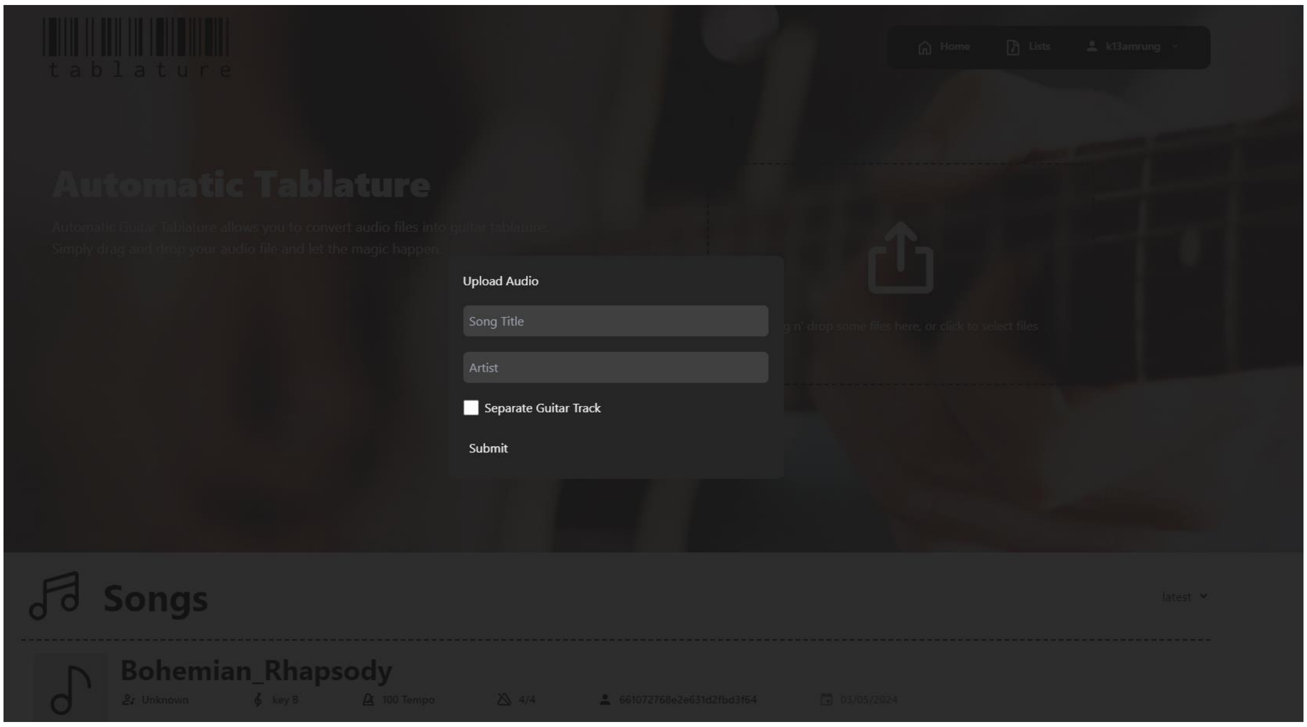

tablature
Home
Lists
k13amrung
Automatic Tablature
Automatic Guitar Tablature allows you to convert audio files into guitar tablature.
Simply drag and drop your audio file and let the magic happen.
Upload Audio
Song Title
Artist
Separate Guitar Track
Submit
Drag n' drop some files here, or click to select files
Songs
latest
Bohemian_Rhapsody
Unknown
key B
100 Tempo
4/4

Figure 53 illustrated the dialog that popped up once the audio input was triggered.

The modal that had appeared would pop up and act as an additional form to insert the title and the name of the artist into the database. The user could tick the checkbox which would add a query parameter called 'separate' to the API endpoint to let the server know it needed to separate the guitar track before predicting the audio.

Back-end Implementation

Once the audio had been requested to the server, it would validate whether the separate query was present. If the query was set to true, the Demucs-4 model would be utilised and the audio would first be passed into the model for post-processing that could take additional time before the separated guitar track would be saved into the local directory and re-assigned to the audio variable to feed into the prediction model.

5.4 Development Environment

5.4.1 Visual Studio Code

Visual Studio Code (VS Code) was a software application that provide better developer experience to write, execute, and debug code. VS Code was also commonly known as part of an Integrated Development Environment (IDE), which was the go-to choice for developers to develop applications due to its comprehensive library of extensions that could further improve upon the developing environment and ease the inconveniences that could come from ever-growing development, such as Prettier that allowed developers to quickly format their documents to easy readability, or 'TypeScript React code snippets' that could quickly create a template of a React class or functional component onto the document which significantly reduced the repetition of rewriting the same code each time a new document got created. Additionally, VS Code had built-in terminals to quickly execute commands to initialise the server, install dependencies, or push to a version control service on the cloud directly from the IDE.

5.4.2 Insomnia

To assess the output of each API route, Insomnia was utilized to test these endpoints for possible different cases. Insomnia was a cross-platform API client for HTTP compatible protocols, especially for RESTful API, that eliminates the pain out of interacting with and designing, debugging, and testing APIs. The purpose of the software contributed to validating the server functioned as intended. For instance, HTTP requests with login credentials could be sent to assess whether the correct response alongside the created token was returned. A combination of API client and a Frontend could be utilized to validate and debug the functionalities of each endpoint to check for errors prior to production.

5.4.3 Git

Git was a distributed Version Control System (VCS) that tracked changes upon files initialised inside of the system. It allowed coordination work to be done among developers in collaboration between features, and provided a method to easily manage source code, track changes, and merge features into a single of branch.

5.4.4 GitHub

While Git was used to track changes made by local files, GitHub was utilised to store these changes in the local files on a repository, essentially worked as a hosting service for Git repositories. From this repository, a new branch would be created when a new feature was being developed in case the new feature could cause possible negatively affect(s) to existing components within the application. Thus, once all components worked accordingly, the branch would be later merged into the main branch.

5.4.5 GitHub Copilot

GitHub Copilot had become part of the development process due to how it could efficiently speed up the coding process. For instance, test cases could be quickly formed from simply prompting Copilot while on a React component to create these test cases. Furthermore, the ability to autocomplete the code by initiating a few characters of codes were incredibly timesaving as it provided less repetition in rewriting similar codes over time.

5.5 Sprint 1

The research of this proposal allowed the researcher to realise the potential appeal of the application in the educational intents. The existence of previous research and studies on the usage of AMT in relation to educational purposes and the range of possibilities students could use this tool for. The paper that majorly influenced this research was a study conducted by university students in Poland (Kasak, Jarina, & Chmulik, 2020), where it has gave valuable insights into the possible advantages of AMT and the additional resources that could be used to address such problems.

5.5.1 TabCNN

The best open-source model that currently existed was ‘TabCNN’. The existing model utilised the multi-task learning methodology to perform classification on each string of the guitar (six strings in total), and within that were 21 possible frets that were played in a moment in time (milliseconds). Additionally, the complexity that came with this model were the lack of documentation, and pre-trained weights that could be easily applied to the author’s own model. Thus, the reverse-engineering of the codes was performed to investigate into the functionalities of these existing codes.

GuitarSet

The dataset from GuitarSet was widely utilised among MIR problems including TabCNN where the dataset was utilised to tackle Guitar problems. The downloaded dataset was supplied with different types of folders that recorded guitar licks and rhythmic chords using a Hexaphonic pickup, shown in Figure 54. The pickup allowed the signal of a six-string guitar to be recorded individually rather than as a mixture of strings. Through the uses of the pickup and high-quality annotations done by the author of GuitarSet, which included chord labels, onset times, and genre information. These annotations offered crucial ground truth for training and evaluating machine learning algorithms and enabled the author to develop and validate model with high accuracy.

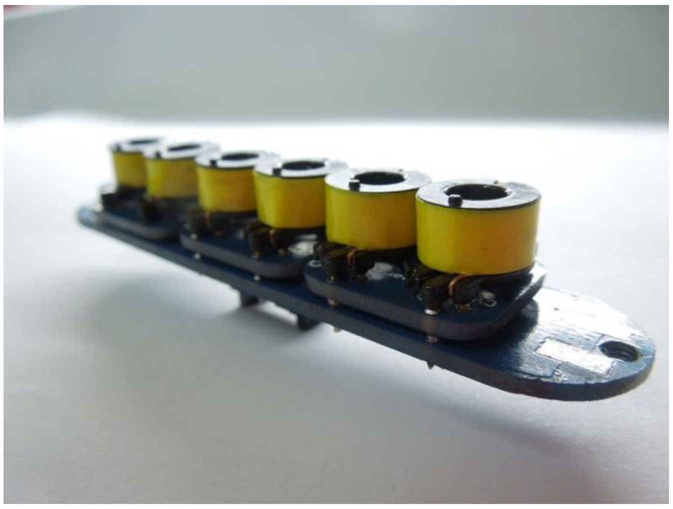
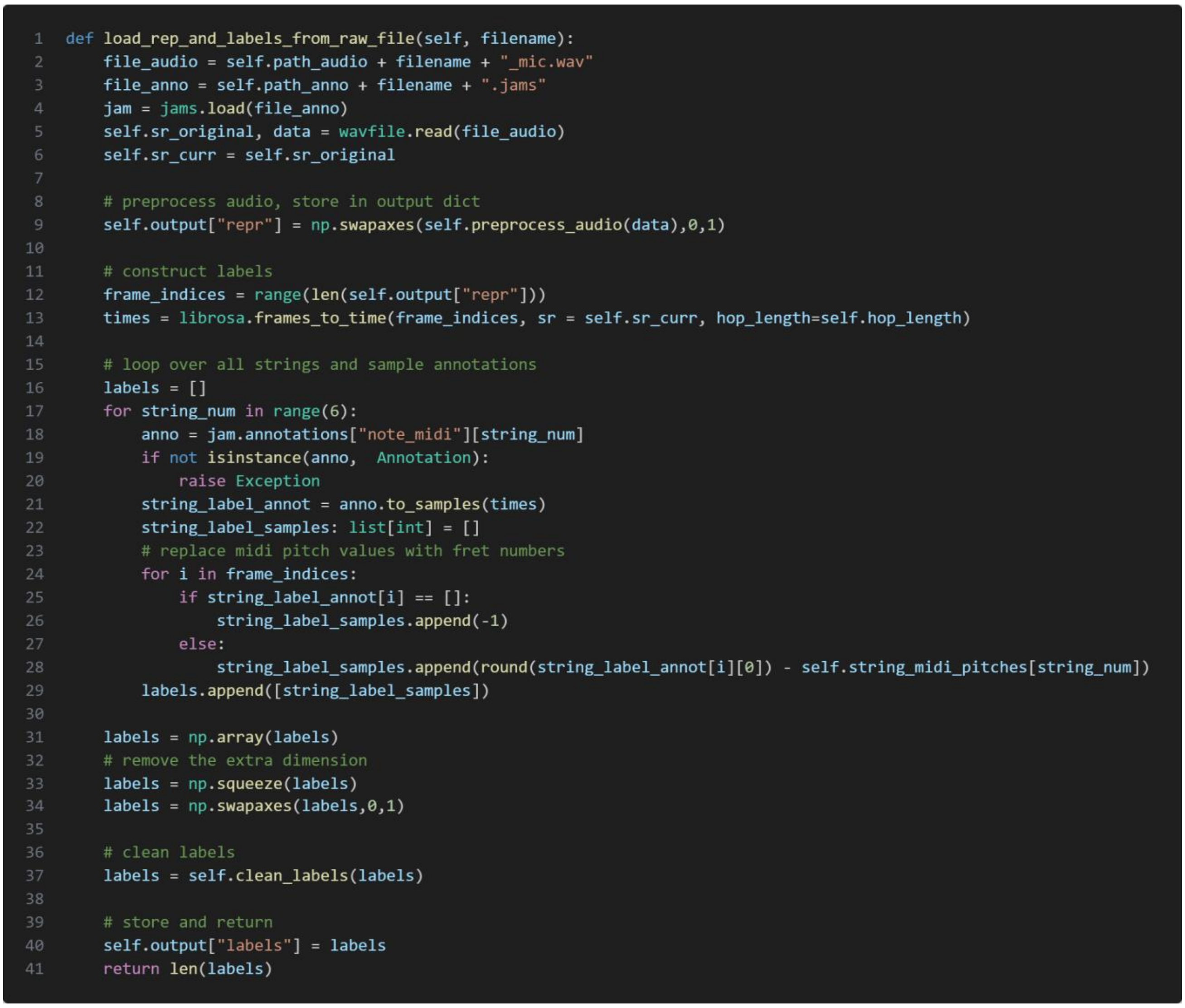


Figure 54 showed the fully assembled Hexophonic pickup (Guzman, 2013)

The script in the TabCNN repository was provided to convert audio into ‘spectrograms’ and attach labels of the onset times and chords to the spectrograms. The class function where different types of spectrograms can be selected as output, albeit the author chose the Constant-Q spectrogram for the output. In Figure 55 shown the function that converted the audio using Librosa and computed the matrices of frequencies, number of bins (the strength of the frequency in decibels), and the time in frames, and the frequencies. These outputs would be used as the input in the training/testing section which was better for CNN to compute due to the way each audio was labelled, where it required precise location of the time and frequency to enhance the accuracy of the predictions.

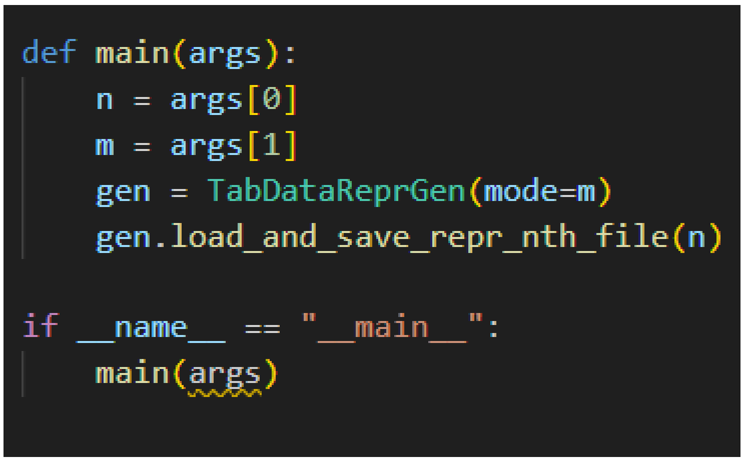


```python
def load_rep_and_labels_from_raw_file(self, filename):
    file_audio = self.path_audio + filename + "_mic.wav"
    file_anno = self.path_anno + filename + ".jams"
    jam = jams.load(file_anno)
    self.sr_original, data = wavfile.read(file_audio)
    self.sr_curr = self.sr_original

    # preprocess audio, store in output dict
    self.output["repr"] = np.swapaxes(self.preprocess_audio(data),0,1)

    # construct labels
    frame_indices = range(len(self.output["repr"]))
    times = librosa.frames_to_time(frame_indices, sr = self.sr_curr, hop_length=self.hop_length)

    # loop over all strings and sample annotations
    labels = []
    for string_num in range(6):
        anno = jam.annotations["note_midi"][string_num]
        if not isinstance(anno,  Annotation):
            raise Exception
        string_label_annot = anno.to_samples(times)
        string_label_samples: list[int] = []
        # replace midi pitch values with fret numbers
        for i in frame_indices:
            if string_label_annot[i] == []:
                string_label_samples.append(-1)
            else:
                string_label_samples.append(round(string_label_annot[i][0]) - self.string_midi_pitches[string_num])
        labels.append([string_label_samples])

    labels = np.array(labels)
    # remove the extra dimension
    labels = np.squeeze(labels)
    labels = np.swapaxes(labels,0,1)

    # clean labels
    labels = self.clean_labels(labels)

    # store and return
    self.output["labels"] = labels
    return len(labels)
```

Figure 55 shows the function that converted audio into a spectrogram, attach labels to each time frame, and saved the output as a .npz file.

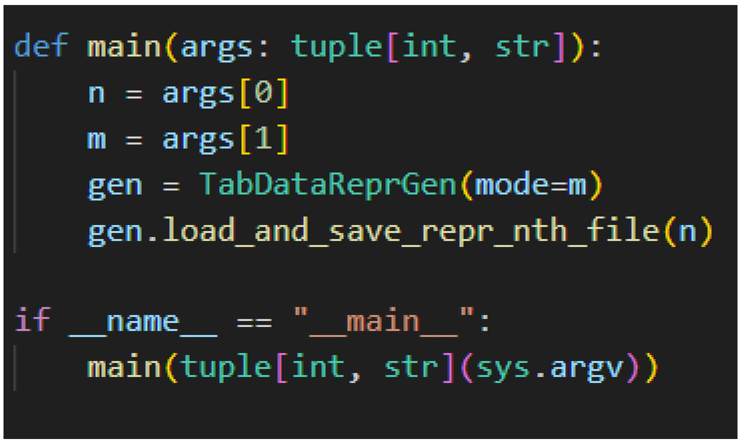
The labels would be loaded from the jam file extension and attached to a window frame of 512 length (which converted to milliseconds of audio) in the spectrogram. The output of this would be saved as a .npz file and later input into the CNN architecture to train the model.

Unfortunately, when it came to executing the code, a rebranding of older functionalities was needed due to older versions of Python, Keras, and TensorFlow, which had led to errors such as deprecation of functions, and multi-processing issue with old syntax seen in Figure 56 and 57. The issue had been resolved by declaring the type of the arguments variable to prevent type errors, where the script would refuse to execute due to undefined variable. By using a revised syntax to take the arguments from the Command Line Interface (CLI), the script was able to execute using the parameter specified in the Bash file.



```python
def main(args):
    n = args[0]
    m = args[1]
    gen = TabDataReprGen(mode=m)
    gen.load_and_save_repr_nth_file(n)

if __name__ == "__main__":
    main(args)
```

Figure 56 shows the deprecated methods due to Python 2.7 in the script to convert audio into CQT spectrogram.



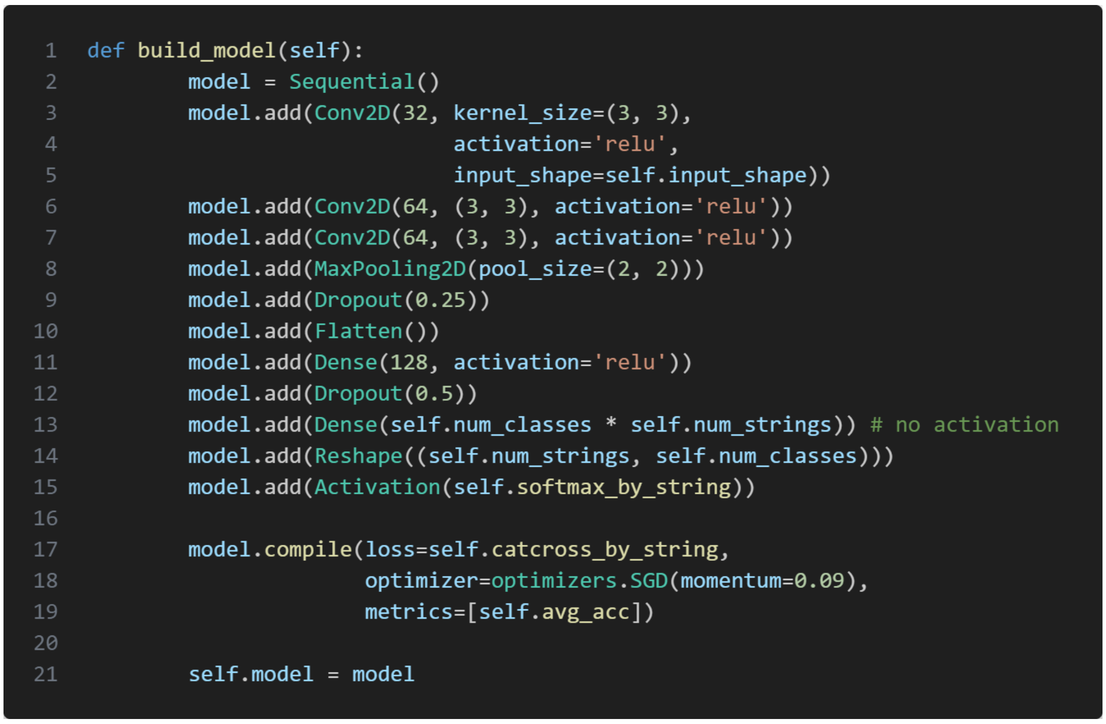
```python
def main(args: tuple[int, str]):
    n = args[0]
    m = args[1]
    gen = TabDataReprGen(mode=m)
    gen.load_and_save_repr_nth_file(n)

if __name__ == "__main__":
    main(tuple[int, str](sys.argv))
```

Figure 57 shows the new argument used when the program was launched.

Training the existing model

The model, shown in Figure 58, consisted of a layer which takes in an input (i.e. CQT matrices) and pass down through each Convolutional layer to extract possible features with an additional Max-pooling layer and Dropout layer to prevent over-fitting the model and hindered it from too tightly coupling with the training data, and may result in lower accuracy. Adjacently, the fully connected layer (Dense layer) was put in place to convert the extracted features into a data type that could be understood by the Activation layer to determine the prediction of the waveform using the SoftMax function due to its suitability with classifying group tasks.

Lastly, the loss function called categorical cross entropy was used to help optimise the model. This worked as a sort of target, wherein the model would aim to minimise the losses curated by the sum of the six strings' inaccuracy, along with the optimiser called 'Stochastic gradient descent' to serve as the guide for the model to find their way down the ravine.



```python
def build_model(self):
        model = Sequential()
        model.add(Conv2D(32, kernel_size=(3, 3),
                             activation='relu',
                             input_shape=self.input_shape))
        model.add(Conv2D(64, (3, 3), activation='relu'))
        model.add(Conv2D(64, (3, 3), activation='relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))
        model.add(Flatten())
        model.add(Dense(128, activation='relu'))
        model.add(Dropout(0.5))
        model.add(Dense(self.num_classes * self.num_strings)) # no activation
        model.add(Reshape((self.num_strings, self.num_classes)))
        model.add(Activation(self.softmax_by_string))

        model.compile(loss=self.catcross_by_string,
                      optimizer=optimizers.SGD(momentum=0.09),
                      metrics=[self.avg_acc])

        self.model = model
```

Figure 58 shows the CNN architecture behind TabCNN.

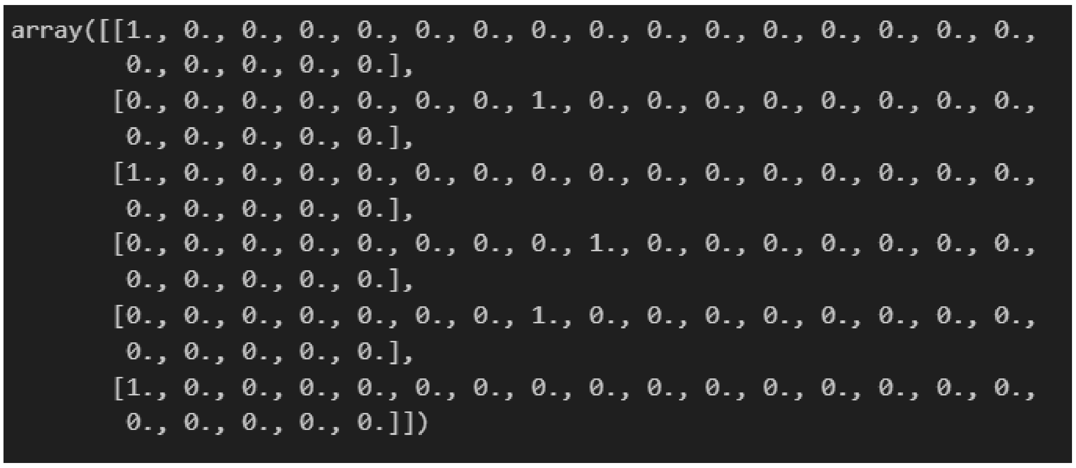
From investigating a similar application researched by Jadhav et. al (2022), where better prediction result was achieved through an optimiser called 'SGD with momentum', and the momentum rate was configured to 0.09. The change reflected in model to output overall accuracy of 89.8 percents from the original 80.3 percents.

Utilisation of the model for predictions

Once the optimal training results were achieved, the utilisation of the tool was also proven to be a challenging aspect. This was due to the lack of documentation provided by the author of TabCNN. The only documentation existed as a form of comments atop some function, and the only option was to read through each function and attempt to figure out their processes.

The most useful file was the TabDataReprGen.py, which provided the methodologies on how each audio input was processed and fed into the input of the CNN. For example:

- Preprocess_audio function – This function converted the waveform into CQT matrices. Because of this, the process to correctly feed the CQT into the model was able to be achieved and recreated for the author’s own application.
- Load_rep_and_labels_from_raw_file – allowed the author to get an insight into how TabCNN would output its prediction and the way to process it to an array shape of (6, 21), where the first element was the strings and the second was the frets as illustrated in Figure 59.



```
array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0.]])
```

Figure 59 illustrated the first element in the output array of the prediction.

5.5.2 Paper Prototype

In order to offer the most optimal experience when it comes to users utilising the application, a wireframe had been developed to roughly outline the basic features and its essential requirements that may be needed to give the users the best browsing experience as possible.

Figure 60 illustrated the paper prototype of the application.

5.6 Sprint 2

5.6.1 Goal

The initial objectives for sprint two was primarily to focus on the design section and implementing wireframes to help initiate the basic structure that the application would need.

5.6.2 Lo-Fi Prototype

From the initial prototype constructed on paper, the author had developed a better insight into the basic components the application would be required to function. The prototypes helped the author to visualise the clearer aspects of the UIs and hinted the libraries and components that would be needed to build the application.

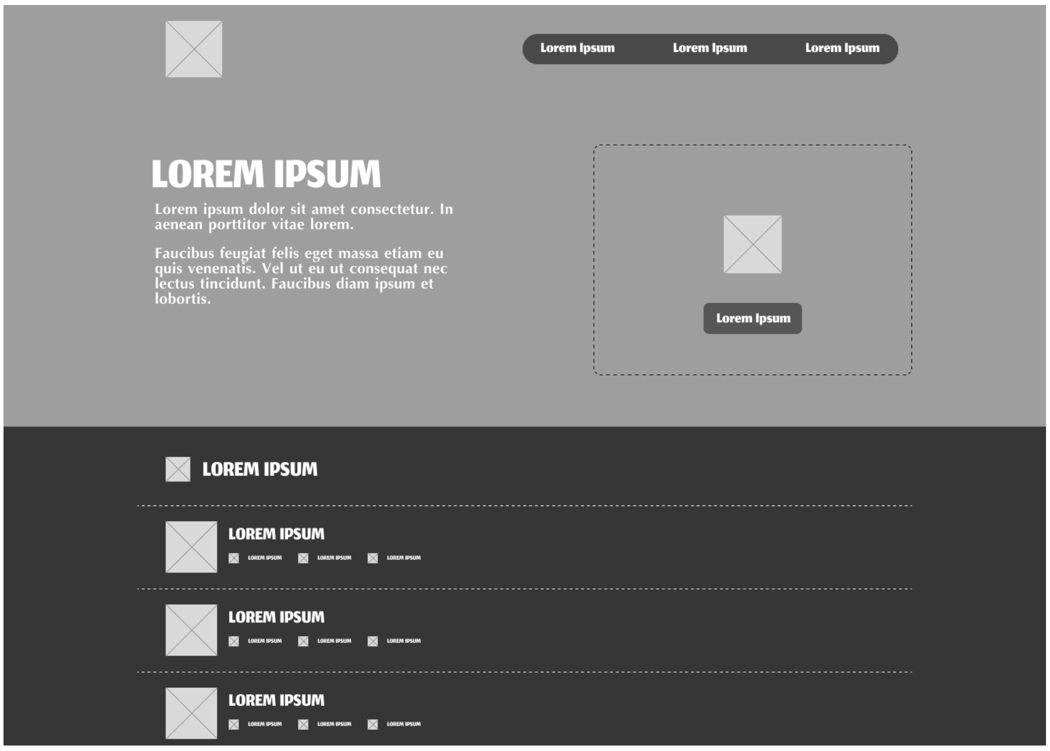

Lorem Ipsum
Lorem Ipsum
Lorem Ipsum
LOREM IPSUM
Lorem ipsum dolor sit amet consectetur. In aenean porttitor vitae lorem.
Faucibus feugiat felis eget massa etiam eu quis venenatis. Vel ut eu ut consequat nec lectus tincidunt. Faucibus diam ipsum et lobortis.
Lorem Ipsum
LOREM IPSUM
LOREM IPSUM
LOREM IPSUM
LOREM IPSUM

Figure 61 shows the Lo-Fi prototype of homepage.

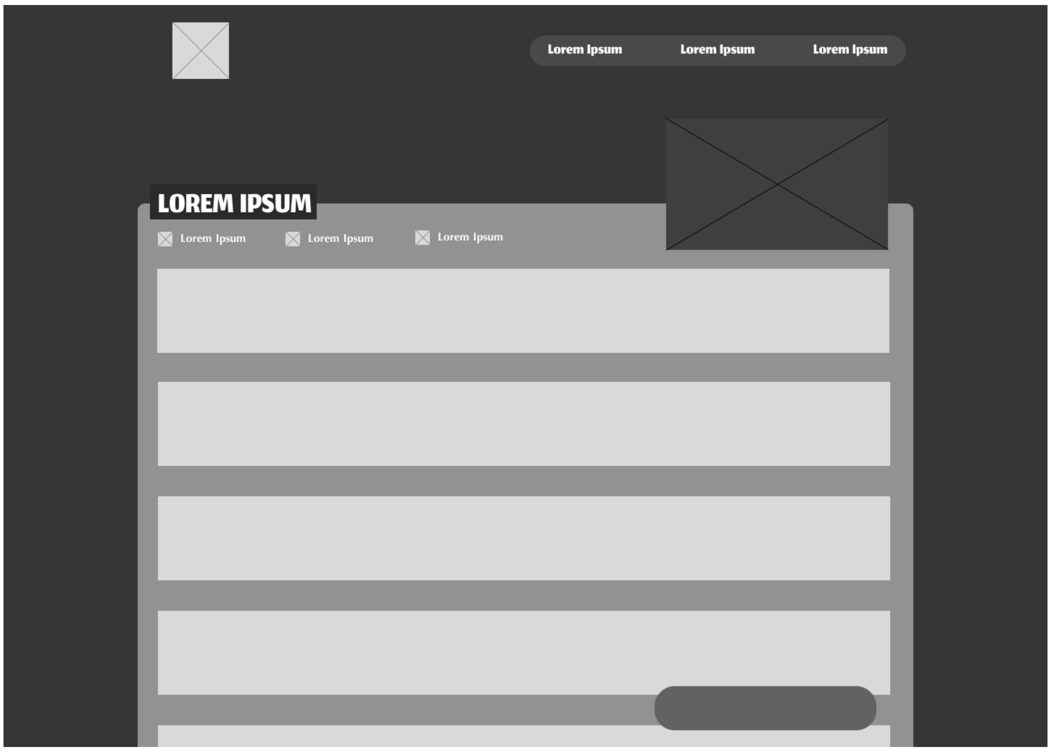


Figure 62 shows the Lo-Fi prototype of result page.

5.6.3 Hi-Fi Prototype

The development of the High-fidelity wireframes allowed the colour palette and icons that would be utilised to filled in any white spaces and create a sense of togetherness. While there were some adjustments to fonts, tones, background, and overall colours, in an attempt to improve readability, accessibility, and better user experiences.

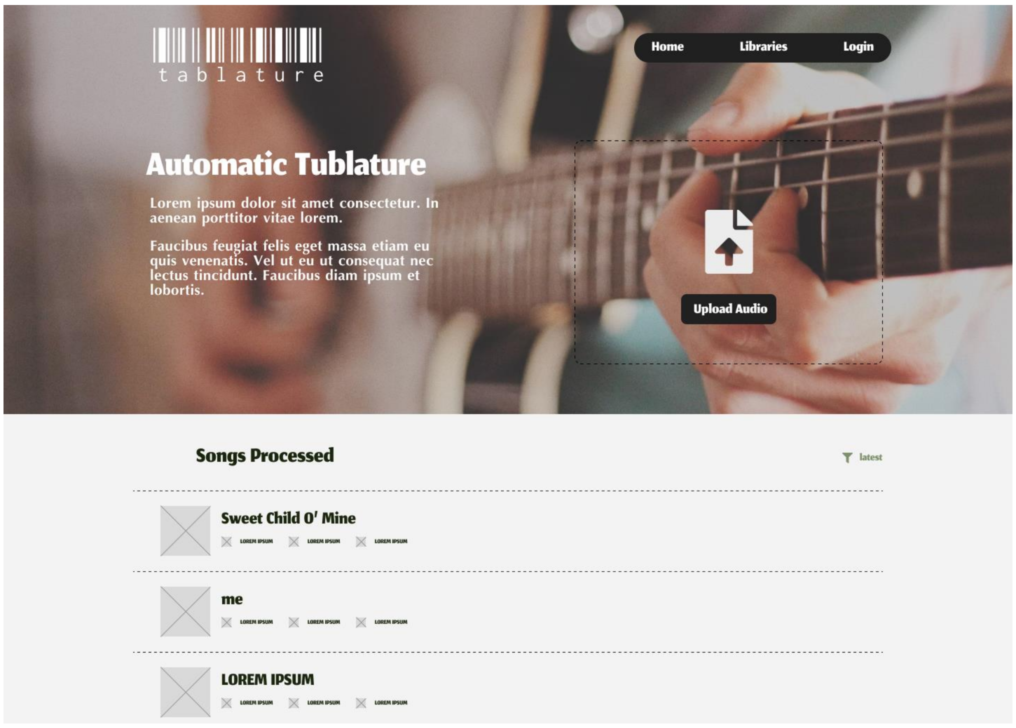

tablature
Home
Libraries
Login
Automatic Tublature
Lorem ipsum dolor sit amet consectetur. In aenean porttitor vitae lorem.
Faucibus feugiat felis eget massa etiam eu quis venenatis. Vel ut eu ut consequat nec lectus tincidunt. Faucibus diam ipsum et lobortis.
Upload Audio
Songs Processed
latest
Sweet Child O' Mine
me
LOREM IPSUM

Figure 63 shows the Hi-Fi Prototype of the homepage.

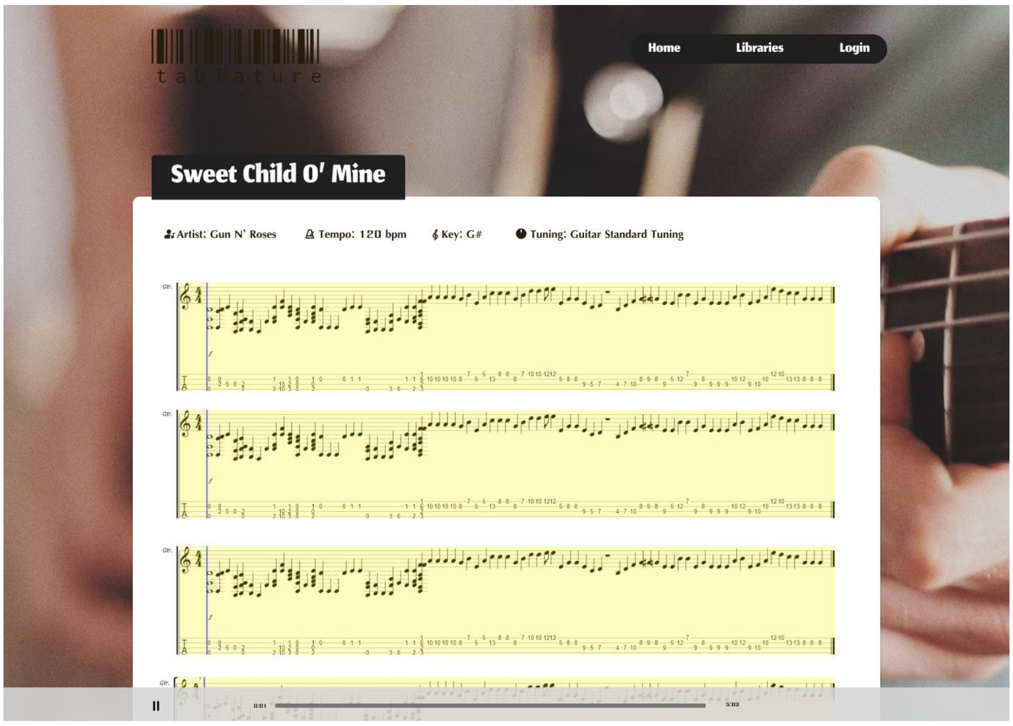


Figure 64 shows the Hi-Fi prototype of the result page.

5.7 Sprint 3

5.7.1 Goal

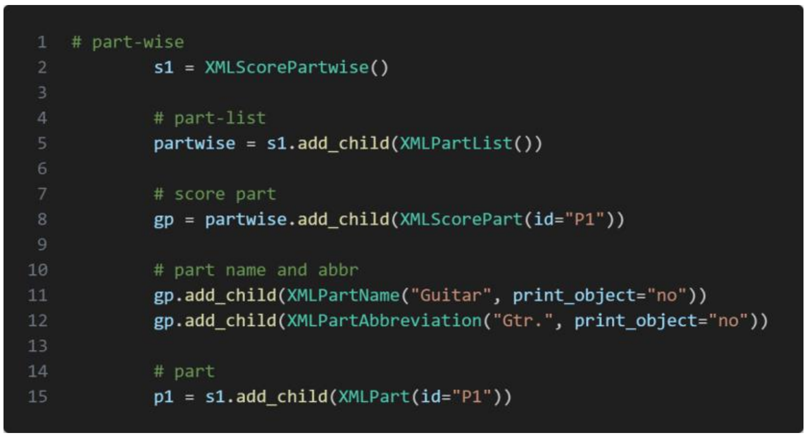
The goal for this Sprint was the creation of the back-end API to support the Front-end application. The objective included the attempt to optimise the DL model, research a way to export the prediction in such a way that can be served to the Front-end, and picking the framework to serve the RESTful API. This phase of the development aimed to optimise the feasible features and explore the capabilities of the proposal.

5.7.2 Prediction Conversion and Results Mapping

Music Score File

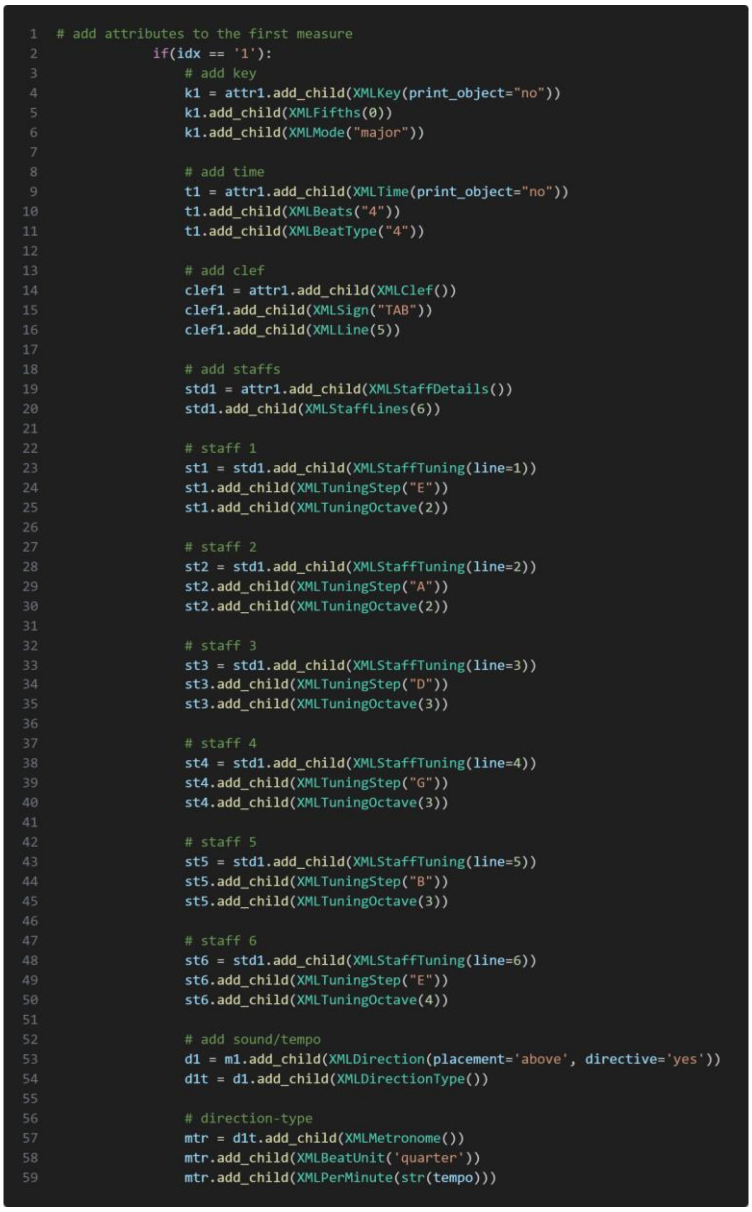
In order to use the results from the conversion for the visualisation, the converted predictions first needed to get compiled to a file that could be used in a Front-end or other software such as Muse Score, and GuitarPro (GP). The common file extension shared among these applications were GP5, MIDI, and MusicXML extensions. Although, the MIDI would be an easier option to increment notes from the predictions and had lots of documentation, it did not have the ability to specify the locations of the frets and the guitar strings. The GP5 files did contain the locations of the frets and strings, but the way to add new notes were not documented. Thus, the author decided to use MusicXML due to its easier implication to use the package.

As illustrated in Figure 65 and 66, the score partwise needed to be initialised to create a MusicXML score. The package worked like a DOM that linked each instrumental part to the score, and nested within these parts were measure with an identifier (i.e. number) to signifies their placement in the score. The first measure always required an attribute element to provide metadata to the instrument, which included the key signature, time signature, tuning, tempo, and clef signature.



```python
# part-wise
        s1 = XMLScorePartwise()

        # part-list
        partwise = s1.add_child(XMLPartList())

        # score part
        gp = partwise.add_child(XMLScorePart(id="P1"))

        # part name and abbr
        gp.add_child(XMLPartName("Guitar", print_object="no"))
        gp.add_child(XMLPartAbbreviation("Gtr.", print_object="no"))

        # part
        p1 = s1.add_child(XMLPart(id="P1"))
```

Figure 65 illustrated the first part of how MusicXML were implemented in Python.

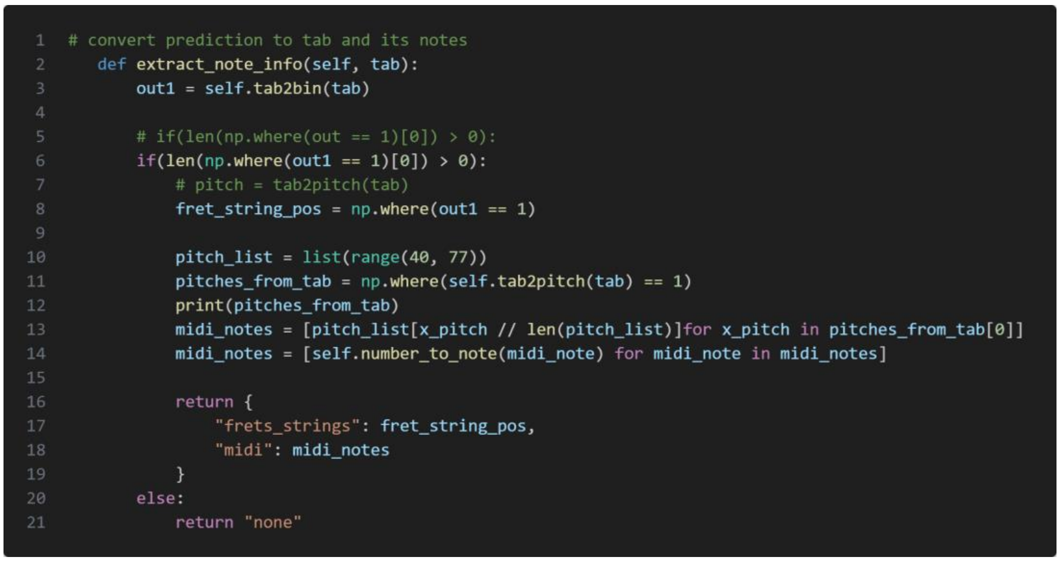


```python
# add attributes to the first measure
            if(idx == '1'):
                # add key
                k1 = attr1.add_child(XMLKey(print_object="no"))
                k1.add_child(XMLFifths(0))
                k1.add_child(XMLMode("major"))

                # add time
                t1 = attr1.add_child(XMLTime(print_object="no"))
                t1.add_child(XMLBeats("4"))
                t1.add_child(XMLBeatType("4"))

                # add clef
                clef1 = attr1.add_child(XMLClef())
                clef1.add_child(XMLSign("TAB"))
                clef1.add_child(XMLLine(5))

                # add staffs
                std1 = attr1.add_child(XMLStaffDetails())
                std1.add_child(XMLStaffLines(6))

                # staff 1
                st1 = std1.add_child(XMLStaffTuning(line=1))
                st1.add_child(XMLTuningStep("E"))
                st1.add_child(XMLTuningOctave(2))

                # staff 2
                st2 = std1.add_child(XMLStaffTuning(line=2))
                st2.add_child(XMLTuningStep("A"))
                st2.add_child(XMLTuningOctave(2))

                # staff 3
                st3 = std1.add_child(XMLStaffTuning(line=3))
                st3.add_child(XMLTuningStep("D"))
                st3.add_child(XMLTuningOctave(3))

                # staff 4
                st4 = std1.add_child(XMLStaffTuning(line=4))
                st4.add_child(XMLTuningStep("G"))
                st4.add_child(XMLTuningOctave(3))

                # staff 5
                st5 = std1.add_child(XMLStaffTuning(line=5))
                st5.add_child(XMLTuningStep("B"))
                st5.add_child(XMLTuningOctave(3))

                # staff 6
                st6 = std1.add_child(XMLStaffTuning(line=6))
                st6.add_child(XMLTuningStep("E"))
                st6.add_child(XMLTuningOctave(4))

                # add sound/tempo
                d1 = m1.add_child(XMLDirection(placement='above', directive='yes'))
                d1t = d1.add_child(XMLDirectionType())

                # direction-type
                mtr = d1t.add_child(XMLMetronome())
                mtr.add_child(XMLBeatUnit('quarter'))
                mtr.add_child(XMLPerMinute(str(tempo)))
```

Figure 66 illustrated the second part of how MusicXML were implemented in Python.

MusicXML format

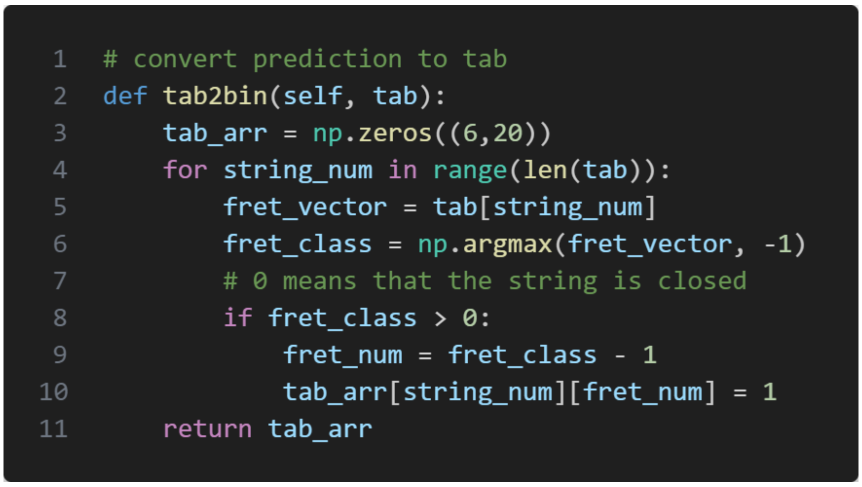
For MusicXML to interpret its score and validate for exportation, the score needed to provide the string and fret numbers, as well as the corresponding MIDI value. The function was written to extract these values called ‘extract_note_info’ function illustrated in Figure 67. This computed the values from each element in the prediction through the use of a For loop, where the string and fret was achieved by the ‘tab2bin’ function.



```python
# convert prediction to tab and its notes
   def extract_note_info(self, tab):
       out1 = self.tab2bin(tab)

       # if(len(np.where(out == 1)[0]) > 0):
       if(len(np.where(out1 == 1)[0]) > 0):
           # pitch = tab2pitch(tab)
           fret_string_pos = np.where(out1 == 1)

           pitch_list = list(range(40, 77))
           pitches_from_tab = np.where(self.tab2pitch(tab) == 1)
           print(pitches_from_tab)
           midi_notes = [pitch_list[x_pitch // len(pitch_list)]for x_pitch in pitches_from_tab[0]]
           midi_notes = [self.number_to_note(midi_note) for midi_note in midi_notes]

           return {
               "frets_strings": fret_string_pos,
               "midi": midi_notes
           }
       else:
           return "none"
```

Figure 67 illustrated the function that map predictions into an object to input into the MusicXML score.

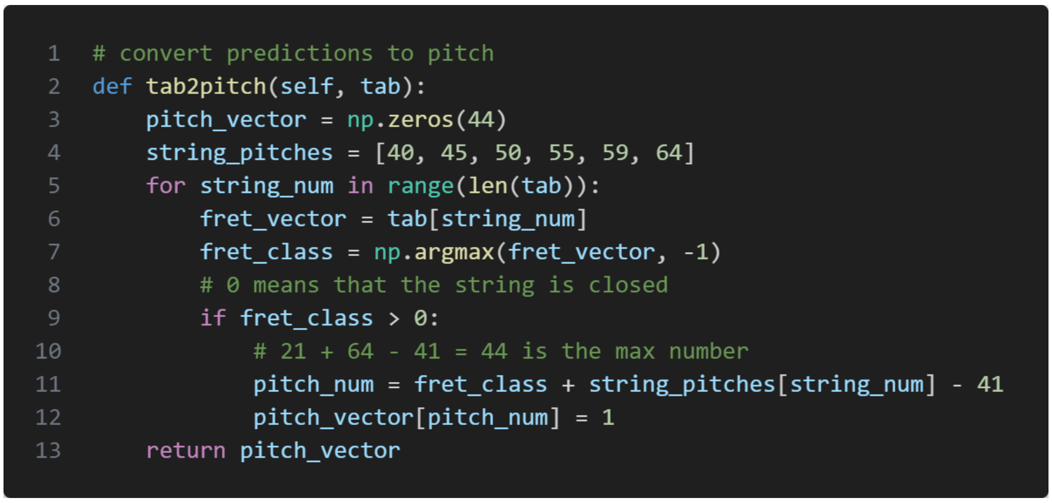
The function took in each output of the prediction where the initial matrix was similar to Figure 68, though each output was in a 6-dimensional array consisted of float values. In order to achieved similar result, the array of each string needed to be passed through np.argmax() function, where it would be essentially iterated through each element to find the largest value and returned its index. Once the index was found, an empty array of matrix (6, 20) consisted of zeros was created to insert the value of one into the location of the index. This function would be returned to the initial function mentioned above and the location of string and fret was able to be retrieved.



```python
# convert prediction to tab
def tab2bin(self, tab):
    tab_arr = np.zeros((6,20))
    for string_num in range(len(tab)):
        fret_vector = tab[string_num]
        fret_class = np.argmax(fret_vector, -1)
        # 0 means that the string is closed
        if fret_class > 0:
            fret_num = fret_class - 1
            tab_arr[string_num][fret_num] = 1
    return tab_arr
```

Figure 68 illustrated the function that converted the matrix of floats into matrix of zeros and ones.

The corresponding MIDI value could be computed using the same method as tab2bin() as shown in Figure 69, the highest value was achieved through np.argmax() and an empty array of length forty-four was created to house the position of the MIDI.



```python
# convert predictions to pitch
def tab2pitch(self, tab):
    pitch_vector = np.zeros(44)
    string_pitches = [40, 45, 50, 55, 59, 64]
    for string_num in range(len(tab)):
        fret_vector = tab[string_num]
        fret_class = np.argmax(fret_vector, -1)
        # 0 means that the string is closed
        if fret_class > 0:
            # 21 + 64 - 41 = 44 is the max number
            pitch_num = fret_class + string_pitches[string_num] - 41
            pitch_vector[pitch_num] = 1
    return pitch_vector
```

Figure 69 illustrated the function that convert the result of the prediction to string and fret combination.

The reason there was only forty-four needed for mapping corresponding string and fret was due to the MIDI values of the guitar, illustrated in Figure 70. The MIDI values for fret positions started from value of forty to seventy-six, which added up to the length of forty-four.

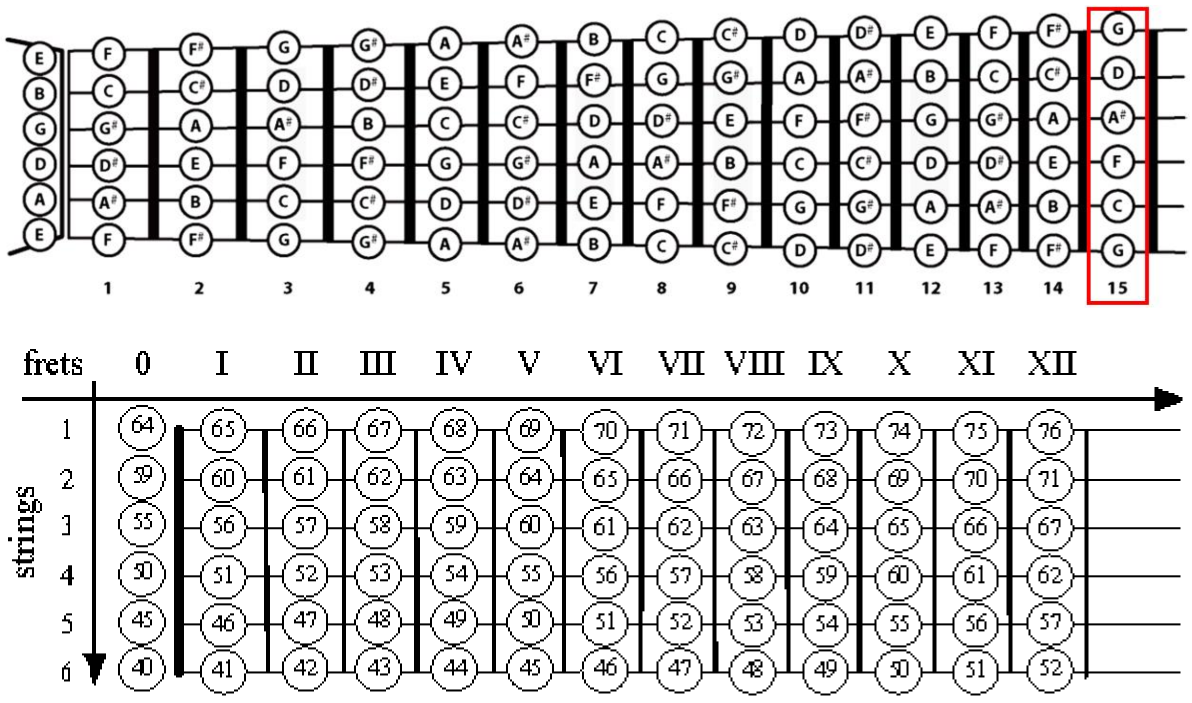
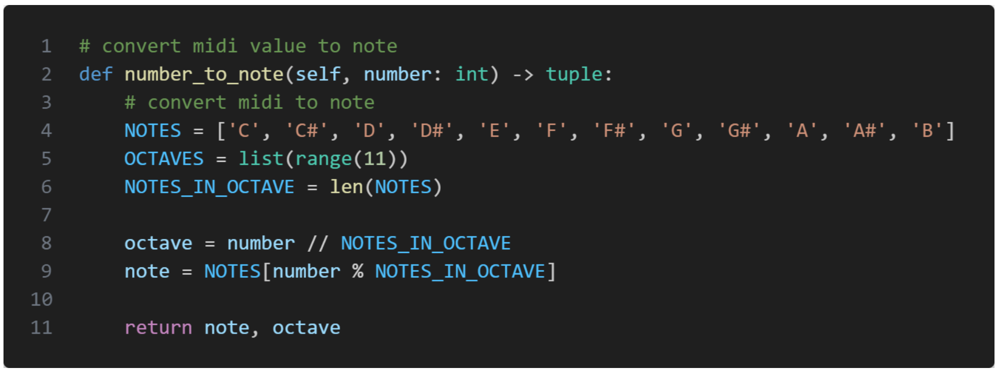


Figure 70 illustrated the conversion table from fret positions to MIDI notes.

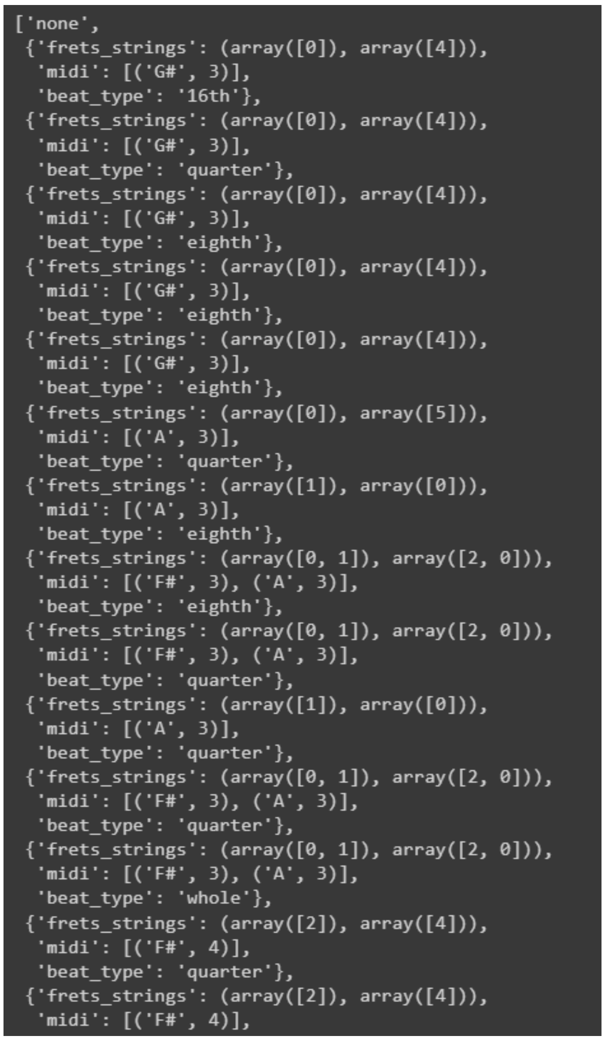
The tab2pitch function returned the array of MIDI notes, and these notes would be pipelined into the number_to_note function to convert them into the name of the note illustrated in Figure 71.



```python
# convert midi value to note
def number_to_note(self, number: int) -> tuple:
    # convert midi to note
    NOTES = ['C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#', 'A', 'A#', 'B']
    OCTAVES = list(range(11))
    NOTES_IN_OCTAVE = len(NOTES)

    octave = number // NOTES_IN_OCTAVE
    note = NOTES[number % NOTES_IN_OCTAVE]

    return note, octave
```

Figure 71 illustrated the function that convert the MIDI value into the name of the note.

Finally, the higher order function performed these low-level functions and returned all the values as an array of objects as shown in Figure 72 below.

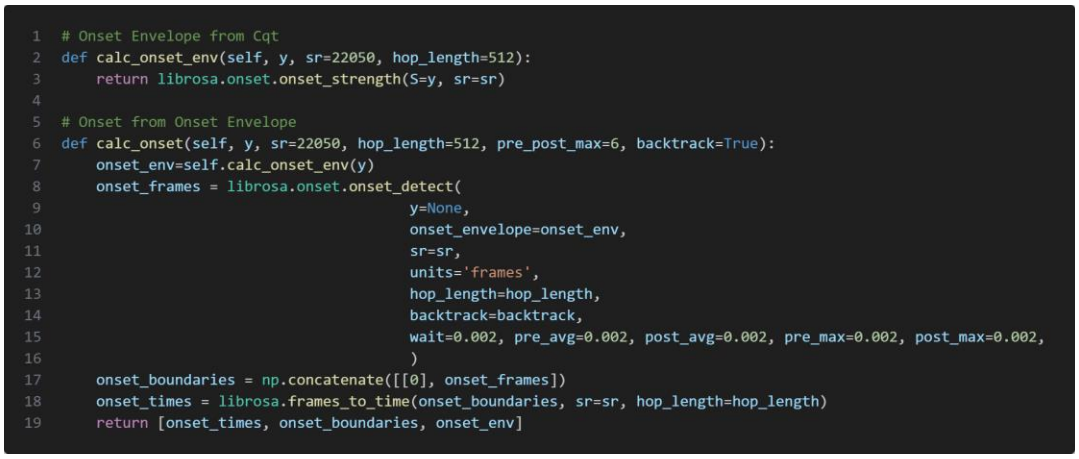


```
['none',
 {'frets_strings': (array([0]), array([4])),
  'midi': [('G#', 3)],
  'beat_type': '16th'},
 {'frets_strings': (array([0]), array([4])),
  'midi': [('G#', 3)],
  'beat_type': 'quarter'},
 {'frets_strings': (array([0]), array([4])),
  'midi': [('G#', 3)],
  'beat_type': 'eighth'},
 {'frets_strings': (array([0]), array([4])),
  'midi': [('G#', 3)],
  'beat_type': 'eighth'},
 {'frets_strings': (array([0]), array([4])),
  'midi': [('G#', 3)],
  'beat_type': 'eighth'},
 {'frets_strings': (array([0]), array([5])),
  'midi': [('A', 3)],
  'beat_type': 'quarter'},
 {'frets_strings': (array([1]), array([0])),
  'midi': [('A', 3)],
  'beat_type': 'eighth'},
 {'frets_strings': (array([0, 1]), array([2, 0])),
  'midi': [('F#', 3), ('A', 3)],
  'beat_type': 'eighth'},
 {'frets_strings': (array([0, 1]), array([2, 0])),
  'midi': [('F#', 3), ('A', 3)],
  'beat_type': 'quarter'},
 {'frets_strings': (array([1]), array([0])),
  'midi': [('A', 3)],
  'beat_type': 'quarter'},
 {'frets_strings': (array([0, 1]), array([2, 0])),
  'midi': [('F#', 3), ('A', 3)],
  'beat_type': 'quarter'},
 {'frets_strings': (array([0, 1]), array([2, 0])),
  'midi': [('F#', 3), ('A', 3)],
  'beat_type': 'whole'},
 {'frets_strings': (array([2]), array([4])),
  'midi': [('F#', 4)],
  'beat_type': 'quarter'},
 {'frets_strings': (array([2]), array([4])),
  'midi': [('F#', 4)],
```

Figure 72 shows the positions of the frets and strings and its corresponding MIDI notes.

Onset Detections

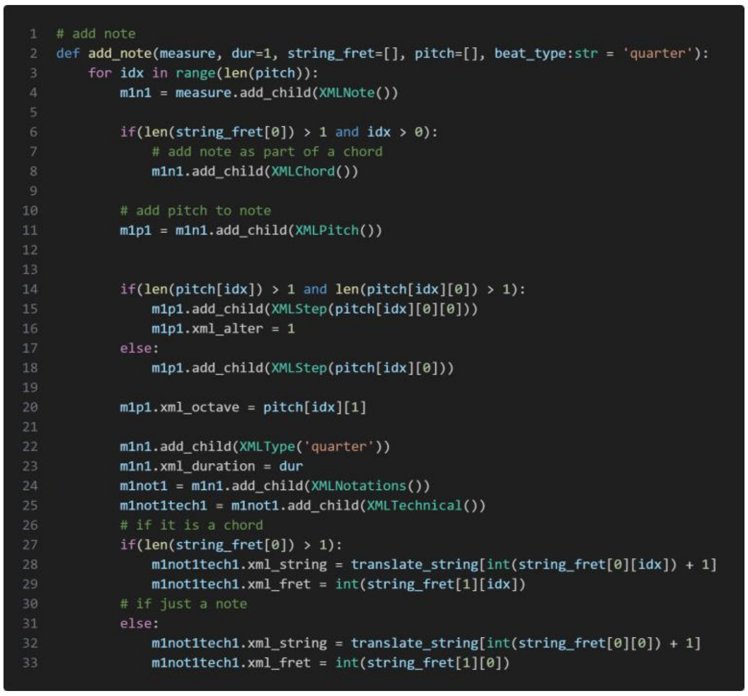
To correctly locate the timing when the note was being played, a detection algorithm needed to be set in place to capture this information from the waveform. Fortunately, Librosa provided a function to retrieve these timings called ‘onset’ detection shown in Figure 73. The function needed an onset strength to properly compute the cues. The onset strength computed the increase in energy of the sound (its loudness in dB) from each time frame and fed into the librosa.onset_detect which returned the time frames where the note was played.



```python
# Onset Envelope from Cqt
def calc_onset_env(self, y, sr=22050, hop_length=512):
    return librosa.onset.onset_strength(S=y, sr=sr)

# Onset from Onset Envelope
def calc_onset(self, y, sr=22050, hop_length=512, pre_post_max=6, backtrack=True):
    onset_env=self.calc_onset_env(y)
    onset_frames = librosa.onset.onset_detect(
                                    y=None,
                                    onset_envelope=onset_env,
                                    sr=sr,
                                    units='frames',
                                    hop_length=hop_length,
                                    backtrack=backtrack,
                                    wait=0.002, pre_avg=0.002, post_avg=0.002, pre_max=0.002, post_max=0.002,
                                    )
    onset_boundaries = np.concatenate([[0], onset_frames])
    onset_times = librosa.frames_to_time(onset_boundaries, sr=sr, hop_length=hop_length)
    return [onset_times, onset_boundaries, onset_env]
```

Figure 73 illustrated the function that detect onsets and returned the frame, time (in milliseconds), and its onset strength.

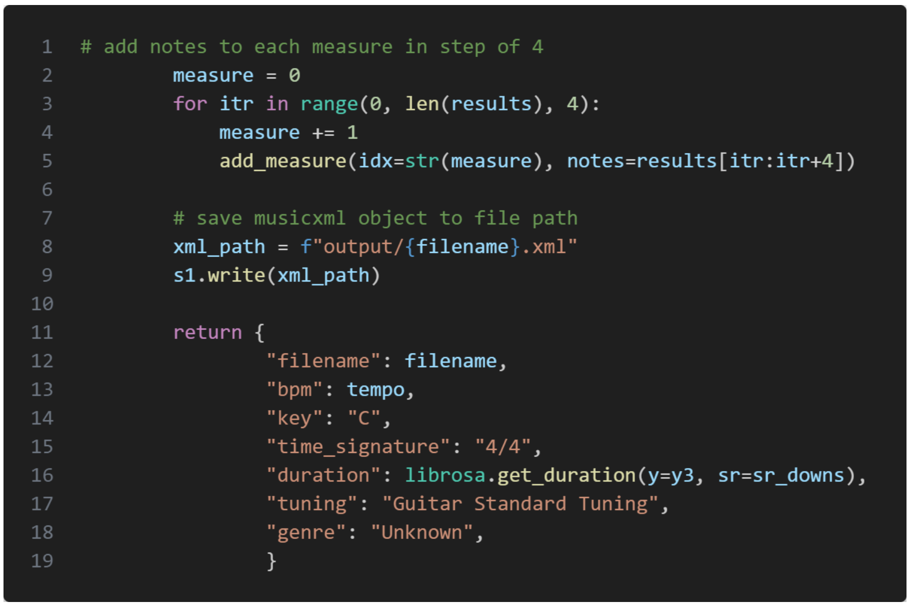
In each measure, the notes could be added from executing the above function, where the returned values would be fed into another function that checked if it was a rest note, an actual note, or a chord. This was simply done by validating whether the element existed to return a rest note, if the frets_strings array contained only one element for a note, or the array contained more than one element for the values to be added into the score as a chord. Then the corresponding MIDI note would also get appended to each note, where the result would be inserted into each bar in the measure as shown in Figure 74.



```python
# add note
def add_note(measure, dur=1, string_fret=[], pitch=[], beat_type:str = 'quarter'):
    for idx in range(len(pitch)):
        m1n1 = measure.add_child(XMLNote())

        if(len(string_fret[0]) > 1 and idx > 0):
            # add note as part of a chord
            m1n1.add_child(XMLChord())

        # add pitch to note
        m1p1 = m1n1.add_child(XMLPitch())


        if(len(pitch[idx]) > 1 and len(pitch[idx][0]) > 1):
            m1p1.add_child(XMLStep(pitch[idx][0][0]))
            m1p1.xml_alter = 1
        else:
            m1p1.add_child(XMLStep(pitch[idx][0]))

        m1p1.xml_octave = pitch[idx][1]

        m1n1.add_child(XMLType('quarter'))
        m1n1.xml_duration = dur
        m1not1 = m1n1.add_child(XMLNotations())
        m1not1tech1 = m1not1.add_child(XMLTechnical())
        # if it is a chord
        if(len(string_fret[0]) > 1):
            m1not1tech1.xml_string = translate_string[int(string_fret[0][idx]) + 1]
            m1not1tech1.xml_fret = int(string_fret[1][idx])
        # if just a note
        else:
            m1not1tech1.xml_string = translate_string[int(string_fret[0][0]) + 1]
            m1not1tech1.xml_fret = int(string_fret[1][0])
```

Figure 74 shows the implementation that was used to add notes to the measure.

As of now, there was no time signature detection in place, instead the time signature was hard coded to always be 4/4 signature, where the iteration was incremented in a step of four to complement this, meaning there would be four notes would be inserted for incrementing bar. Furthermore, MusicXML also required each note to be described alongside its beat type, meaning the score needed to know how fast that note was played for. Now, the type was hard coded as a ‘quarter’ note. Finally, the score would then be exported and stored in a local directory as illustrated in Figure 75.



```python
# add notes to each measure in step of 4
        measure = 0
        for itr in range(0, len(results), 4):
            measure += 1
            add_measure(idx=str(measure), notes=results[itr:itr+4])

        # save musicxml object to file path
        xml_path = f"output/{filename}.xml"
        s1.write(xml_path)

        return {
                "filename": filename,
                "bpm": tempo,
                "key": "C",
                "time_signature": "4/4",
                "duration": librosa.get_duration(y=y3, sr=sr_downs),
                "tuning": "Guitar Standard Tuning",
                "genre": "Unknown",
                }
```

Figure 75 shows For loop that iterated through each result in the prediction to add measure and append notes to each measure. The score was then saved to the local directory.

5.7.3 Serverless Functions

During this sprint, there was an attempt to initiate serverless functions to await requests from the client. The purpose of setting up the functions was done to minimise costs of having a VM container running to serve the API to the client.

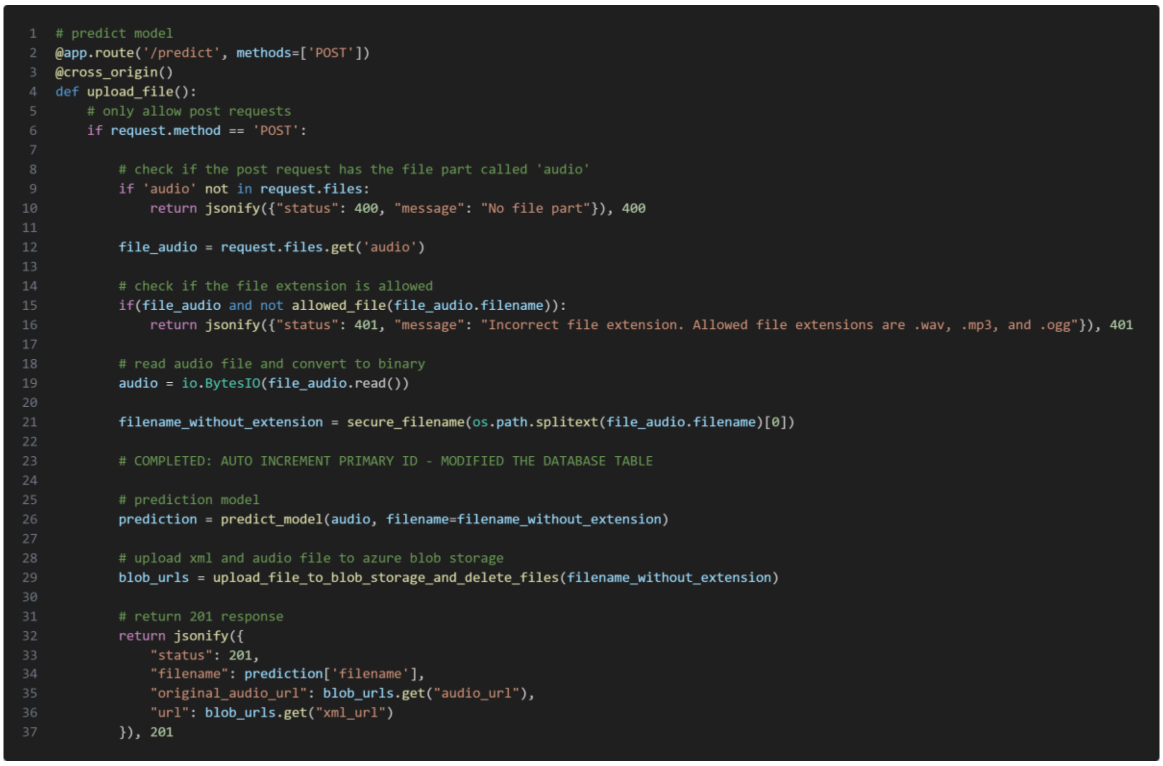
A multiple of cloud services was explored to fulfil this objective i.e. AWS, GCP, and Azure. Though, upon the search through community forums, and Stack Overflow, it was proven futile to implement the API as a serverless function. This was due to the package, Librosa, used by the author to analyse audio. The cloud provider such as AWS only allowed the maximum deployment package to be under 250 megabytes (MB), by which Librosa alone would exceed the specified limitation, let alone all the packages needed to be installed on a

Lambda function. Similarly, Azure's operating system (OS) image of the VM that used to operate the function did not come pre-installed with a C++ library used by the package. The only solution to resolve the issue was to install a custom OS image to the server to install the missing library. Albeit it would completely reconcile the purpose of a serverless function. Thus, the author decided to design a back-end server using a web-application framework called Flask.

5.7.4 API Endpoints

Flask was utilised due to its popularity and well-written documentation. The majority of the structure shared similarities with ExpressJS and Laravel, which the author had extensive experience and familiarity with, though there were numerous functionalities that were differed.

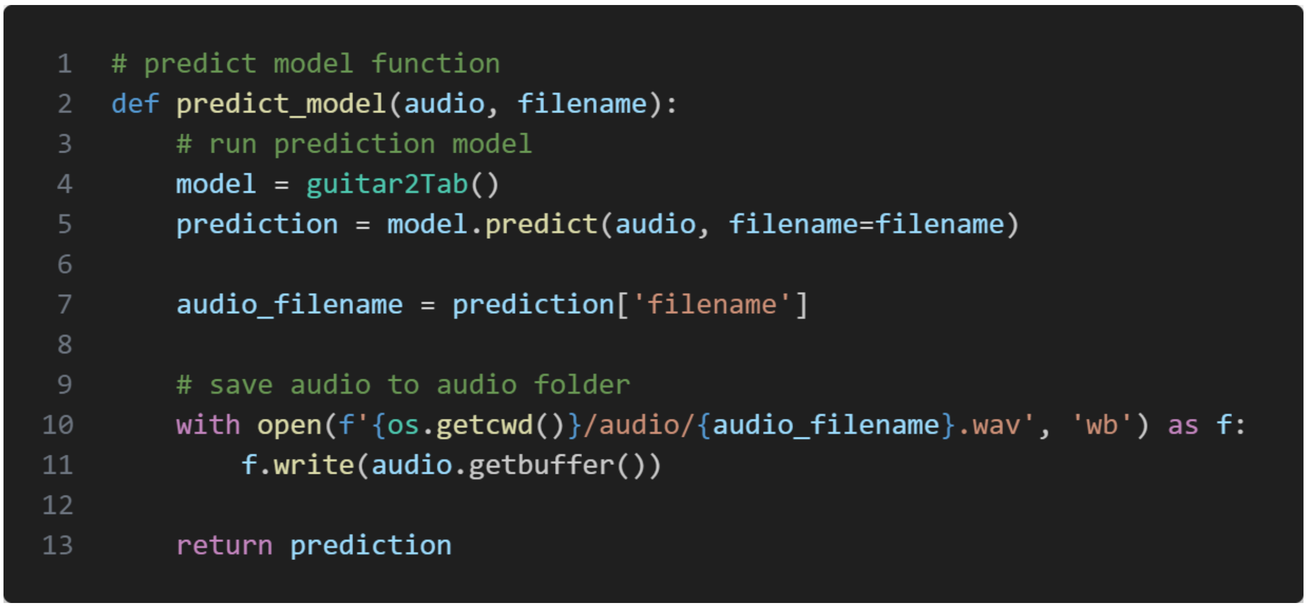
As illustrated in Figure 76, the snippet of code shown the prediction route that takes in a file request. The file extension permitted only audio file as an input, otherwise an error would be returned with the message showing incorrect file extension, or no file supplied in the multipart form of the client. If the file extension was correct, the audio will be read and converted to binary data and passed to the DL model.



```python
# predict model
@app.route('/predict', methods=['POST'])
@cross_origin()
def upload_file():
    # only allow post requests
    if request.method == 'POST':

        # check if the post request has the file part called 'audio'
        if 'audio' not in request.files:
            return jsonify({"status": 400, "message": "No file part"}), 400

        file_audio = request.files.get('audio')

        # check if the file extension is allowed
        if(file_audio and not allowed_file(file_audio.filename)):
            return jsonify({"status": 401, "message": "Incorrect file extension. Allowed file extensions are .wav, .mp3, and .ogg"}), 401

        # read audio file and convert to binary
        audio = io.BytesIO(file_audio.read())

        filename_without_extension = secure_filename(os.path.splitext(file_audio.filename)[0])

        # COMPLETED: AUTO INCREMENT PRIMARY ID - MODIFIED THE DATABASE TABLE

        # prediction model
        prediction = predict_model(audio, filename=filename_without_extension)

        # upload xml and audio file to azure blob storage
        blob_urls = upload_file_to_blob_storage_and_delete_files(filename_without_extension)

        # return 201 response
        return jsonify({
            "status": 201,
            "filename": prediction['filename'],
            "original_audio_url": blob_urls.get("audio_url"),
            "url": blob_urls.get("xml_url")
        }), 201
```

Figure 76 shows the 'predict' route that takes in an audio input, passed it to DL model and return the prediction as a JSON file.

The function shown in Figure 77 took the audio and perform an audio analysis which extract key features such as the tempo, beat tracking, key signature, quantisation, and onset times. These findings would be appended as metadata of the MusicXML file and returned the stored location of the file on the local directory.



```python
# predict model function
def predict_model(audio, filename):
    # run prediction model
    model = guitar2Tab()
    prediction = model.predict(audio, filename=filename)

    audio_filename = prediction['filename']

    # save audio to audio folder
    with open(f'{os.getcwd()}/audio/{audio_filename}.wav', 'wb') as f:
        f.write(audio.getbuffer())

    return prediction
```

Figure 77 illustrated the function that sent back the key features of the audio.

5.8 Sprint 4

5.8.1 Goal

The goal for this Sprint was to put in place a Front-end application to establish some connection between the Front-end and Back-end applications. This Sprint was also utilised to test drive the AlphaTab library and its capabilities.

5.8.2 Front-end application

As mentioned earlier, the React app would be hosted with the contribution of AlphaTab to house the musicXML file to display the music sheet. The library was the core library that upheld the entire Frontend and would vastly contribute to the interactive music sheet the author had envisioned.

5.8.3 Audio Player Component

The audio player, illustrated in Figure 78 was implemented to enhance the UX and UI, but also to store the metadata of the MIDI audio and the actual audio. This was done to keep track of the timestamps, states, and the volumes of both audio players.

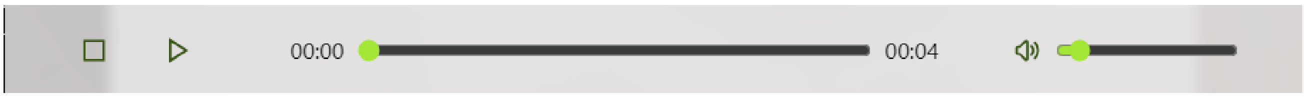
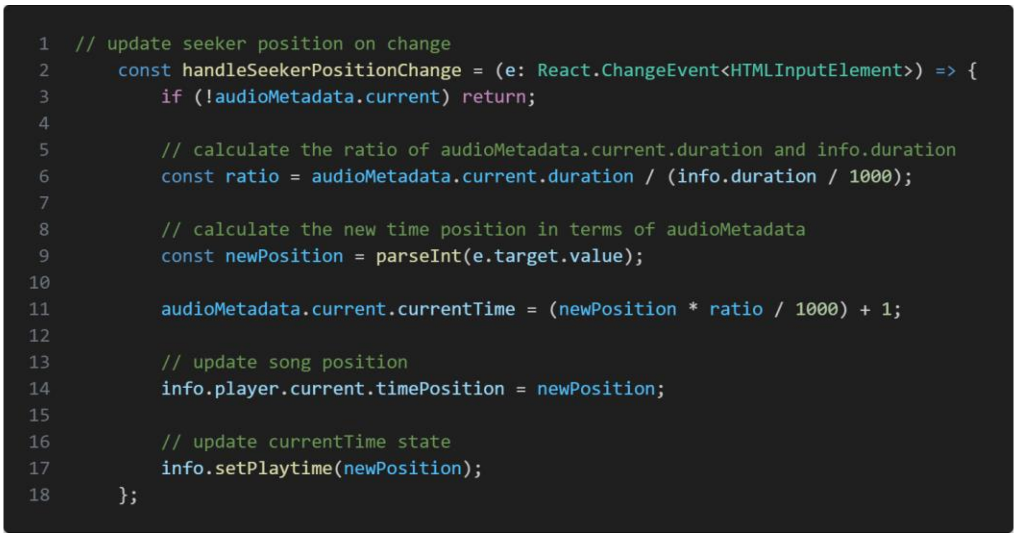


Figure 78 shows the audio player component with the ability to play/pause, seek, and change volume.

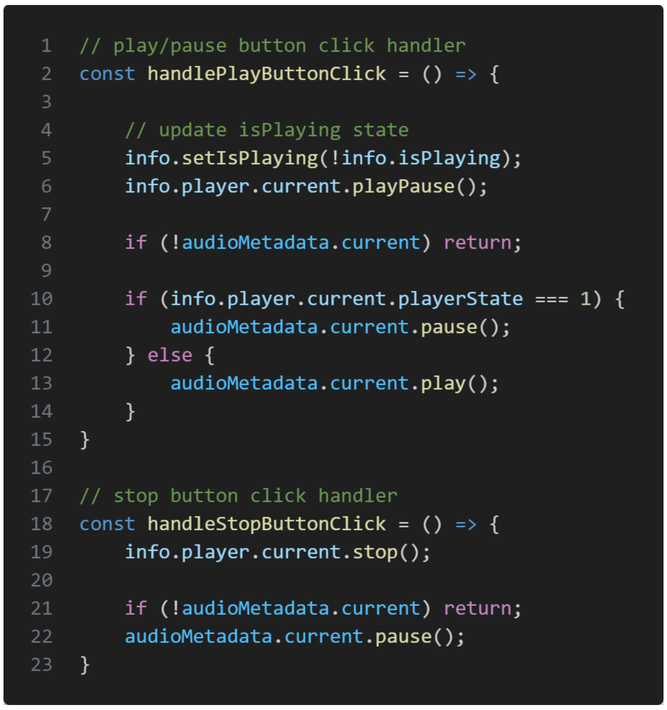
Since the user would be allowed to switch between audio outputs, these stored metadata were important to enforce the synchronization API, which aimed to update the current time of each player. Due to the missing notes that occurred in the backend API, this needed to be address by calculating the ratio between the two durations and simply multiply the duration of the actual audio to the ratio to get time difference that the MIDI audio should seek to as illustrated in Figure 79.



```
// update seeker position on change
    const handleSeekerPositionChange = (e: React.ChangeEvent<HTMLInputElement>) => {
        if (!audioMetadata.current) return;

        // calculate the ratio of audioMetadata.current.duration and info.duration
        const ratio = audioMetadata.current.duration / (info.duration / 1000);

        // calculate the new time position in terms of audioMetadata
        const newPosition = parseInt(e.target.value);

        audioMetadata.current.currentTime = (newPosition * ratio / 1000) + 1;

        // update song position
        info.player.current.timePosition = newPosition;

        // update currentTime state
        info.setPlaytime(newPosition);
    };
```

Figure 79 illustrated the function that handle seeking through the audio.

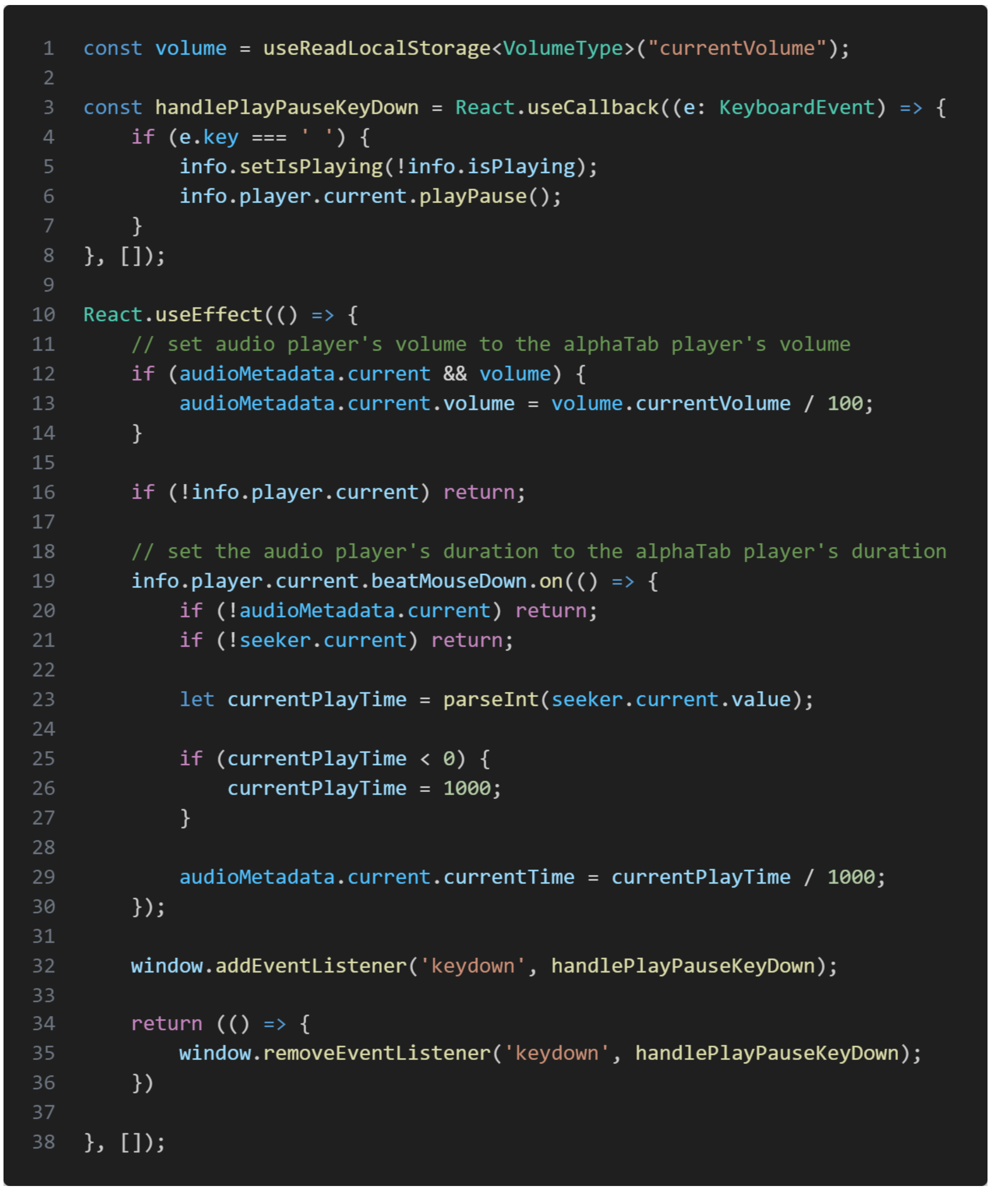
Other standard features were also implemented to bring about the quality of life to the user such as the play/pause and stop functionalities. Additionally, the icons were added to create seamless interface to change between icons as shown in Figure 80 i.e. the icon of playing and pausing would be changed to the respecting icons. Once the icon button was clicked by the user the audio player would either play/pause.



```
// play/pause button click handler
const handlePlayButtonClick = () => {

    // update isPlaying state
    info.setIsPlaying(!info.isPlaying);
    info.player.current.playPause();

    if (!audioMetadata.current) return;

    if (info.player.current.playerState === 1) {
        audioMetadata.current.pause();
    } else {
        audioMetadata.current.play();
    }
}

// stop button click handler
const handleStopButtonClick = () => {
    info.player.current.stop();

    if (!audioMetadata.current) return;
    audioMetadata.current.pause();
}
```

Figure 80 illustrated the functionality that switch icons to play/pause the icon.

Shortcuts were implemented to allow quick interactions with the audio player without having to move the mouse. This was done using Window’s Event Listener, where an event would be specified as the first argument to capture, and a callback function could be passed as the second argument to execute once an event occurred. For example, a ‘keydown’ event could captured and by passing the handlePlayPauseKeyDown callback, the space key could be utilized as a play and pause shortcut that the user could exploit. Furthermore, another event listener was also utilized to update the player’s duration to the same as the duration of AlphaTab’s player once the audio was played as shown in Figure 81.

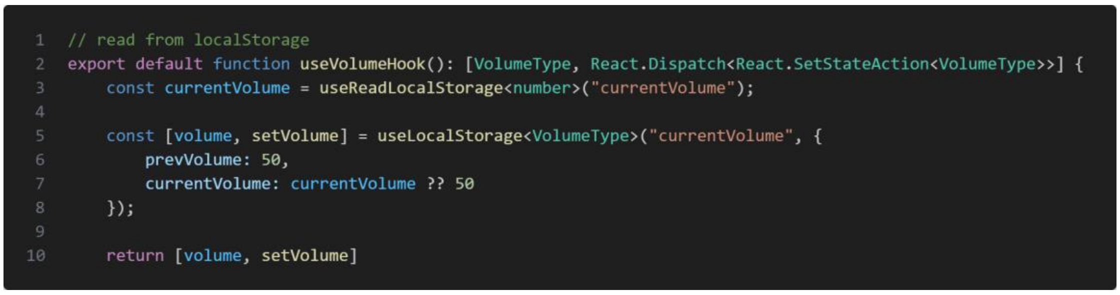


```
const volume = useReadLocalStorage<VolumeType>("currentVolume");

const handlePlayPauseKeyDown = React.useCallback((e: KeyboardEvent) => {
    if (e.key === ' ') {
        info.setIsPlaying(!info.isPlaying);
        info.player.current.playPause();
    }
}, []);

React.useEffect(() => {
    // set audio player's volume to the alphaTab player's volume
    if (audioMetadata.current && volume) {
        audioMetadata.current.volume = volume.currentVolume / 100;
    }

    if (!info.player.current) return;

    // set the audio player's duration to the alphaTab player's duration
    info.player.current.beatMouseDown.on(() => {
        if (!audioMetadata.current) return;
        if (!seeker.current) return;

        let currentPlayTime = parseInt(seeker.current.value);

        if (currentPlayTime < 0) {
            currentPlayTime = 1000;
        }

        audioMetadata.current.currentTime = currentPlayTime / 1000;
    });

    window.addEventListener('keydown', handlePlayPauseKeyDown);

    return (() => {
        window.removeEventListener('keydown', handlePlayPauseKeyDown);
    })

}, []);
```

Figure 81 illustrated the event listener to handle keyboard event.

Once a user navigated to another page, where a component would be unmounted, the event listener would also need to be removed to free memory. This was done using the useEffect hook, and by using the return function to tell React what to do once the component was unmounted from the page – in this case, to remove event listener.

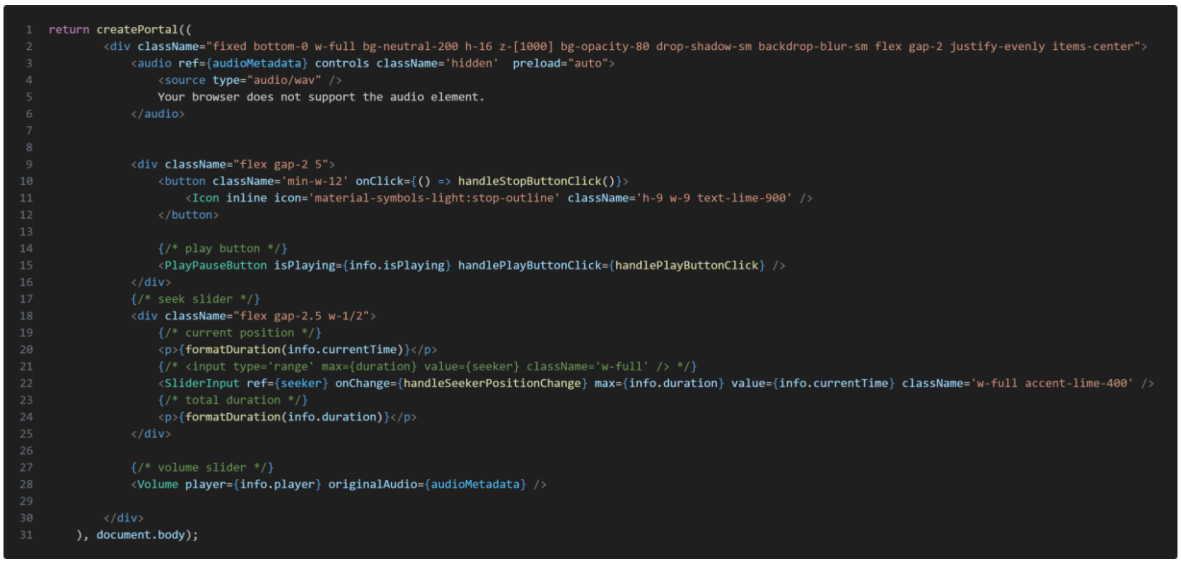
Audio volume synchronization – the value of the audio would be checked against the value in the local storage. To fetch to the local storage, there were a couple of custom hooks that were executed in the background i.e. the useVolumeHook – which utilized the useReadLocalStorage to check for existing key called ‘currentVolume’ in the local storage. If the key existed, it would use the value inside the key as the actual volume and store it using useLocalStorage hook as a getter and setter of the audio volume, where it would then get returned to the audio player component. Otherwise, a default value (50) would be assigned to the currentVolume, in the case that it’s the user’s first time using the site. Additionally, there was an especially simple yet challenging complication that was addressed in this Sprint. This was the ability to toggle between mute/unmute.



```
// read from localStorage
export default function useVolumeHook(): [VolumeType, React.Dispatch<React.SetStateAction<VolumeType>>] {
    const currentVolume = useReadLocalStorage<number>("currentVolume");

    const [volume, setVolume] = useLocalStorage<VolumeType>("currentVolume", {
        prevVolume: 50,
        currentVolume: currentVolume ?? 50
    });

    return [volume, setVolume]
```

Figure 82 illustrated the useVolume hook that store the value of the volume.

The component itself would be mounted as a Portal onto the document body, where it would be the most practical since the placement of the player needed to be fixated at the bottom to allowed for accessibility and to provide the best possible UX to the users as illustrated in Figure 83.



```
return createPortal((
        <div className="fixed bottom-0 w-full bg-neutral-200 h-16 z-[1000] bg-opacity-80 drop-shadow-sm backdrop-blur-sm flex gap-2 justify-evenly items-center">
            <audio ref={audioMetadata} controls className='hidden'  preload="auto">
                <source type="audio/wav" />
                Your browser does not support the audio element.
            </audio>


            <div className="flex gap-2 5">
                <button className='min-w-12' onClick={() => handleStopButtonClick()}>
                    <Icon inline icon='material-symbols-light:stop-outline' className='h-9 w-9 text-lime-900' />
                </button>

                {/* play button */}
                <PlayPauseButton isPlaying={info.isPlaying} handlePlayButtonClick={handlePlayButtonClick} />
            </div>
            {/* seek slider */}
            <div className="flex gap-2.5 w-1/2">
                {/* current position */}
                <p>{formatDuration(info.currentTime)}</p>
                {/* <input type='range' max={duration} value={seeker} className='w-full' /> */}
                <SliderInput ref={seeker} onChange={handleSeekerPositionChange} max={info.duration} value={info.currentTime} className='w-full accent-lime-400' />
                {/* total duration */}
                <p>{formatDuration(info.duration)}</p>
            </div>

            {/* volume slider */}
            <Volume player={info.player} originalAudio={audioMetadata} />

        </div>
    ), document.body);
```

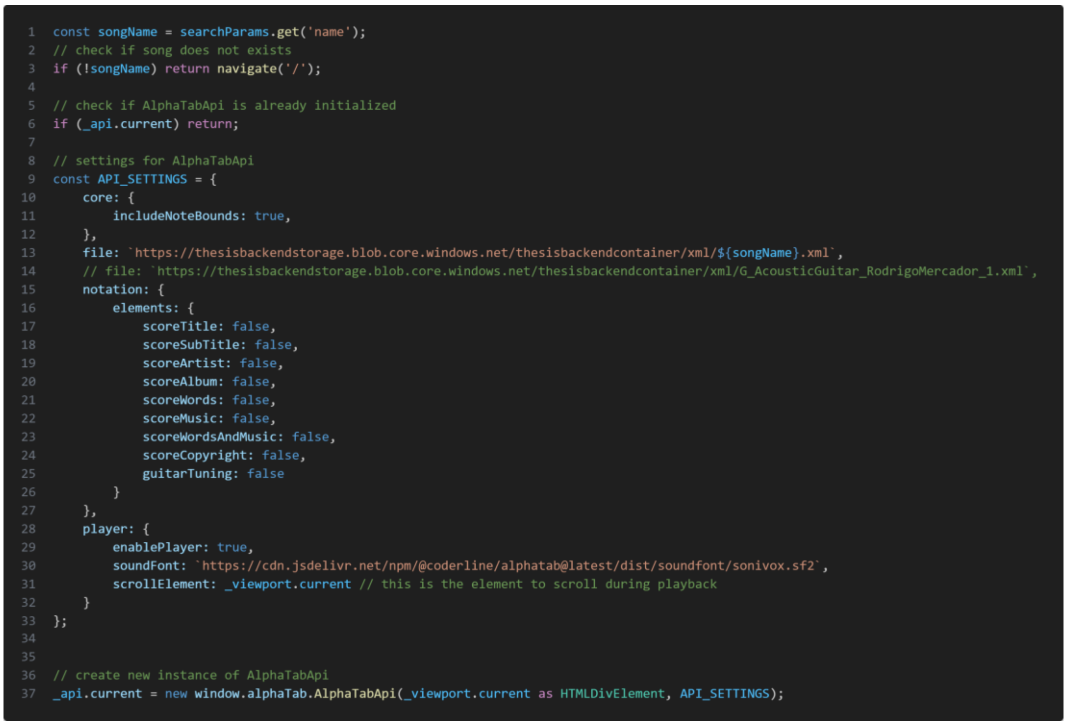
Figure 83 illustrated the createPortal API that mount the component outside of the body document.

5.8.4 Tab Visualiser

Once the audio had been successfully predicted, the user would be navigated to the interactive music sheet, where they would have the ability to point and click to any section of the song. By utilising a JavaScript library called ‘AlphaTab’, the author was able to implement such feature with the API that provided ways to track the current beat, note, and bar in a music sheet.

In order to set up AlphaTab for utilisation, there was some complication that was worth mentioning, where the library would struggle to be integrate into a React application. After extensive research, the author had discovered that Create-React-App and Vite bundlers would not work with AlphaTab’s worker, both of which the author had tried to use with no avails. According to Fheyen (2023) and Matevsenlab (2023), whom discussed about these very issues with the developer of AlphaTab and found that either bundlers were supported by the library at the moment due to ‘rather terrible support of Web Workers and Audio Worklets’, of which the library relied heavily on. The developer of the library suggested that one should customise the Webpack configuration to change how Webpack bundled dependencies. Albeit the attempt, the author was not managed to solve the issue. Thus, the final resource was to utilise AlphaTab’s Content Delivery Network (CDN) to import the dependency or install the library as static files in the public path of the application. Though using this method, the developer experience could be less enjoyable since there won’t be autocompletes, but there was no other choice until the developer decided to increase compatibility to web bundlers.

To initalise the library, a setting must be first provided into the AlphaTab API to enable a couple of features such as IncludeNoteBounds for note editability, hide elements in the music sheet, load in MusicXML file, and MIDI player as shown in Figure 84.



```
const songName = searchParams.get('name');
// check if song does not exists
if (!songName) return navigate('/');

// check if AlphaTabApi is already initialized
if (_api.current) return;

// settings for AlphaTabApi
const API_SETTINGS = {
    core: {
        includeNoteBounds: true,
    },
    file: `https://thesisbackendstorage.blob.core.windows.net/thesisbackendcontainer/xml/${songName}.xml`,
    // file: `https://thesisbackendstorage.blob.core.windows.net/thesisbackendcontainer/xml/G_AcousticGuitar_RodrigoMercador_1.xml`,
    notation: {
        elements: {
            scoreTitle: false,
            scoreSubTitle: false,
            scoreArtist: false,
            scoreAlbum: false,
            scoreWords: false,
            scoreMusic: false,
            scoreWordsAndMusic: false,
            scoreCopyright: false,
            guitarTuning: false
        }
    },
    player: {
        enablePlayer: true,
        soundFont: `https://cdn.jsdelivr.net/npm/@coderline/alphatab@latest/dist/soundfont/sonivox.sf2`,
        scrollElement: _viewport.current // this is the element to scroll during playback
    }
};


// create new instance of AlphaTabApi
_api.current = new window.alphaTab.AlphaTabApi(_viewport.current as HTMLDivElement, API_SETTINGS);
```

Figure 84 illustrated the initialisation of AlphaTab.

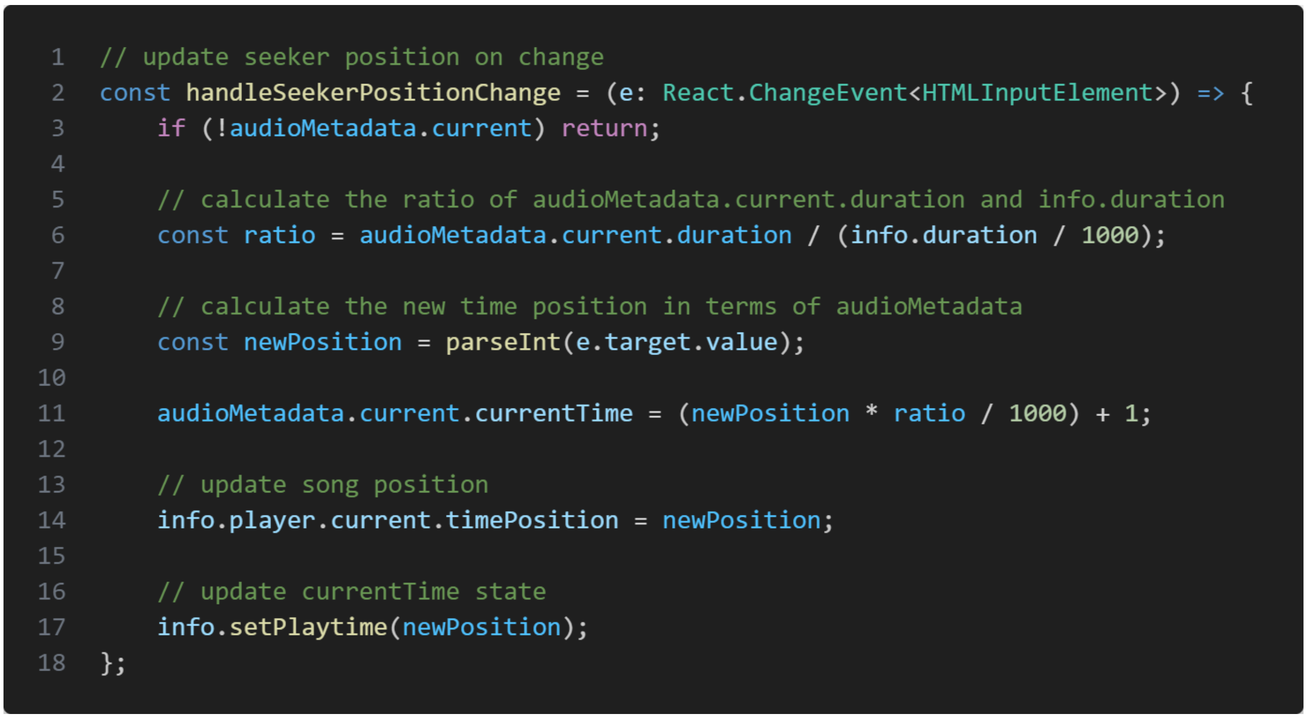
Since the AlphaTab was utilised in React, the best practice to initialise the API would be in the useEffect to prevent the API from re-rendering once the component was updated or mounted on the page. The API would also take in event listeners to execute codes based on the conditions such as:

1. Toggle loading screen – the feature utilized the renderStarted and renderFinished event listeners to toggle the display property in CSS of the element to show/hide the loading screen.
2. Set song details – this utilized the scoreLoaded event listener to store the state of the loaded music sheet to display the song title in the header, as well as other information such as tempo, tuning, time signature, key signature, and artist name in the sub header.
3. Set the volume of the MIDI player – if the user had set the volume to another value prior, the cached volume would be fetched from the local storage to enhance the UX.

Once the music score had been loaded the interactive sheet would be displayed to the user, where they would be able to seek to the section would like to listen to and either choose to edit the sheet if they are logged in.

5.8.5 Audio Syncing

At the moment, the MusicXML file produced by the server was majorly hard-coded, where each note detected would be mapped one after the other, in the addition of rest notes to place pauses in between frames that no sound was played. Due to this, the actual length of the audio and that produced from the server would need to be offset in order to align the actual audio to the MIDI audio as illustrated in Figure 85.



```
// update seeker position on change
const handleSeekerPositionChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    if (!audioMetadata.current) return;

    // calculate the ratio of audioMetadata.current.duration and info.duration
    const ratio = audioMetadata.current.duration / (info.duration / 1000);

    // calculate the new time position in terms of audioMetadata
    const newPosition = parseInt(e.target.value);

    audioMetadata.current.currentTime = (newPosition * ratio / 1000) + 1;

    // update song position
    info.player.current.timePosition = newPosition;

    // update currentTime state
    info.setPlaytime(newPosition);
};
```

Figure 85 illustrated the seeking functionality of the audio player.

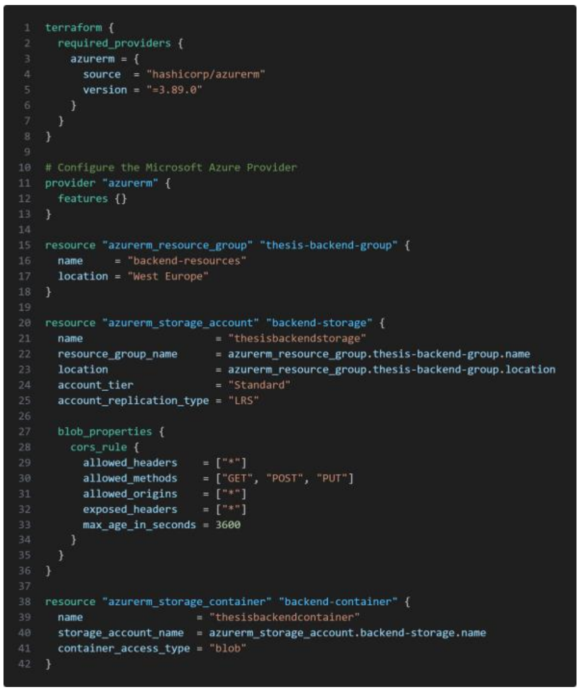
The offset happened when the position of the seeker was changed, it could be simply offset by calculating the ratio between the duration of the actual audio and that from the MIDI audio. The ratio would be multiplied by the duration of the MIDI audio, where the new value would be used to seek the actual audio to.

5.8.6 Back-end Deployment

The Back-end Development went through different iterations of providers due to the difficulties faced during each provider. To summarise, AWS did have a lot of utilities, albeit it provided the closest region that provided near zero latency to browse and fetch the Front-end/Back-end. Thus, Azure was chosen where free credits were also provided as part of the GitHub educational package and allowed the flexibility with experimentation with different services. These services were:

File Storage

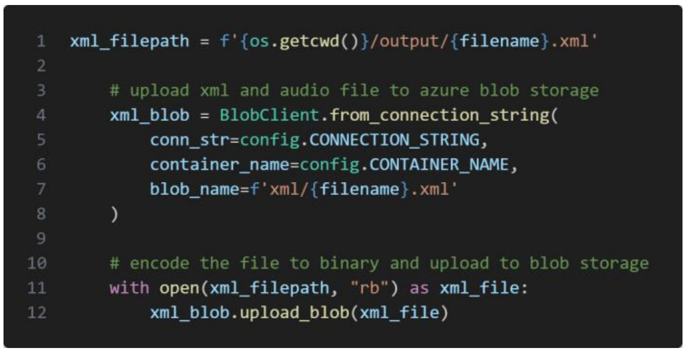
Blob Storage was the service that allowed audio, MusicXML, and assets to be stored and accessed by the Front-end/Back-end. The set up was rather simple and share a similarity with how MongoDB connect to its database. Once the storage had been created on the Terraform, an Infrastructure as a Code (IaaC) tool that provided a declarative configuration to defined infrastructures (Azure). As shown in Figure 86, the service could be easily set up in a code language with some configuration to allow access from cross-origins. This meant that local machines that ran the Front-end for development would be able to access these assets as well the Back-end.



```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "=3.89.0"
    }
  }
}

# Configure the Microsoft Azure Provider
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "thesis-backend-group" {
  name     = "backend-resources"
  location = "West Europe"
}

resource "azurerm_storage_account" "backend-storage" {
  name                     = "thesisbackendstorage"
  resource_group_name      = azurerm_resource_group.thesis-backend-group.name
  location                 = azurerm_resource_group.thesis-backend-group.location
  account_tier             = "Standard"
  account_replication_type = "LRS"

  blob_properties {
    cors_rule {
      allowed_headers    = ["*"]
      allowed_methods    = ["GET", "POST", "PUT"]
      allowed_origins    = ["*"]
      exposed_headers    = ["*"]
      max_age_in_seconds = 3600
    }
  }
}

resource "azurerm_storage_container" "backend-container" {
  name                  = "thesisbackendcontainer"
  storage_account_name  = azurerm_storage_account.backend-storage.name
  container_access_type = "blob"
}
```

Figure 86 illustrated code defined to create a Blob Storage on Azure.

The Back end handled the prediction and outputs the MusicXML, and audio assets to the storage service. This was done by connecting the Blob storage via Azure-blob-storage package on Python, where the connection string could be provided to the BlobClient alongside the name of the container and the name as well as the path of the Blob that would store the assets (shown in Figure 87). These codes were repeated for the audio asset to also upload it to the storage.

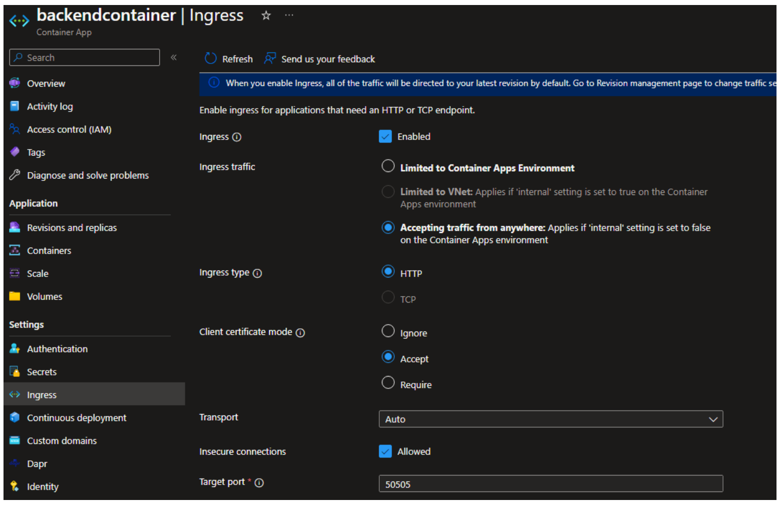


```python
xml_filepath = f'{os.getcwd()}/output/{filename}.xml'

    # upload xml and audio file to azure blob storage
    xml_blob = BlobClient.from_connection_string(
        conn_str=config.CONNECTION_STRING,
        container_name=config.CONTAINER_NAME,
        blob_name=f'xml/{filename}.xml'
    )

    # encode the file to binary and upload to blob storage
    with open(xml_filepath, "rb") as xml_file:
        xml_blob.upload_blob(xml_file)
```

Figure 87 illustrated how the BlobClient works in VS Code.

Container

Azure Container App and Container Registry were used to store and host the docker image on the service. This eliminated the time needed to setup and configure the system to match the local version of the running application. The Back-end application could be simply compiled as a Docker Image and pushed to the Container Registry on the cloud, where the Image would be pulled into the Container App for continuous deployment. Though, there was a technical difficulty once the application was deployed where the hosted site would not re-route to serve the API. After some investigation as illustrated in Figure 88, the author had discovered the port specified on the Ingress and that in the Docker Image was misconfigured. Once the port was aligned, the application was able to function as intended.



When you enable Ingress, all of the traffic will be directed to your latest revision by default. Go to Revision management page to change traffic se

Enable ingress for applications that need an HTTP or TCP endpoint.

Ingress: ✓ Enabled

Ingress traffic: Limited to Container Apps Environment / Limited to VNet: Applies if 'internal' setting is set to true on the Container Apps environment / [✓ Accepting traffic from anywhere: Applies if 'internal' setting is set to false on the Container Apps environment]

Ingress type: [✓ HTTP] / TCP

Client certificate mode: Ignore / [✓ Accept] / Require

Transport: Auto

Insecure connections: ✓ Allowed

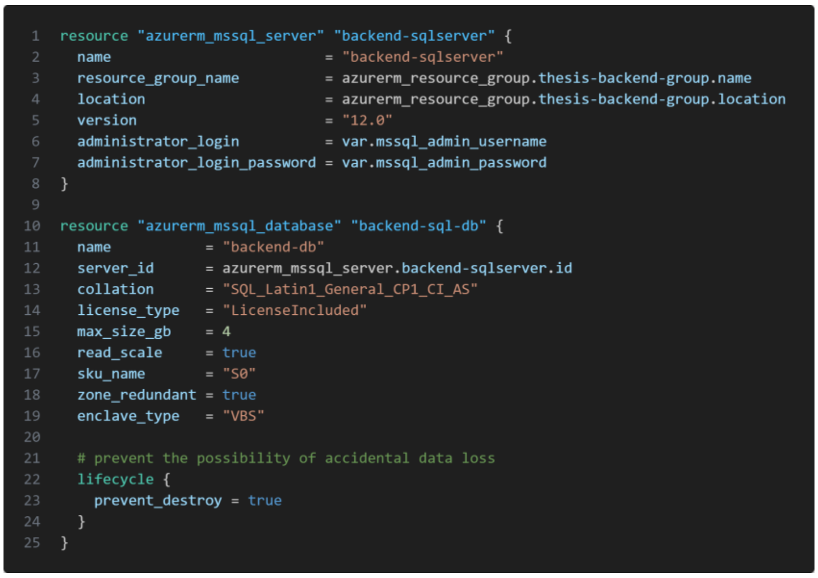
Target port *: 50505

Figure 88 illustrated the Ingress option on Azure Container App.

Database

Azure SQL Database was also utilised to store and retrieve user and song information from a database that was in the same ecosystem as the cloud provider, as it would reduce latency in communicating between different services. The service was also initiated by

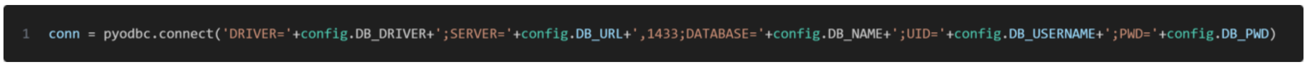
Terraform to be easily configured and removed if the plan was changed due to the fact that, in order to integrate the service into the backend API, a driver called ‘ODBC Driver for SQL Server’ needed to be installed to the backend OS and a Python package called pyodbc was also installed onto the backend to provide access to the database (DB) as illustrated in Figure 89.



```
resource "azurerm_mssql_server" "backend-sqlserver" {
  name                         = "backend-sqlserver"
  resource_group_name          = azurerm_resource_group.thesis-backend-group.name
  location                     = azurerm_resource_group.thesis-backend-group.location
  version                      = "12.0"
  administrator_login          = var.mssql_admin_username
  administrator_login_password = var.mssql_admin_password
}

resource "azurerm_mssql_database" "backend-sql-db" {
  name           = "backend-db"
  server_id      = azurerm_mssql_server.backend-sqlserver.id
  collation      = "SQL_Latin1_General_CP1_CI_AS"
  license_type   = "LicenseIncluded"
  max_size_gb    = 4
  read_scale     = true
  sku_name       = "S0"
  zone_redundant = true
  enclave_type   = "VBS"

  # prevent the possibility of accidental data loss
  lifecycle {
    prevent_destroy = true
  }
}
```

Figure 89 illustrated the initialisation of Terraform.

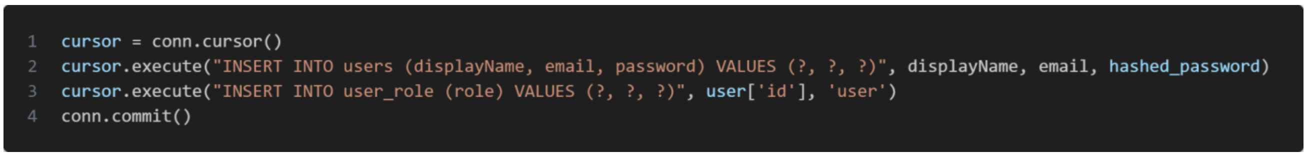
As shown in Figure 90, the functionality worked very similarly to Azure Blob Storage, where a connection string was provided in the configuration page and the DB could be accessible by the API.



```
conn = pyodbc.connect('DRIVER='+config.DB_DRIVER+';SERVER='+config.DB_URL+',1433;DATABASE='+config.DB_NAME+';UID='+config.DB_USERNAME+';PWD='+config.DB_PWD)
```

Figure 90 illustrated how the backend API connects to the DB.

The Figure shown in 91 illustrated how the application utilised Pyodbc to store data onto the DB. Due to the prior experience with SQL in previous college and professional works, majority syntax and structures were easily implemented into the API, despite the differences in the language barrier of Python.

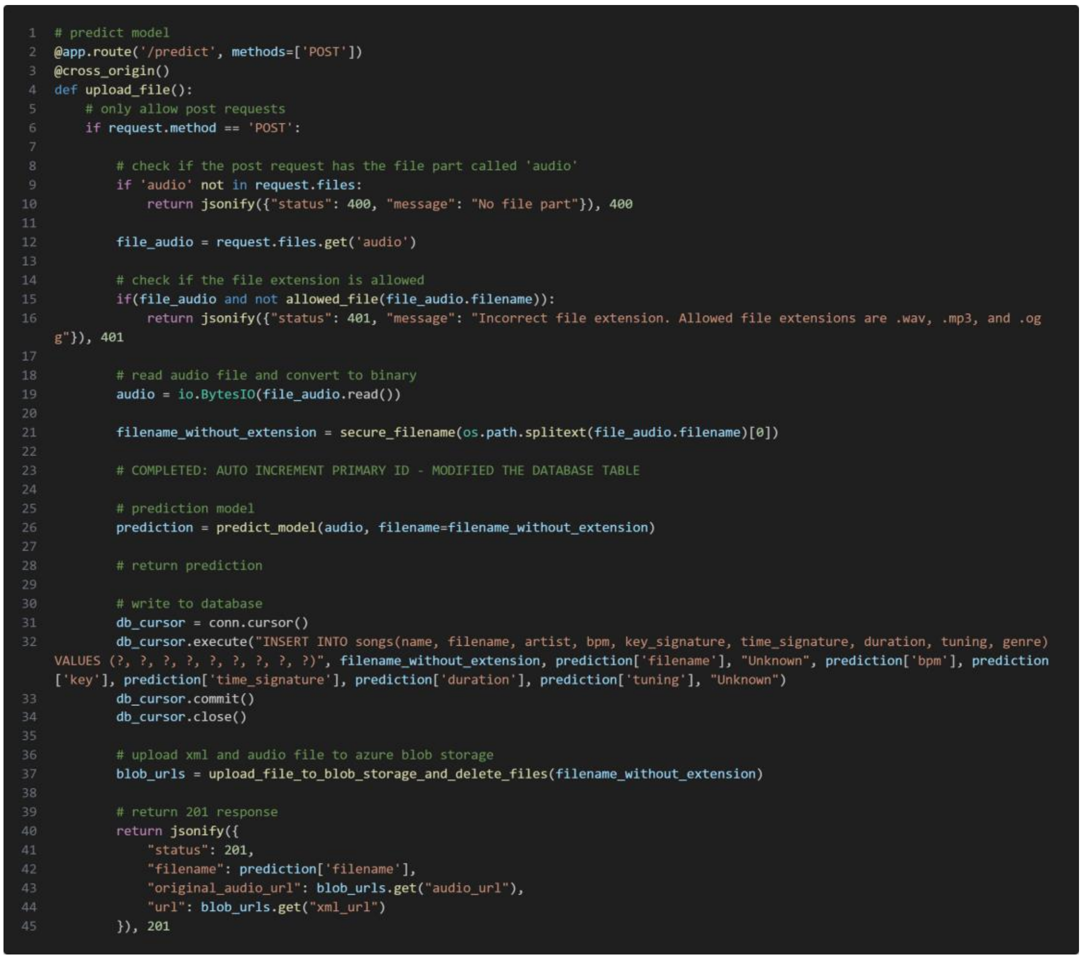


```
cursor = conn.cursor()
cursor.execute("INSERT INTO users (displayName, email, password) VALUES (?, ?, ?)", displayName, email, hashed_password)
cursor.execute("INSERT INTO user_role (role) VALUES (?, ?, ?)", user['id'], 'user')
conn.commit()
```

Figure 91 illustrated the syntax used to fetch to the DB for insert new user.

Back-end Implementation

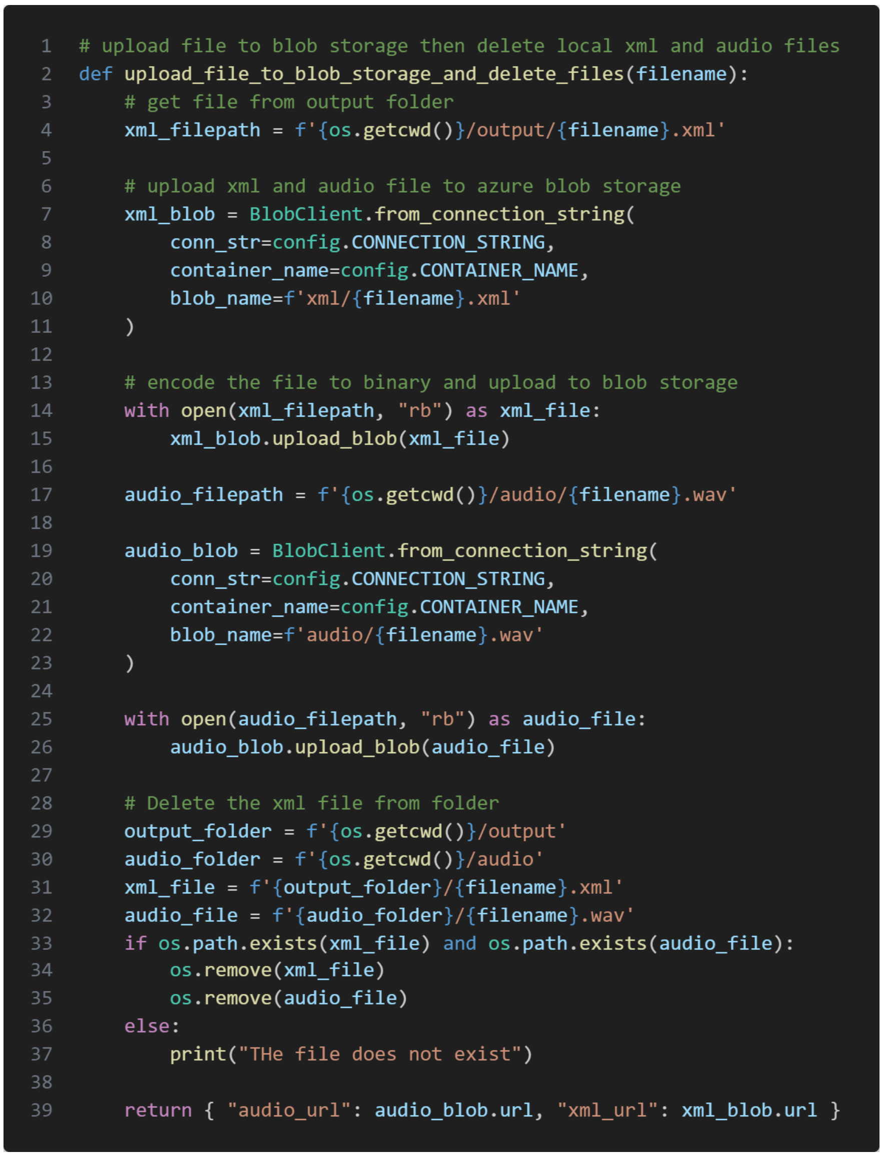
While the Front end handled the user’s request, the Back end needed to handle the processing of the audio. The route called ‘predict’ was defined to only take in a POST request and a file request named ‘audio’. Otherwise, a 400 HTTP response would be returned to specify that no file existed on the request header. The validation to detect file extensions was also implemented in the Back end to gatekeep incorrect Mime types i.e. image file type such as PNG, JPEG, or GIF as shown in Figure 92.



```python
# predict model
@app.route('/predict', methods=['POST'])
@cross_origin()
def upload_file():
    # only allow post requests
    if request.method == 'POST':

        # check if the post request has the file part called 'audio'
        if 'audio' not in request.files:
            return jsonify({"status": 400, "message": "No file part"}), 400

        file_audio = request.files.get('audio')

        # check if the file extension is allowed
        if(file_audio and not allowed_file(file_audio.filename)):
            return jsonify({"status": 401, "message": "Incorrect file extension. Allowed file extensions are .wav, .mp3, and .ogg"}), 401

        # read audio file and convert to binary
        audio = io.BytesIO(file_audio.read())

        filename_without_extension = secure_filename(os.path.splitext(file_audio.filename)[0])

        # COMPLETED: AUTO INCREMENT PRIMARY ID - MODIFIED THE DATABASE TABLE

        # prediction model
        prediction = predict_model(audio, filename=filename_without_extension)

        # return prediction

        # write to database
        db_cursor = conn.cursor()
        db_cursor.execute("INSERT INTO songs(name, filename, artist, bpm, key_signature, time_signature, duration, tuning, genre) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)", filename_without_extension, prediction['filename'], "Unknown", prediction['bpm'], prediction['key'], prediction['time_signature'], prediction['duration'], prediction['tuning'], "Unknown")
        db_cursor.commit()
        db_cursor.close()

        # upload xml and audio file to azure blob storage
        blob_urls = upload_file_to_blob_storage_and_delete_files(filename_without_extension)

        # return 201 response
        return jsonify({
            "status": 201,
            "filename": prediction['filename'],
            "original_audio_url": blob_urls.get("audio_url"),
            "url": blob_urls.get("xml_url")
        }), 201
```

Figure 92 illustrated the predict route that process the audio file and output XML file.

Once all validations were met, the audio file needed to be read to memory, which converted binary data into streams of bytes. This was done using the IO module in Python to provide a file-like object interface for streams of bytes. The streams would be fed into the predict_model function, save the MusicXML and audio file would be uploaded to Azure storage service called ‘Blob Storage’, and insert new column into the SQL database accordingly. Finally, a 201 HTTP response would be returned alongside the name of the file, the audio file, and the URL of MusicXML that the Front end could utilized to fetch to retrieve the files necessary to display music sheet as shown in Figure 93.

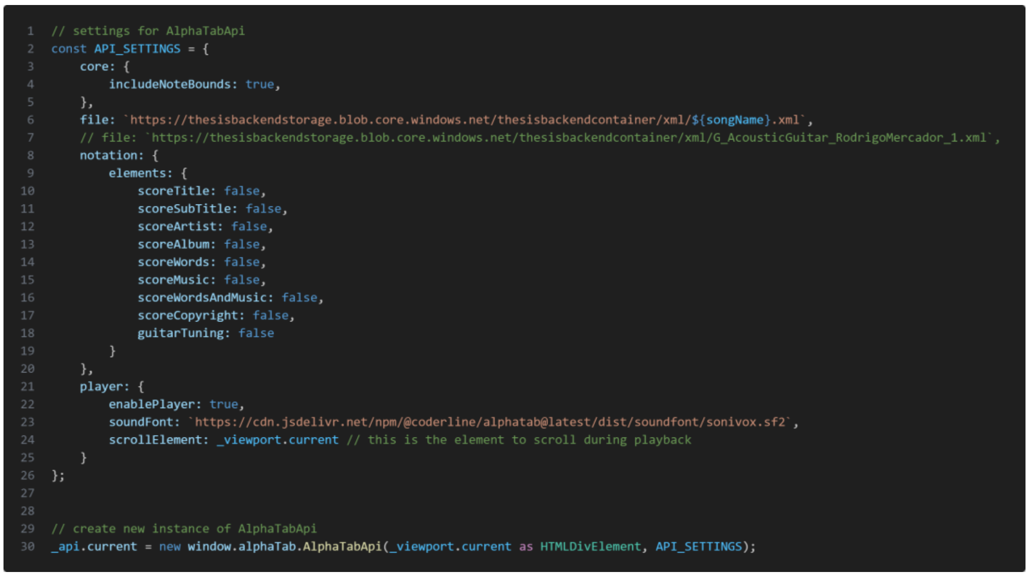


```python
# upload file to blob storage then delete local xml and audio files
def upload_file_to_blob_storage_and_delete_files(filename):
    # get file from output folder
    xml_filepath = f'{os.getcwd()}/output/{filename}.xml'

    # upload xml and audio file to azure blob storage
    xml_blob = BlobClient.from_connection_string(
        conn_str=config.CONNECTION_STRING,
        container_name=config.CONTAINER_NAME,
        blob_name=f'xml/{filename}.xml'
    )

    # encode the file to binary and upload to blob storage
    with open(xml_filepath, "rb") as xml_file:
        xml_blob.upload_blob(xml_file)

    audio_filepath = f'{os.getcwd()}/audio/{filename}.wav'

    audio_blob = BlobClient.from_connection_string(
        conn_str=config.CONNECTION_STRING,
        container_name=config.CONTAINER_NAME,
        blob_name=f'audio/{filename}.wav'
    )

    with open(audio_filepath, "rb") as audio_file:
        audio_blob.upload_blob(audio_file)

    # Delete the xml file from folder
    output_folder = f'{os.getcwd()}/output'
    audio_folder = f'{os.getcwd()}/audio'
    xml_file = f'{output_folder}/{filename}.xml'
    audio_file = f'{audio_folder}/{filename}.wav'
    if os.path.exists(xml_file) and os.path.exists(audio_file):
        os.remove(xml_file)
        os.remove(audio_file)
    else:
        print("THe file does not exist")

    return { "audio_url": audio_blob.url, "xml_url": xml_blob.url }
```

Figure 93 illustrated the function that handle file upload and store files on Azure Blob Storage.

5.8.7 Dynamic Route

There were a couple of methods to fetch the API endpoint from the URL path. Either through the use of slug or query, the name of the file could be retrieved from the Blob Storage on Azure. The author decided to utilise the query string methodology to fetch to the API, where the query string on the URL path would be retrieved from the useSearchParams hook from React Router DOM with the query called ‘name’, which could be accessed through a getter function to retrieve its value. This value would be checked for truthy, and if it did not exist, the user would be redirected back to the home page. Otherwise, its value would be fed into the configuration of AlphaTab to load in the score as shown in Figure 94.



```
// settings for AlphaTabApi
const API_SETTINGS = {
    core: {
        includeNoteBounds: true,
    },
    file: `https://thesisbackendstorage.blob.core.windows.net/thesisbackendcontainer/xml/${songName}.xml`,
    // file: `https://thesisbackendstorage.blob.core.windows.net/thesisbackendcontainer/xml/G_AcousticGuitar_RodrigoMercador_1.xml`,
    notation: {
        elements: {
            scoreTitle: false,
            scoreSubTitle: false,
            scoreArtist: false,
            scoreAlbum: false,
            scoreWords: false,
            scoreMusic: false,
            scoreWordsAndMusic: false,
            scoreCopyright: false,
            guitarTuning: false
        }
    },
    player: {
        enablePlayer: true,
        soundFont: `https://cdn.jsdelivr.net/npm/@coderline/alphatab@latest/dist/soundfont/sonivox.sf2`,
        scrollElement: _viewport.current // this is the element to scroll during playback
    }
};


// create new instance of AlphaTabApi
_api.current = new window.alphaTab.AlphaTabApi(_viewport.current as HTMLDivElement, API_SETTINGS);
```

Figure 94 illustrated the dynamic routing to Azure Blob Storage depending on the song name.

5.9 Sprint 5

5.9.1 Interim Presentation

The Sprint was significant to the development of the application due to the Interim presentation that was concurrently happening in the beginning of the week. It would be a great opportunity to present the overview of the project to the allocated supervisor and second reader, including recent updates on the progression of the development. The features of the Frontend worked as intended but the accuracy of model was mistakenly interpreted an extra note in a chord or even in a note with some missing note on a frame of the audio.

After the presentation, a couple of feedback were given help steer the author to the right direction. There was the main focus to highlight the need for a different model of prediction, through a different method of using the inputs. Furthermore, there were some recommendations to attempt to increase the model’s accuracy, and enhanced the model to detect common chord shapes to make the model recognised these shapes more commonly.

5.9.2 Goal

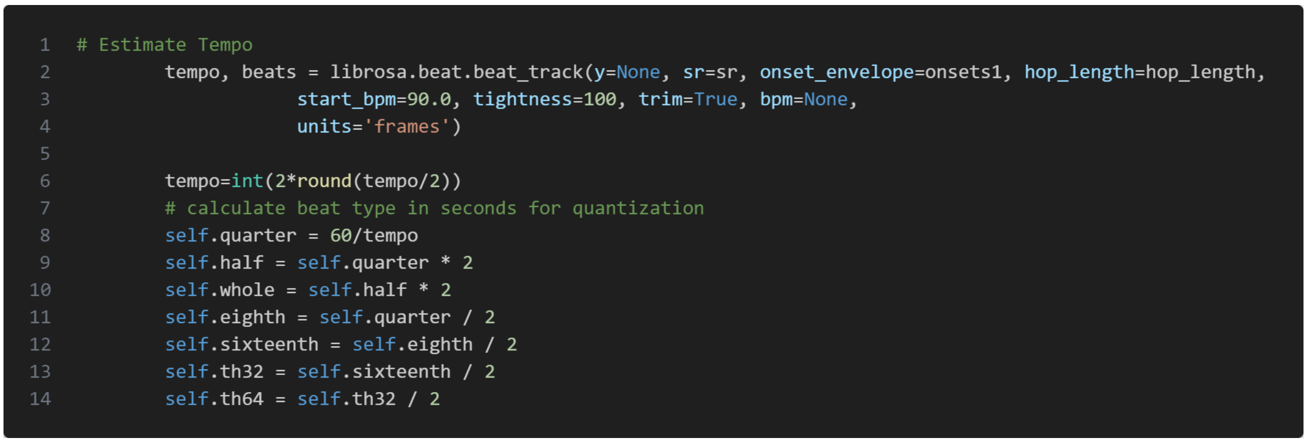
The goal was to try to attempt to increase the model accuracy and fixed the results to capture better timings for the onsets and offsets.

5.9.3 Quantization

Since the beat type was hard coded in Sprint 3 to tell the score how fast a particular note was being played, the supervisor recommended the author to look into the method of quantization, where the duration of a note could be computed into a beat type format.

This was done through the use of Librosa to calculate the estimated tempo of the song, where the tempo could be divided into each beat type through the knowledge that rhythmic values could be calculated to find its absolute time in milliseconds and allowed the duration of each note to be mapped onto a rhythmic value i.e. quarter notes, eighth notes, sixteenth notes and so on. And that, in the Beat Per Minute (BPM), a beat would be equivalent to a quarter note, and calculated using the formula: 60 (seconds) / Tempo (in BPM).

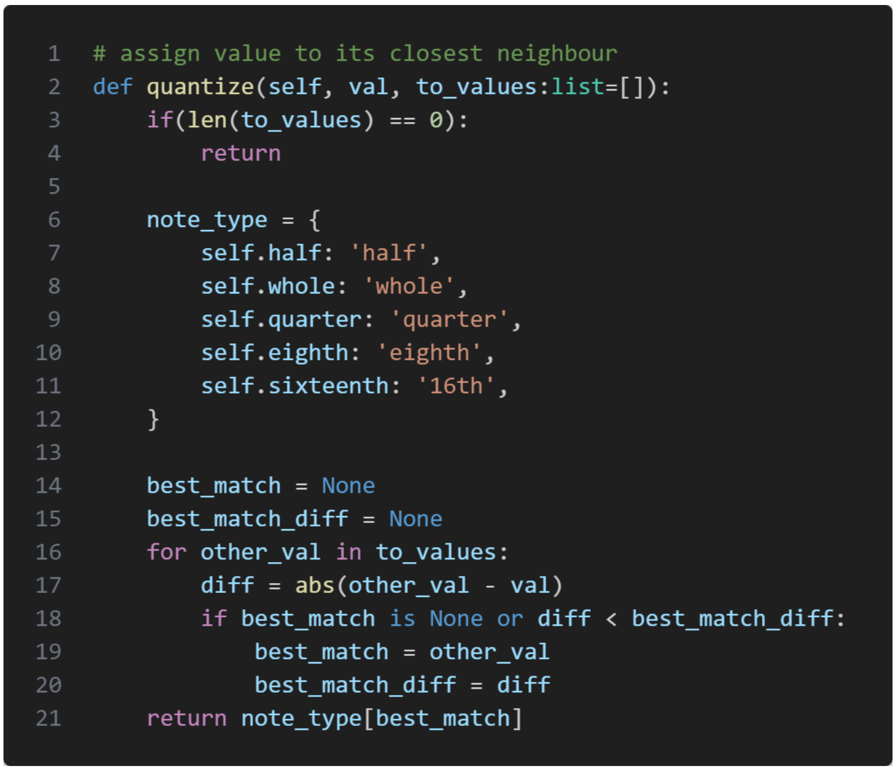
Consequently, each rhythmic value would be halves of what come before it e.g. an eighth note would be divided by the value of the quarter note by two, and so on as illustrated in Figure 95.



```python
# Estimate Tempo
        tempo, beats = librosa.beat.beat_track(y=None, sr=sr, onset_envelope=onsets1, hop_length=hop_length,
                    start_bpm=90.0, tightness=100, trim=True, bpm=None,
                    units='frames')

        tempo=int(2*round(tempo/2))
        # calculate beat type in seconds for quantization
        self.quarter = 60/tempo
        self.half = self.quarter * 2
        self.whole = self.half * 2
        self.eighth = self.quarter / 2
        self.sixteenth = self.eighth / 2
        self.th32 = self.sixteenth / 2
        self.th64 = self.th32 / 2
```

Figure 95 illustrated the tempo estimation of Librosa.

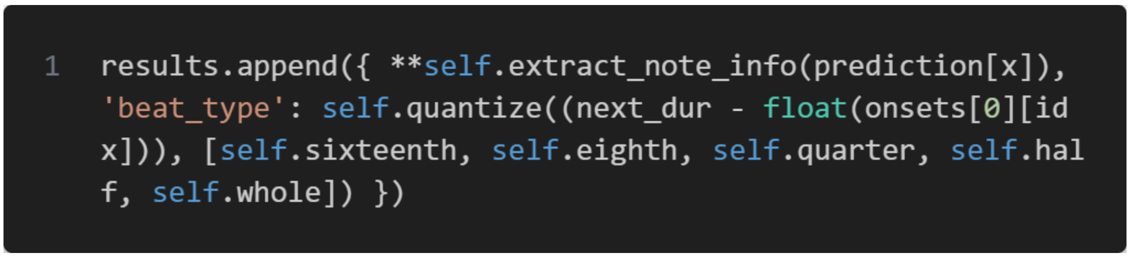
These rhythmic values would be stored, and utilised once the difference between onsets and offsets was calculated to get the duration of the note. This duration would then be passed into the quantize function to map the absolute time to its nearest value of the rhythmic value shown in Figure 96.



```python
# assign value to its closest neighbour
def quantize(self, val, to_values:list=[]):
    if(len(to_values) == 0):
        return

    note_type = {
        self.half: 'half',
        self.whole: 'whole',
        self.quarter: 'quarter',
        self.eighth: 'eighth',
        self.sixteenth: '16th',
    }

    best_match = None
    best_match_diff = None
    for other_val in to_values:
        diff = abs(other_val - val)
        if best_match is None or diff < best_match_diff:
            best_match = other_val
            best_match_diff = diff
    return note_type[best_match]
```

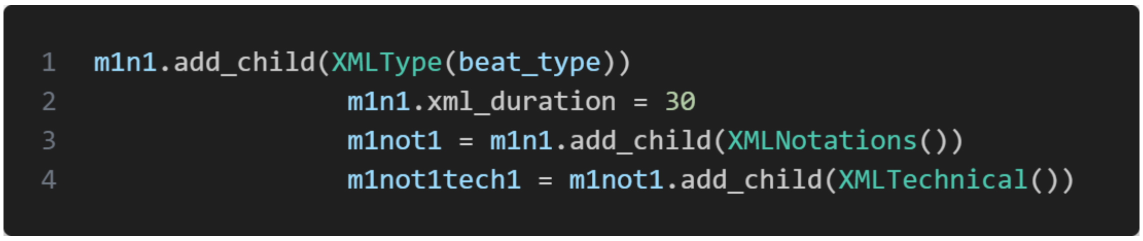
Figure 96 illustrated the quantization functionality.

Lastly, the beat type would be returned and added to the results array as ‘beat_type’ illustrated in Figure 97. Once the array was iterated through the add_note function, the beat_Type property would be utilised in the XMLType property shown in Figure 98.



```
results.append({ **self.extract_note_info(prediction[x]),
'beat_type': self.quantize((next_dur - float(onsets[0][id
x])), [self.sixteenth, self.eighth, self.quarter, self.hal
f, self.whole]) })
```

Figure 97 illustrated the computed quantization being inserted into results.

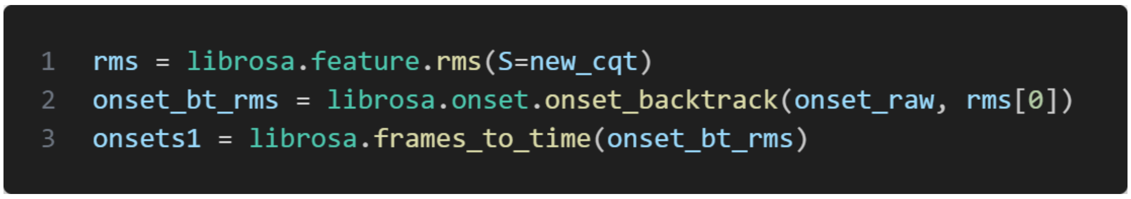


```
m1n1.add_child(XMLType(beat_type))
                m1n1.xml_duration = 30
                m1not1 = m1n1.add_child(XMLNotations())
                m1not1tech1 = m1not1.add_child(XMLTechnical())
```

Figure 98 illustrated the rhythmic value added into the note in the music score.

5.9.4 Improving Onset Detection

The model predicted tablatures from audio waveform for each and every time frames. This meant that the entirety of the music score was upheld by the onset detection for the accuracy. Through extensive research, the author resided to utlising an onset backtracking due to the onsets tended to miss the actual frame that contained actual prediction by a few frames. Thus, the method of backtracking to some previous frame would be beneficial to gain accuracy. Fortunately, Librosa provided the base function that would allow the better detection to be achieved via the onset_backtrack function, which takes in the computation of the root-mean-square value from the matrix value of the spectrogram as illustrated in Figure 99 below.



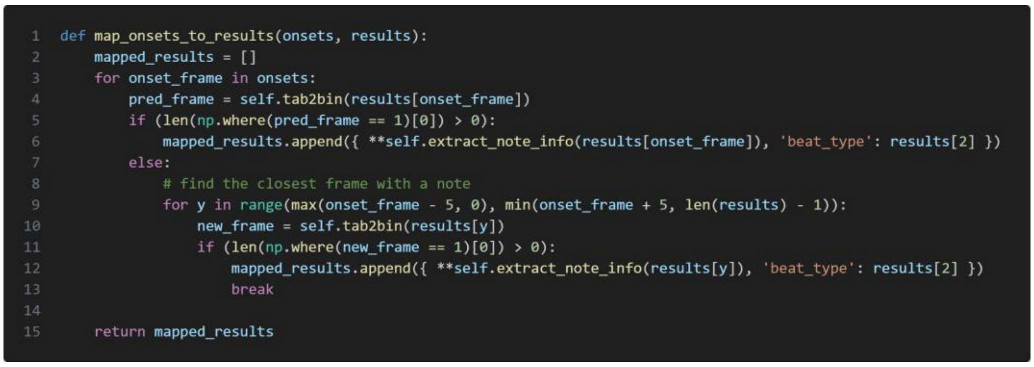
```
rms = librosa.feature.rms(S=new_cqt)
onset_bt_rms = librosa.onset.onset_backtrack(onset_raw, rms[0])
onsets1 = librosa.frames_to_time(onset_bt_rms)
```

Figure 99 illustrated the back-tracking onset algorithm of Librosa.

5.9.5 Debugging Missed Notes in Predictions

To tackle the missing notes in prediction which was caused by the onset detection that detected a frame off from the actual frame that contain a playing note, a function was implemented to map the onset to the closest frame that contained a note.

The Figure 100 below showed the function that iterated through the results of extract_note_info function and compute if there was a note in that frame. If the note does not exist find the nearest frame (five frames behind and five frames forward) and kept checking until a note was found.

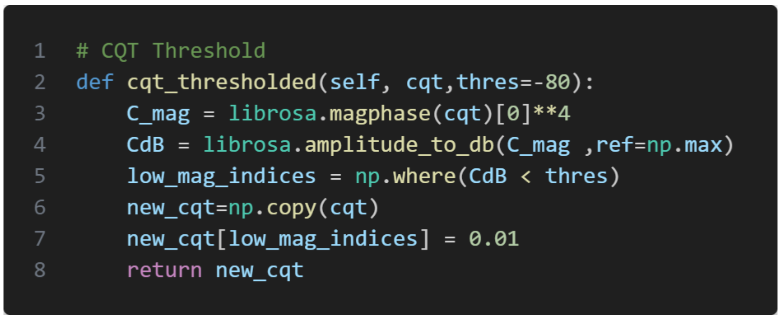


```python
def map_onsets_to_results(onsets, results):
    mapped_results = []
    for onset_frame in onsets:
        pred_frame = self.tab2bin(results[onset_frame])
        if (len(np.where(pred_frame == 1)[0]) > 0):
            mapped_results.append({ **self.extract_note_info(results[onset_frame]), 'beat_type': results[2] })
        else:
            # find the closest frame with a note
            for y in range(max(onset_frame - 5, 0), min(onset_frame + 5, len(results) - 1)):
                new_frame = self.tab2bin(results[y])
                if (len(np.where(new_frame == 1)[0]) > 0):
                    mapped_results.append({ **self.extract_note_info(results[y]), 'beat_type': results[2] })
                    break

    return mapped_results
```

Figure 100 illustrated the map function that helped extract the note/chord from the predicition.

5.9.6 Remove Noises from CQT

Another method found through extensive research was the use of threshold to essentially lower the noises from previous frame that could bled through to the adjacent frame. Though, the entirety of these noises could not be completely removed to sustain the actual note being played. The function shown in Figure 101 converted the matrix of CQT back to waveform amplitude. This amplitude would then be converted to the loudness in Decibels, where only the index of each loudness below the given threshold would be stored on a list for removal.



```python
# CQT Threshold
def cqt_thresholded(self, cqt,thres=-80):
    C_mag = librosa.magphase(cqt)[0]**4
    CdB = librosa.amplitude_to_db(C_mag ,ref=np.max)
    low_mag_indices = np.where(CdB < thres)
    new_cqt=np.copy(cqt)
    new_cqt[low_mag_indices] = 0.01
    return new_cqt
```

Figure 101 illustrated the function that helped lower the noise that was bled from the previous note in the song.

5.10 Sprint 6

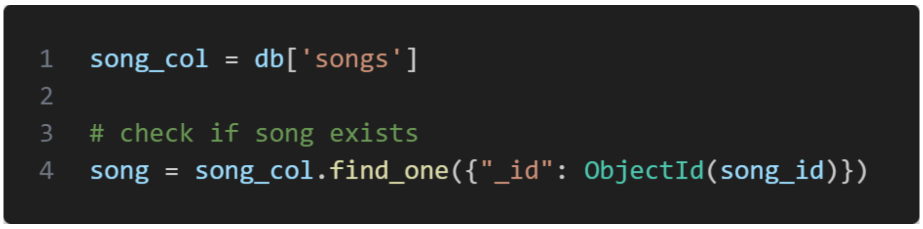
5.10.1 Goal

The interim had made it clear that SQL Server on Azure could not be utilised and needed to be migrated elsewhere. The missing features that were mentioned in the functional requirements were also needed to implement and to update the Front-end application to fit what was described in the requirements and the feasibility section.

5.10.2 Database Migration

After the interim presentation, the author realised that the server on Azure was also utilised by the architect of the college. This caused them to block any ingress routes from unauthorised users that were not of staff members. Thus, the database needed to be migrated a service outside of Azure and the author would know there would not cause another interruption in the application development. The database was then chosen to be hosted on MongoDB as the author had had previous knowledge in developing from previous college works that the service worked without an interference.

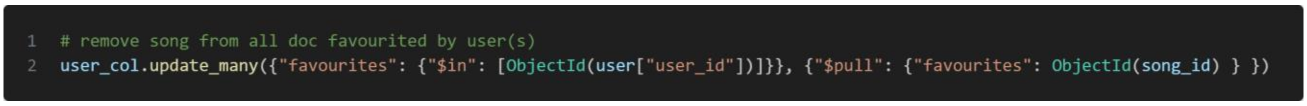
The migration had been accomplished in a straight-forward manner, where a lot of the syntax were similar to the JavaScript counterpart of the library. Through this change came a better developer experience that provided descriptive and semantic names and easy to use API functions that helped filter and query to the DB as shown in Figure 102 below.



```
song_col = db['songs']

# check if song exists
song = song_col.find_one({"_id": ObjectId(song_id)})
```

Figure 102 illustrated the query to retrieve the songs from the database.

Additionally, the structure of the DB and its relational links needed to be changed to fit the No-SQL structure type. As they were two entirely different types of SQL, the method to implement relationships were totally contrast to each other. For instance, the SQL database would link its relationship via creating a foreign key constraint to the primary key of the other table, where the relationship could be called through the use of a join query. At the same time, the No-SQL database utilised the referencing method to store an ID (reference) to a document, which could be called through the use of aggregation pipeline to join the two documents. Through the incredibly versatile API of Pymongo, a more complex work such as looking through the collection of documents to query for documents with certain value were able to be easily searched through the collection via a mere line of code shown in Figure 103.



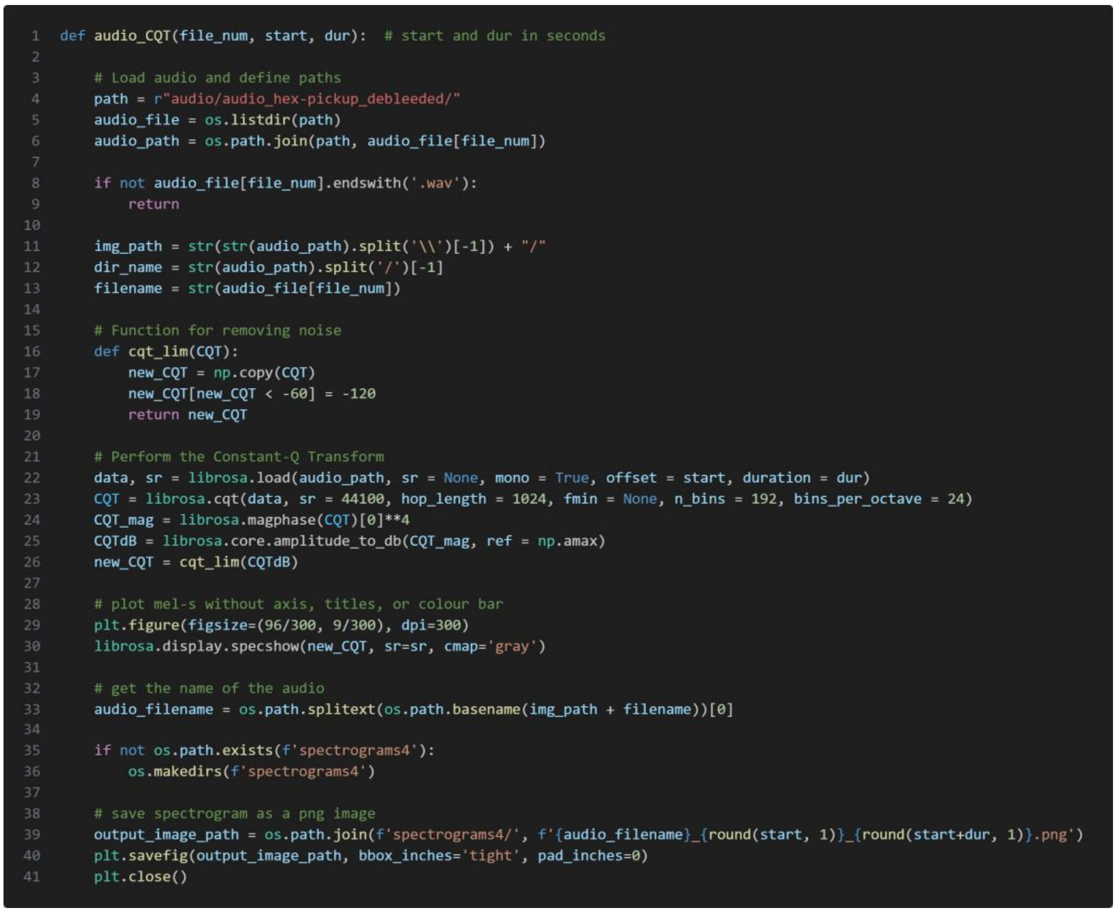
```python
# remove song from all doc favourited by user(s)
user_col.update_many({"favourites": {"$in": [ObjectId(user["user_id"])]}}, {"$pull": {"favourites": ObjectId(song_id) } })
```

Figure 103 illustrated the query that update the favourite of a song into the database.

5.10.3 Utilisation of Spectrograms for AMT

During this Sprint, the author had decided to attempt to utilised the model from Jadhav et al. (2022), where they utilised the images of an actual spectrograms to train the model and the structures of the model was recreated from the description of the paper.

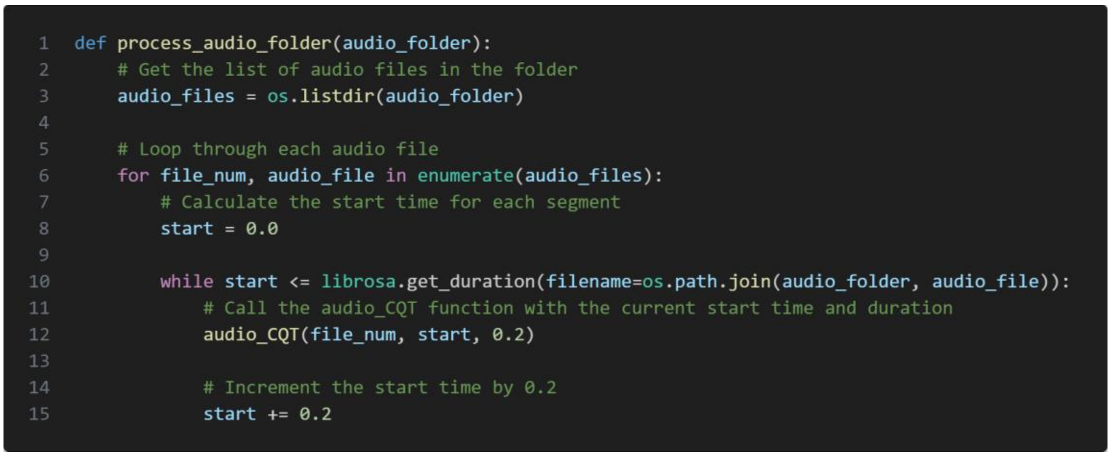
To do this, the audio must first need to be converted into a spectrogram image of duration 0.2 seconds. The Figure 104 shown the function that was utilised to convert the waveform to an image. Essentially, the function would take in the index of the file to load the audio using Librosa. Then a noise suppression would be applied to the audio to lower the bleeding of sounds that was lower than 60 dB to prevent sounds from affecting the prediction of its adjacent note/chord. The suppressed CQT waveform would finally be plot on the figure and saved onto the folder.



```python
def audio_CQT(file_num, start, dur):  # start and dur in seconds

    # Load audio and define paths
    path = r"audio/audio_hex-pickup_debleeded/"
    audio_file = os.listdir(path)
    audio_path = os.path.join(path, audio_file[file_num])

    if not audio_file[file_num].endswith('.wav'):
        return

    img_path = str(str(audio_path).split('\\')[-1]) + "/"
    dir_name = str(audio_path).split('/')[-1]
    filename = str(audio_file[file_num])

    # Function for removing noise
    def cqt_lim(CQT):
        new_CQT = np.copy(CQT)
        new_CQT[new_CQT < -60] = -120
        return new_CQT

    # Perform the Constant-Q Transform
    data, sr = librosa.load(audio_path, sr = None, mono = True, offset = start, duration = dur)
    CQT = librosa.cqt(data, sr = 44100, hop_length = 1024, fmin = None, n_bins = 192, bins_per_octave = 24)
    CQT_mag = librosa.magphase(CQT)[0]**4
    CQTdB = librosa.core.amplitude_to_db(CQT_mag, ref = np.amax)
    new_CQT = cqt_lim(CQTdB)

    # plot mel-s without axis, titles, or colour bar
    plt.figure(figsize=(96/300, 9/300), dpi=300)
    librosa.display.specshow(new_CQT, sr=sr, cmap='gray')

    # get the name of the audio
    audio_filename = os.path.splitext(os.path.basename(img_path + filename))[0]

    if not os.path.exists(f'spectrograms4'):
        os.makedirs(f'spectrograms4')

    # save spectrogram as a png image
    output_image_path = os.path.join(f'spectrograms4/', f'{audio_filename}_{round(start, 1)}_{round(start+dur, 1)}.png')
    plt.savefig(output_image_path, bbox_inches='tight', pad_inches=0)
    plt.close()
```

Figure 104 illustrated the function that convert 0.2 seconds clip of audio in to a CQT-spectrogram image.

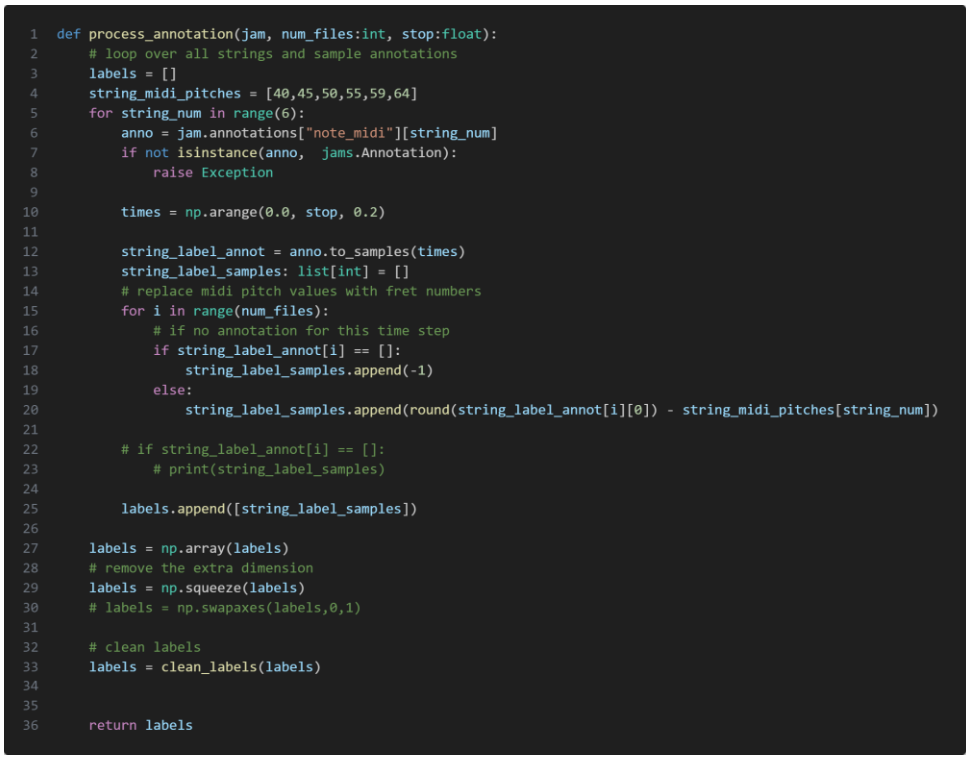
Finally, the For loop, shown in Figure 105, would handle the incrementation of the duration to only extracted 0.2 seconds of the audio to save as the image.



```python
def process_audio_folder(audio_folder):
    # Get the list of audio files in the folder
    audio_files = os.listdir(audio_folder)

    # Loop through each audio file
    for file_num, audio_file in enumerate(audio_files):
        # Calculate the start time for each segment
        start = 0.0

        while start <= librosa.get_duration(filename=os.path.join(audio_folder, audio_file)):
            # Call the audio_CQT function with the current start time and duration
            audio_CQT(file_num, start, 0.2)

            # Increment the start time by 0.2
            start += 0.2
```

Figure 105 illustrated the function that iterate through the audio files and increment 0.2 seconds until the length of the audio duration.

The similar functionality was done to save the labels of the string and fret positions as seen in Figure 106.

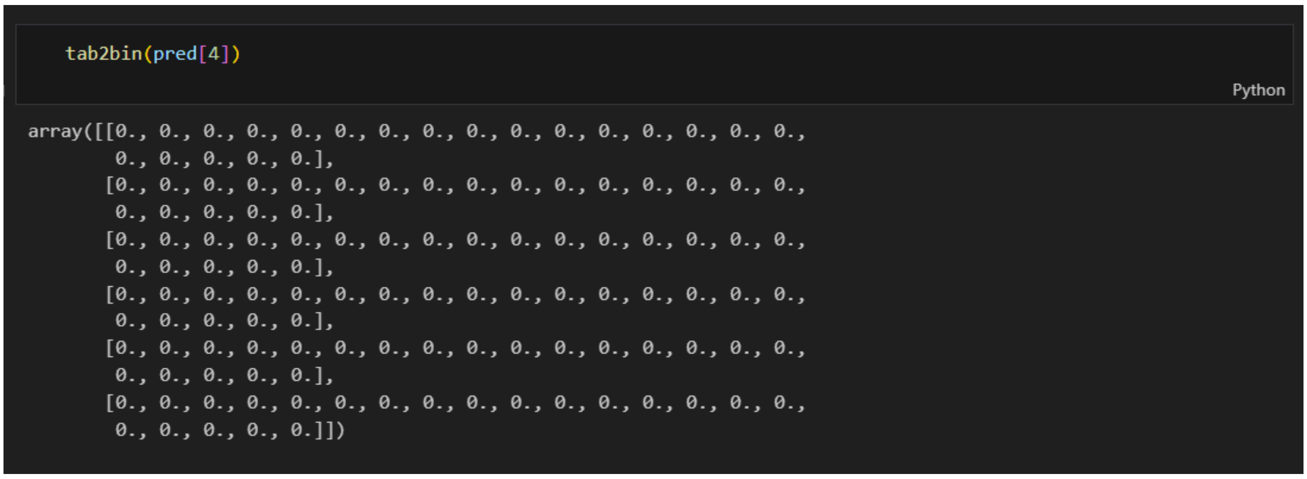


```python
def process_annotation(jam, num_files:int, stop:float):
    # loop over all strings and sample annotations
    labels = []
    string_midi_pitches = [40,45,50,55,59,64]
    for string_num in range(6):
        anno = jam.annotations["note_midi"][string_num]
        if not isinstance(anno,  jams.Annotation):
            raise Exception

        times = np.arange(0.0, stop, 0.2)

        string_label_annot = anno.to_samples(times)
        string_label_samples: list[int] = []
        # replace midi pitch values with fret numbers
        for i in range(num_files):
            # if no annotation for this time step
            if string_label_annot[i] == []:
                string_label_samples.append(-1)
            else:
                string_label_samples.append(round(string_label_annot[i][0]) - string_midi_pitches[string_num])

        # if string_label_annot[i] == []:
            # print(string_label_samples)

        labels.append([string_label_samples])

    labels = np.array(labels)
    # remove the extra dimension
    labels = np.squeeze(labels)
    # labels = np.swapaxes(labels,0,1)

    # clean labels
    labels = clean_labels(labels)


    return labels
```

Figure 106 illustrated the functionality that iterate through the jams file in the increment of 0.2 seconds until the length of the audio duration.

Once all the images of these spectrogram were saved, it needed to be read by the script where the two implementations of the TabCNN and the new paper bridged. The images were converted to matrix of pixel values to be feed to the model.

The Figure shown was the architecture of the model, which was differed from TabCNN's, it takes in the matrix and output six possible outcomes or strings as the prediction. This was also known as multi-task learning. Though there was a challenging problem that had arose from the data, where the trained model was working really well with small data, but once the whole dataset was fed to train the model, it would failed to recognize the string and fret com binations. The result of the prediction was as shown in Figure 107 below.



```python
tab2bin(pred[4])
```

Python

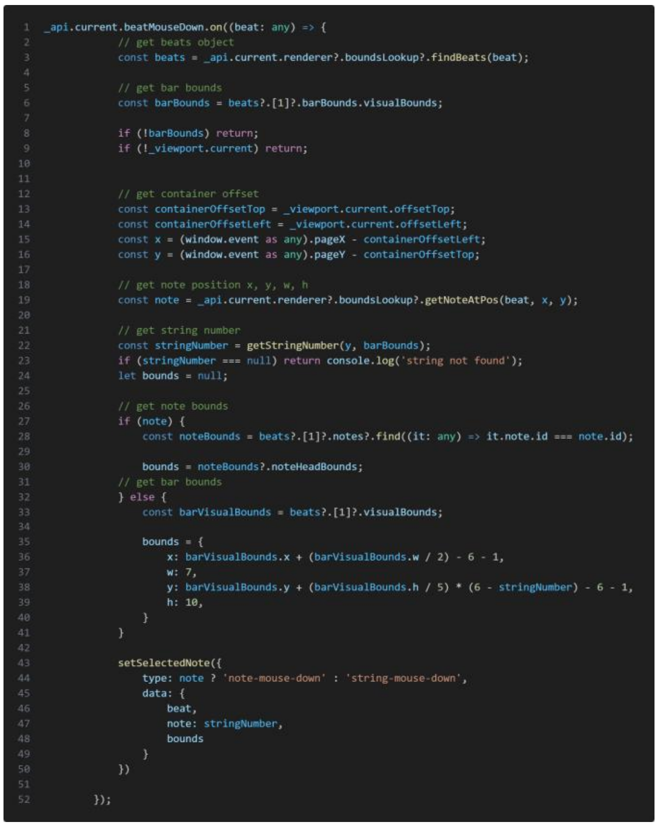
```
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0.]])
```

Figure 107 illustrated the result of the prediction.

5.10.4 Tab Editor

The implementation of an editor would serve as a great way to counter the inaccuracy of the prediction model. By searching through GitHub discussion page of AlphaTab, there were opened discussions about different ways to add an edit functionality to the AlphaTab, as it did not supplied an API to easily edit a note in the music sheet according to Ghost's discussion (2016). Though, the discussion was also commented on by the author of AlphaTab, and further elaborated on the capabilities of its internal API.

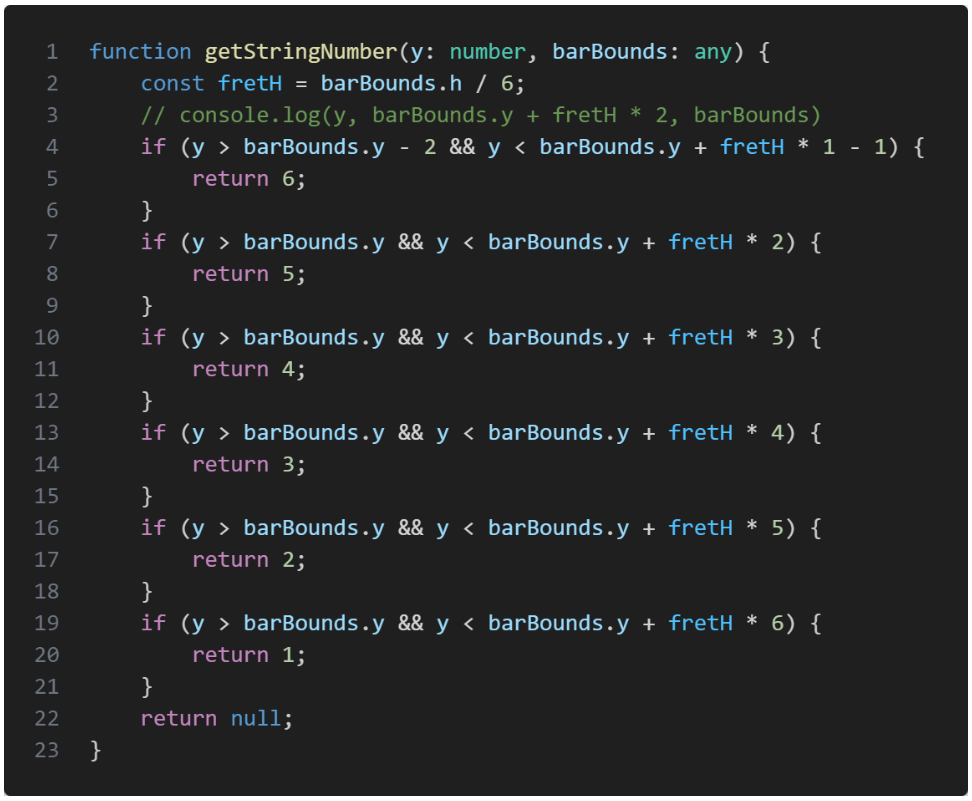
The ability to check all and individual notes, beats, bars, and master bars were implemented, which was provided through the function called 'boundsLookup', where additional functions were housed inside that could be utilized to implement the edit functionality to the author's own application. The note hovered by the mouse cursor could be retrieved via boundsLookup.getNoteAtPos() function, where it took in the currently selected beat, the X and Y coordinate of the mouse as the arguments to determine the note and returned its object. This function would be used in conjunction with the event listener to call to a callback to find the selected note on Figure 108.



```
_api.current.beatMouseDown.on((beat: any) => {
            // get beats object
            const beats = _api.current.renderer?.boundsLookup?.findBeats(beat);

            // get bar bounds
            const barBounds = beats?.[1]?.barBounds.visualBounds;

            if (!barBounds) return;
            if (!_viewport.current) return;


            // get container offset
            const containerOffsetTop = _viewport.current.offsetTop;
            const containerOffsetLeft = _viewport.current.offsetLeft;
            const x = (window.event as any).pageX - containerOffsetLeft;
            const y = (window.event as any).pageY - containerOffsetTop;

            // get note position x, y, w, h
            const note = _api.current.renderer?.boundsLookup?.getNoteAtPos(beat, x, y);

            // get string number
            const stringNumber = getStringNumber(y, barBounds);
            if (stringNumber === null) return console.log('string not found');
            let bounds = null;

            // get note bounds
            if (note) {
                const noteBounds = beats?.[1]?.notes?.find((it: any) => it.note.id === note.id);

                bounds = noteBounds?.noteHeadBounds;
            // get bar bounds
            } else {
                const barVisualBounds = beats?.[1]?.visualBounds;

                bounds = {
                    x: barVisualBounds.x + (barVisualBounds.w / 2) - 6 - 1,
                    w: 7,
                    y: barVisualBounds.y + (barVisualBounds.h / 5) * (6 - stringNumber) - 6 - 1,
                    h: 10,
                }
            }

            setSelectedNote({
                type: note ? 'note-mouse-down' : 'string-mouse-down',
                data: {
                    beat,
                    note: stringNumber,
                    bounds
                }
            })

        });
```

Figure 108 illustrated the function that check for notes, beats, bars, and master bars.

Once the event listener had been fired, it would capture the beat that was selected. This was handy as it could be fed into the findBeats function to get the visual boundaries of the beat, where the location of the string number could be computed through the getStringNumber function shown in the Figure 109 below.



```
function getStringNumber(y: number, barBounds: any) {
    const fretH = barBounds.h / 6;
    // console.log(y, barBounds.y + fretH * 2, barBounds)
    if (y > barBounds.y - 2 && y < barBounds.y + fretH * 1 - 1) {
        return 6;
    }
    if (y > barBounds.y && y < barBounds.y + fretH * 2) {
        return 5;
    }
    if (y > barBounds.y && y < barBounds.y + fretH * 3) {
        return 4;
    }
    if (y > barBounds.y && y < barBounds.y + fretH * 4) {
        return 3;
    }
    if (y > barBounds.y && y < barBounds.y + fretH * 5) {
        return 2;
    }
    if (y > barBounds.y && y < barBounds.y + fretH * 6) {
        return 1;
    }
    return null;
}
```

Figure 109 illustrated the function that computed the nearest string number based on position of the mouse.

The string number needed to be located such that in the instance that the selected note was empty, the placeholder input needed to be put in place around the string that was clicked. Otherwise, the existing note would be highlighted by the placeholder input to numbers could be specified to change the note.

5.10.5 Favourite

The users needed to able to favourite the song they want to re-visit later. To do this, the feature needed to be implemented on both the client and server, where the client would handle the visualisation of the list of favourites, while the server handled the query to Create, Read, Update, and Delete the favourite from the user, and song.

Favourite Page

Front-end Implementation

The feature had been quickly implemented as a Hi-Fi prototype, illustrated in Figure 110, to allow the author to visually characterise requirements needed for the page to function.

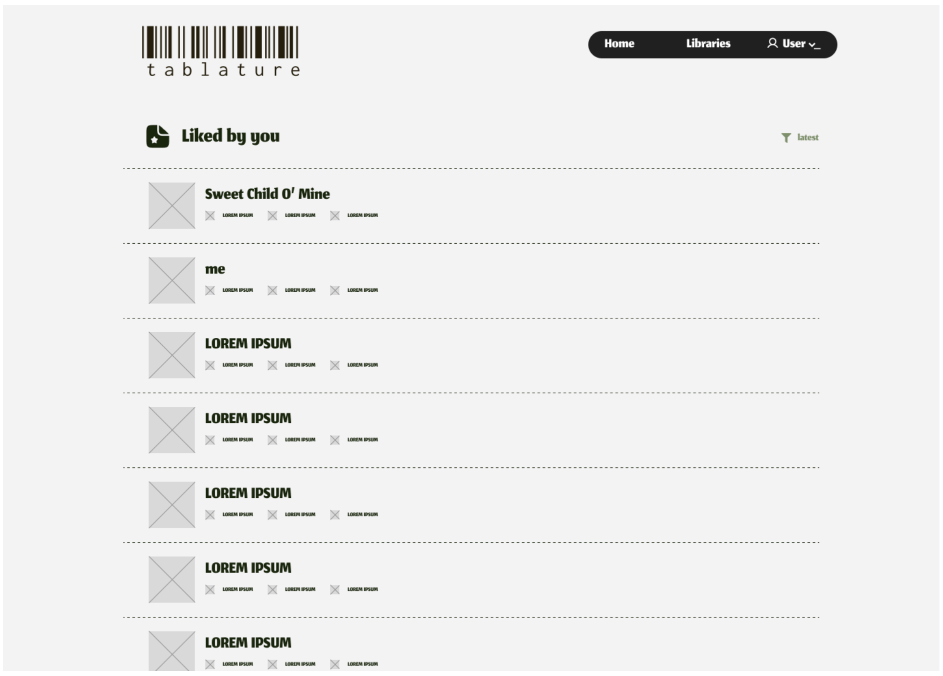
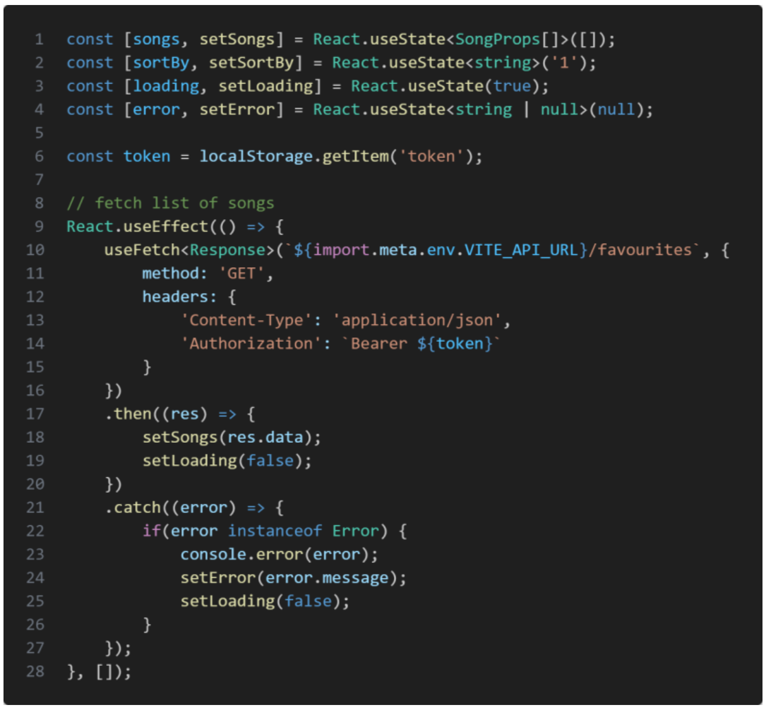


Figure 110 illustrated the Hi-Fi prototype of the Favourite page.

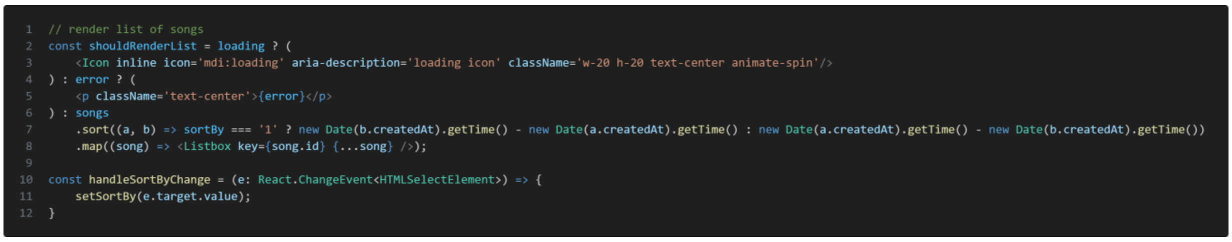
The component would fetch to the server once the page had been mounted via useEffect, where their JWT token would be passed along the headers to provide the server with the user information to retrieve their favourited songs illustrated in Figure 111.



```
const [songs, setSongs] = React.useState<SongProps[]>([]);
const [sortBy, setSortBy] = React.useState<string>('1');
const [loading, setLoading] = React.useState(true);
const [error, setError] = React.useState<string | null>(null);

const token = localStorage.getItem('token');

// fetch list of songs
React.useEffect(() => {
    useFetch<Response>(`${import.meta.env.VITE_API_URL}/favourites`, {
        method: 'GET',
        headers: {
            'Content-Type': 'application/json',
            'Authorization': `Bearer ${token}`
        }
    })
    .then((res) => {
        setSongs(res.data);
        setLoading(false);
    })
    .catch((error) => {
        if(error instanceof Error) {
            console.error(error);
            setError(error.message);
            setLoading(false);
        }
    });
}, []);
```

Figure 111 illustrated the fetch API that retrieved all the songs favourited by the user.

These favourites would then be stored using a useState hook and the data would be mapped into a list of components for each favourite as shown in Figure 112

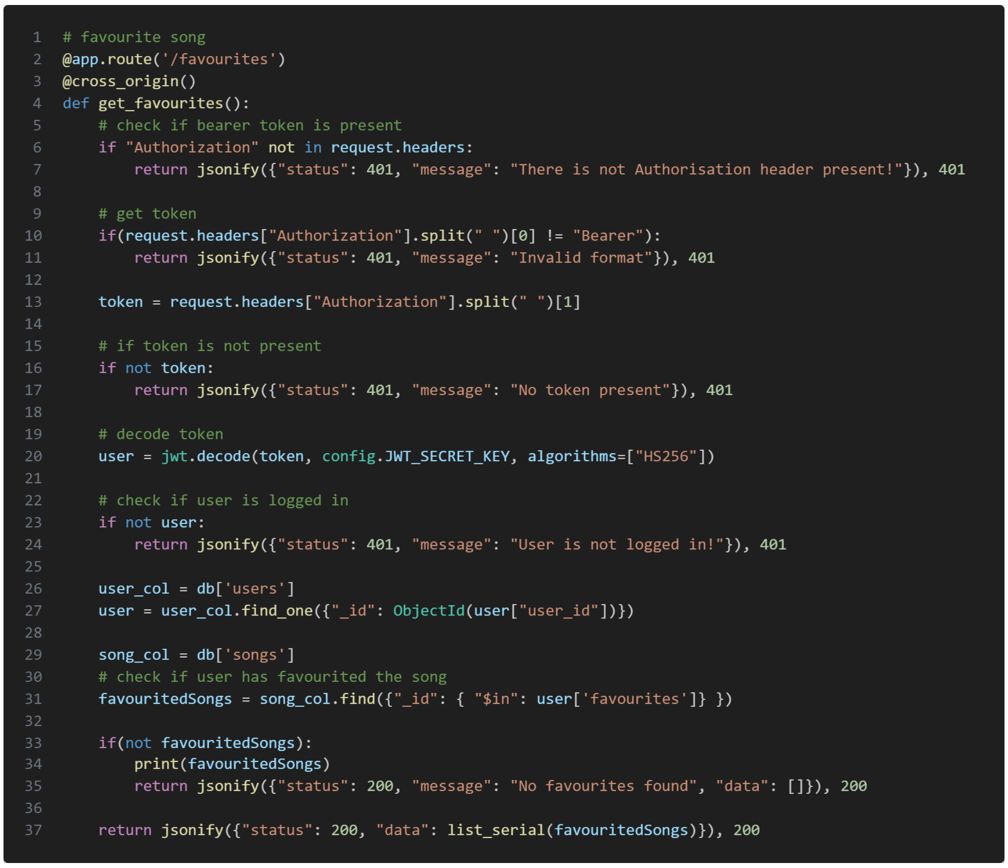


```
// render list of songs
const shouldRenderList = loading ? (
    <Icon inline icon='mdi:loading' aria-description='loading icon' className='w-20 h-20 text-center animate-spin'/>
) : error ? (
    <p className='text-center'>{error}</p>
) : songs
    .sort((a, b) => sortBy === '1' ? new Date(b.createdAt).getTime() - new Date(a.createdAt).getTime() : new Date(a.createdAt).getTime() - new Date(b.createdAt).getTime())
    .map((song) => <Listbox key={song.id} {...song} />);

const handleSortByChange = (e: React.ChangeEvent<HTMLSelectElement>) => {
    setSortBy(e.target.value);
}
```

Figure 112 illustrated the mapping function that iterated through the array of favourites and map it into the component.

Back-end Implementation

The Figure 113 shown below handle the retrieval of the song(s) favourited by the user. The API endpoint would need to acquire the token from the user, which would be passed from the client. This token would then be decoded through the JWT library to retrieve the ID of the user. Once the user ID had been retrieved the server would execute a search query to the database with the user ID and the favourited songs were able to be sent back to the client through the form of a JSON response.



```
# favourite song
@app.route('/favourites')
@cross_origin()
def get_favourites():
    # check if bearer token is present
    if "Authorization" not in request.headers:
        return jsonify({"status": 401, "message": "There is not Authorisation header present!"}), 401

    # get token
    if(request.headers["Authorization"].split(" ")[0] != "Bearer"):
        return jsonify({"status": 401, "message": "Invalid format"}), 401

    token = request.headers["Authorization"].split(" ")[1]

    # if token is not present
    if not token:
        return jsonify({"status": 401, "message": "No token present"}), 401

    # decode token
    user = jwt.decode(token, config.JWT_SECRET_KEY, algorithms=["HS256"])

    # check if user is logged in
    if not user:
        return jsonify({"status": 401, "message": "User is not logged in!"}), 401

    user_col = db['users']
    user = user_col.find_one({"_id": ObjectId(user["user_id"])})

    song_col = db['songs']
    # check if user has favourited the song
    favouritedSongs = song_col.find({"_id": { "$in": user['favourites']} })

    if(not favouritedSongs):
        print(favouritedSongs)
        return jsonify({"status": 200, "message": "No favourites found", "data": []}), 200

    return jsonify({"status": 200, "data": list_serial(favouritedSongs)}), 200
```

Figure 113 illustrated the retrieval of the songs favourited by the user.

5.10.6 Forking songs

The ability to edit the tablature had been developed as a conceptual implementation. This meant that the editability was met in terms of its functional requirement to select, edit, and add new note into the tablature. Though, the non-functional features that were not be able in the course of the project were the ability to add different beat types, adding new bar to the music sheet, and the ability to delete the note was not added.

Fork Edits

To be able to edit the existing tab, the edit button that had been added to the current tab would navigate to the edit page as shown in the Figure 114 and 115. Once the new page was mounted, it would fetch to the server using the token of the currently logged-in user, in order to assert the user id as the owner of the new edited tab and inserted it alongside the filename for the client to locate the MusicXML and audio files on Azure Blob Storage.

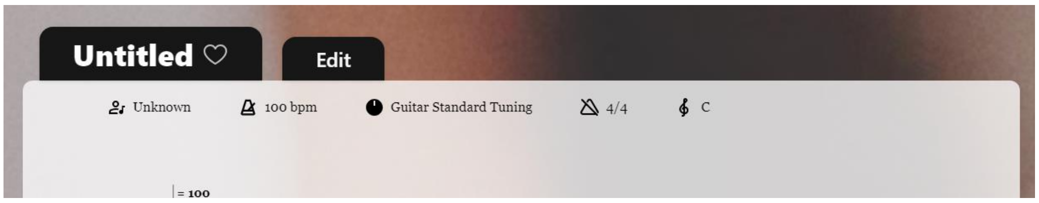


Figure 114 illustrated the edit button on the Tab Visualiser page.

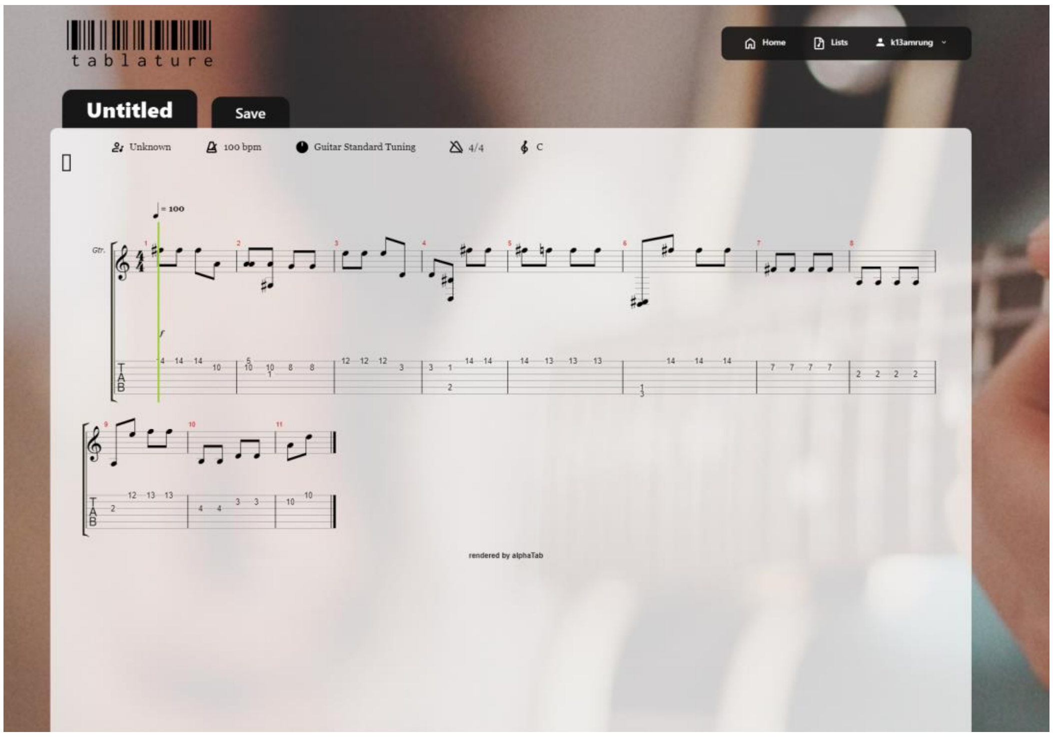


Figure 115 illustrated the Edit page.

Update Edits

Once the user was logged in and able to use the edit page, the save button, illustrated in Figure 116, would be put in place of the edit button to send an API request to the server, where it would attempt to overwrite the existing MusicXML file saved on Blob Storage.

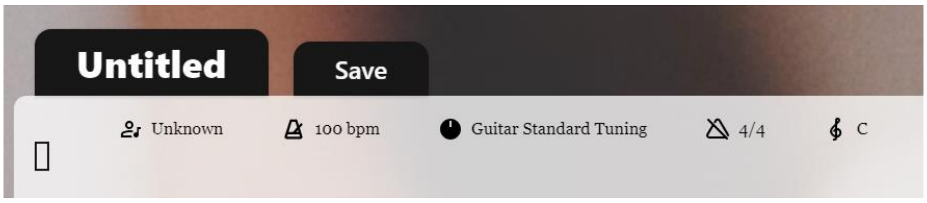


Figure 116 illustrated the save button on the Edit page.

Delete Edits

In the case that the song needed to be removed, the delete button could be utilised which also send an API request to the server, where it would be checked against the token to validate the user’s ownership to the edit. If the validation was met, the MusicXML file would be removed from the Blob Storage along with the document in MongoDB. Otherwise, the message would be returned to the client stating that the user was not an authorised user and cannot perform the action.

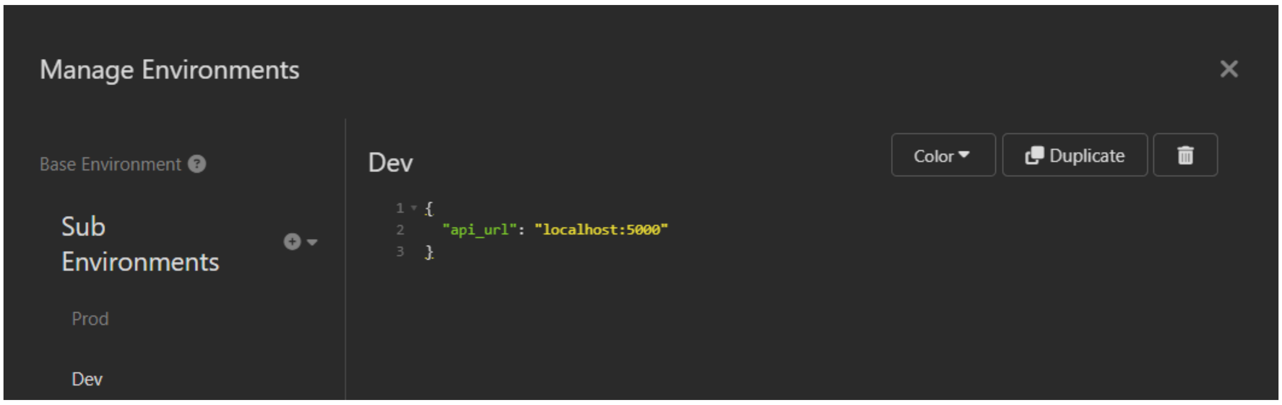
5.11 Sprint 7

5.11.1 Goal

The application needed to be tested to further fine-tune the application and fix possible errors. The Sprint would mainly be targeted to debug both Front-end and Back-end applications.

5.11.2 Unit testing

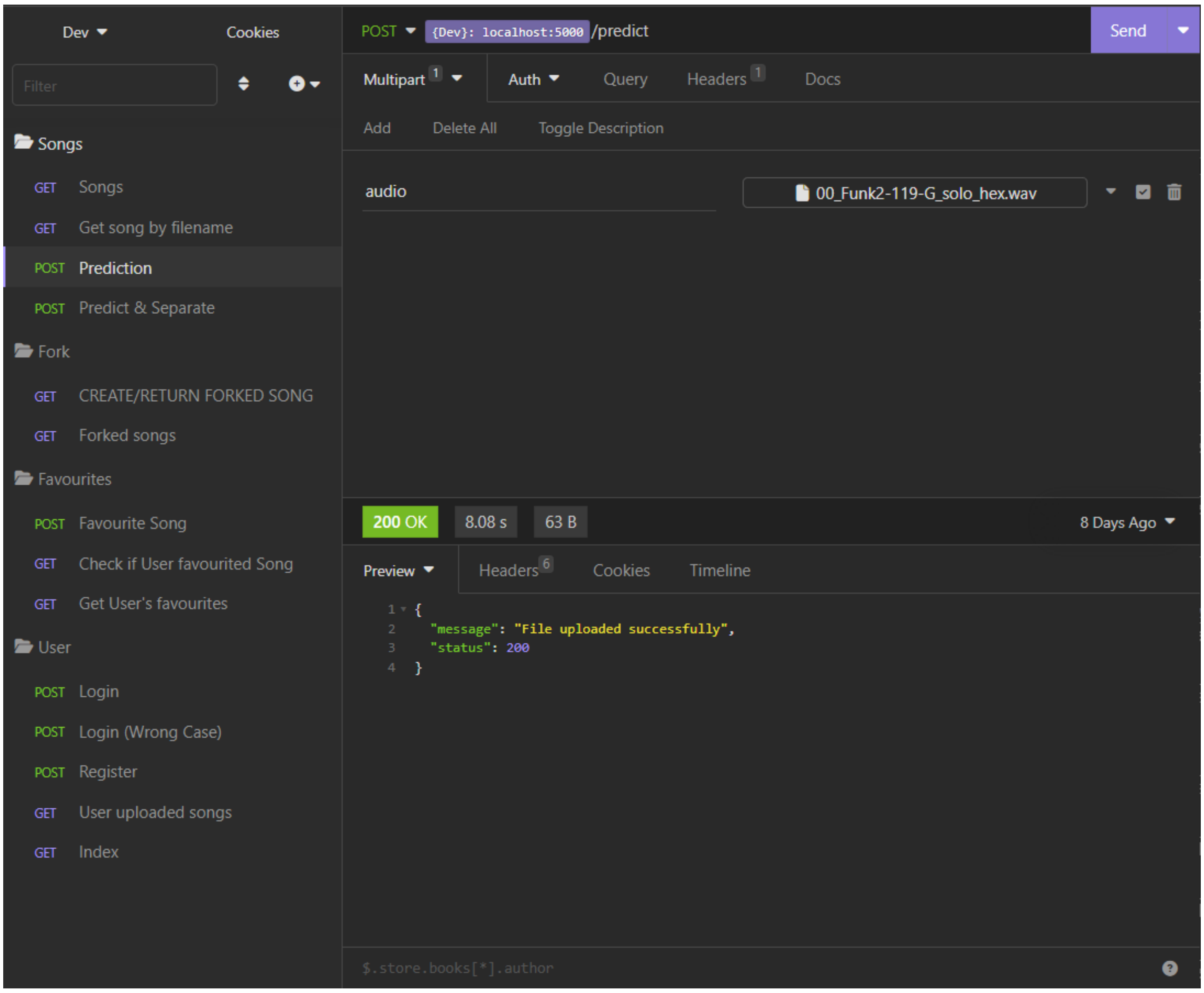
The application had been tested via Insomnia to validate and ensure each route returned with the correct responses. Since the application would be hosted on a cloud provider, the software needed to be able to change between the production and development environments i.e. the URL needed to change accordingly. These environments could be easily created through the environments tab, where each environment could be created, and the API URL would be inserted as shown in Figure 117 below.



```
{
  "api_url": "localhost:5000"
}
```

Figure 117 illustrated the software that the author used for unit testing.

From there, the respective routes would be categorised by its route groups such as songs, favourites, and users. Each route, shown in Figure 118, would be tested via a variety of test cases such as no form input, incorrect input(s), and vice versa.



POST {Dev}: localhost:5000/predict

audio: 00_Funk2-119-G_solo_hex.wav

200 OK 8.08 s 63 B

```
{
  "message": "File uploaded successfully",
  "status": 200
}
```

Figure 118 illustrated the unit-testing of different API endpoints

5.11.3 User testing

The user testing was performed after basic functionality of the application was fully implemented and working as intended. By doing user testing, the author was able to evaluate the user experience and whether the feature was easy for the user to utilise. To conduct the user testing, the author had created multiple tasks for the testers to follow, where the flow of the application would be tested alongside the AI prediction. The following list were the series of tasks set out to be completed by the applicant:

1. Create an account.
2. Upload an audio.
3. Find the uploaded song(s).
4. Favourite a song.
5. Edit a song.

6. Save the song.
7. Find existing forked song.
8. Logout from the site.

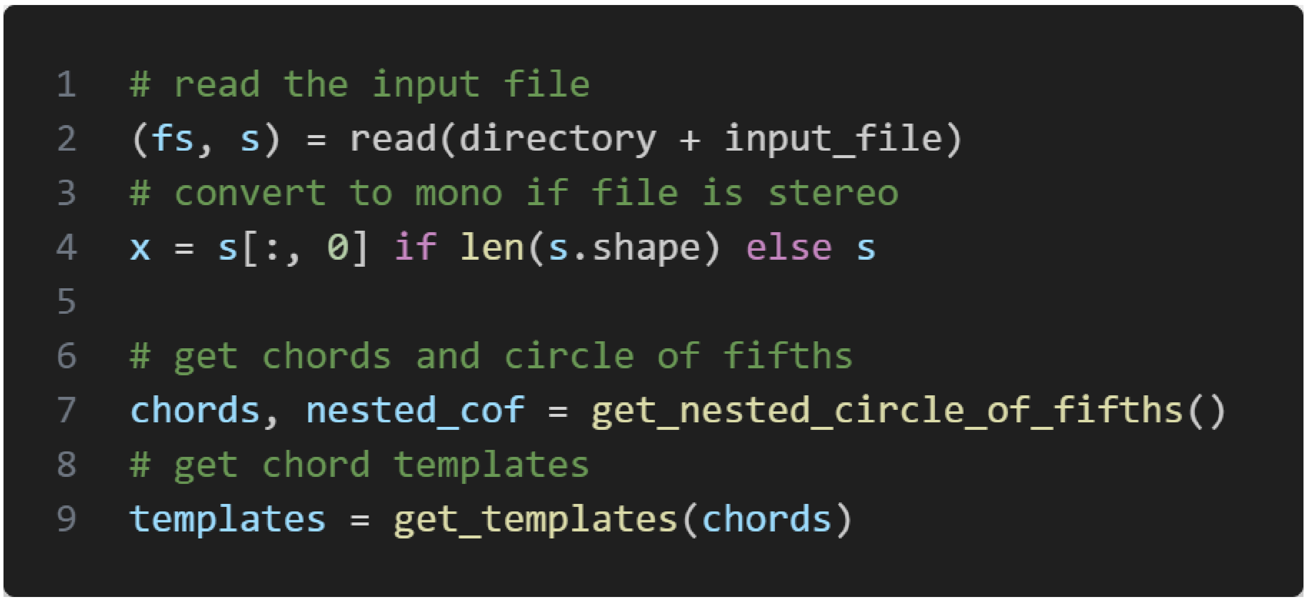
Although the applicant would be given the same instructions, some would be given an audio with a different way to retrieve the prediction, such as an audio that needed separation before prediction, and an audio with only guitar track. The test would aim to see the user's navigational flow to see if the application was difficult to navigate or needed changing.

The user testing phrase then allowed the author to gain a better insight into the outsiders' perspectives to test the application's functionality and accessibility. The result of the tests was used to make necessary changes to the UI and UX, where bugs and major patches would be applied to the final application.

5.11.4 Adding chord detection

The final features that had been recommended by the supervisor was to add some sort of chord algorithm, which would help a guitarist a hint of the chord progression in a song. Though due to the short time constraints, it had hindered the author from taking the research route, and instead, only stick to existing model and chord detection algorithm. The existing model/algorithm that could be found were the Autochord and Chord-Recognition repository. Even though, the Autochord was easier to implement and had a better developer experience with more documentation. It only could be installed in Python version 2.7, due to the discoutinued package that only could run in the version mentioned or lower. Thus, the only choice was to utilise the Chord-Recognition repository from Orchidas.

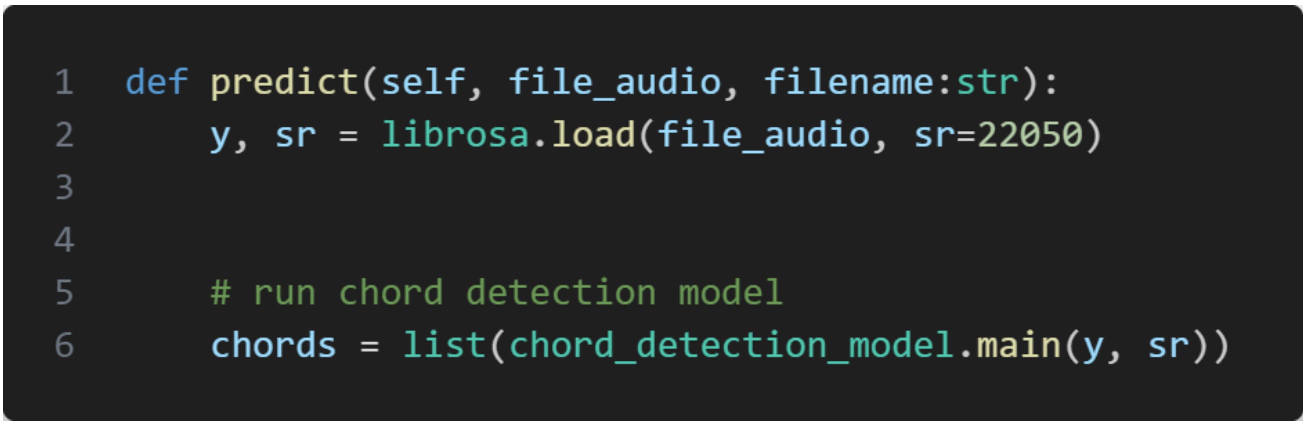
The detector inherited from two papers which utilised two entirely different algorithms called 'Template matching' and 'Hidden Markov Model' (HMM). Originally, the detector would be executed on launch and only took in actual audio from a folder and output timestamp and their respective chord estimations, as well as an optional CQT spectrogram. In order to utilise the algorithm in the current state of the application, the script needed to be altered to run as a function rather than on execution and it needed to use the bytes of audio to process the output. Luckily, the main function utilised the same type of argument as the prediction model with an exception of a different library to load the audio, which being Scipy shown in Figure 119. The only process needed to be altered was to change the audio-loading method to utilised Librosa.



```python
# read the input file
(fs, s) = read(directory + input_file)
# convert to mono if file is stereo
x = s[:, 0] if len(s.shape) else s

# get chords and circle of fifths
chords, nested_cof = get_nested_circle_of_fifths()
# get chord templates
templates = get_templates(chords)
```

Figure 119 illustrated the way Chord-Recognition repository load their audio.

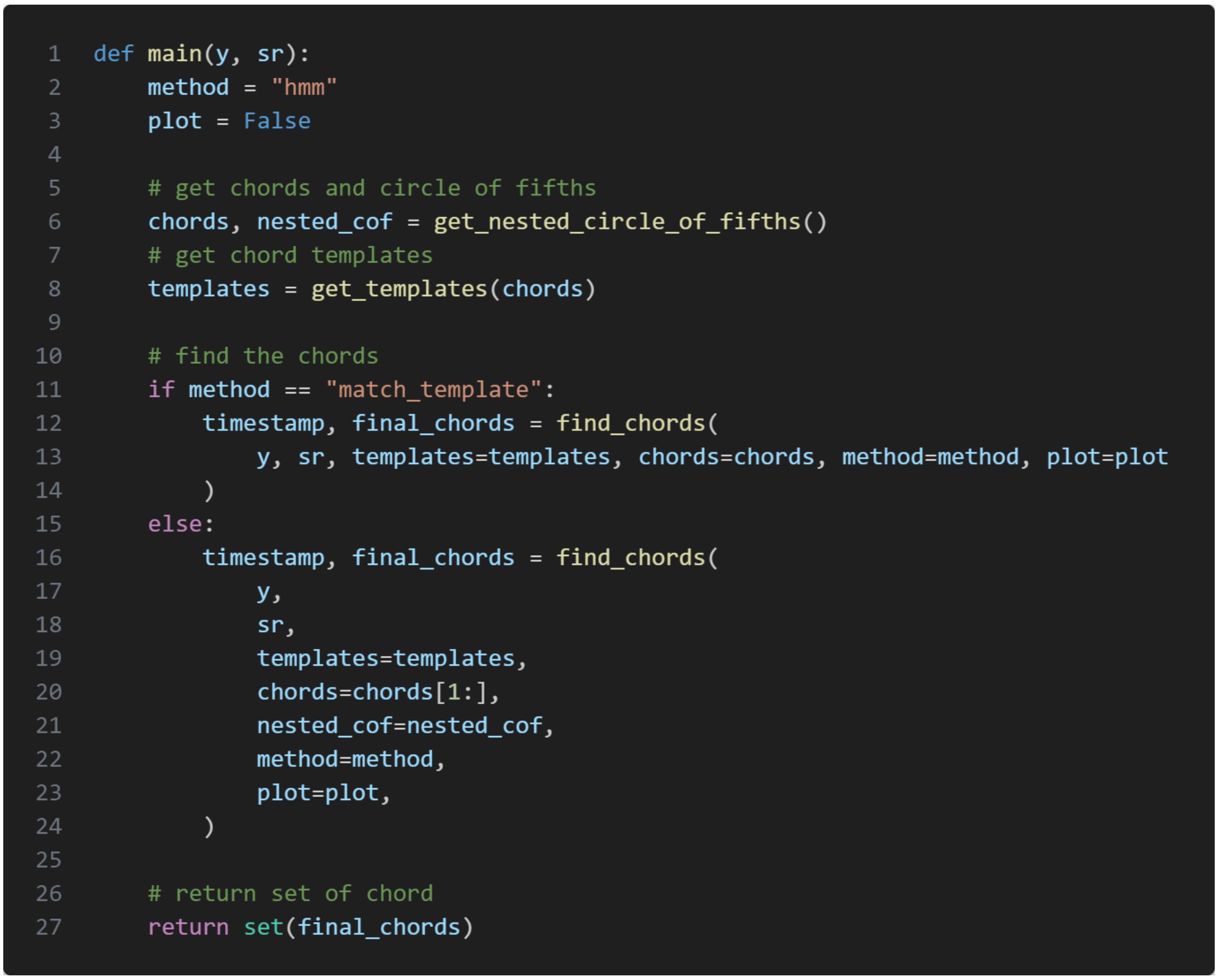
Instead, the chord detection would be executed in the prediction model, where Librosa would load the audio once to avoid the error that caused from loading two audio simultaneously as shown in Figure 120.



```python
def predict(self, file_audio, filename:str):
    y, sr = librosa.load(file_audio, sr=22050)


    # run chord detection model
    chords = list(chord_detection_model.main(y, sr))
```

Figure 120 illustrated the change made to load the audio.

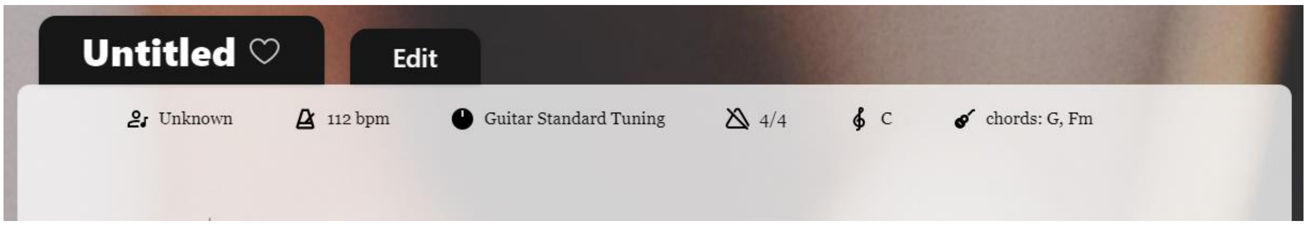
Once the audio had been loaded the matrix Y and the sample rate would be passed as the argument to the chord detection model, where the result would be output as a set to eliminate the same note from the list as illustrated in Figure 121.



```python
def main(y, sr):
    method = "hmm"
    plot = False

    # get chords and circle of fifths
    chords, nested_cof = get_nested_circle_of_fifths()
    # get chord templates
    templates = get_templates(chords)

    # find the chords
    if method == "match_template":
        timestamp, final_chords = find_chords(
            y, sr, templates=templates, chords=chords, method=method, plot=plot
        )
    else:
        timestamp, final_chords = find_chords(
            y,
            sr,
            templates=templates,
            chords=chords[1:],
            nested_cof=nested_cof,
            method=method,
            plot=plot,
        )

    # return set of chord
    return set(final_chords)
```

Figure 121 illustrated the output of the chord detection model.

The set of chords would then passed as the output of the tab prediction model and stored in the DB to be shown in the Front-end as illustrated in Figure () below.



Untitled ♡ Edit

Unknown 112 bpm Guitar Standard Tuning 4/4 C chords: G, Fm

Figure 122 illustrated the display of the chord detection in the Front-end application.

5.12 Sprint 8

5.12.1 Goal

The final Sprint would conclude the journey of the project, where no new features would be implemented, only debugging and testing of features would be done during this final stretch.

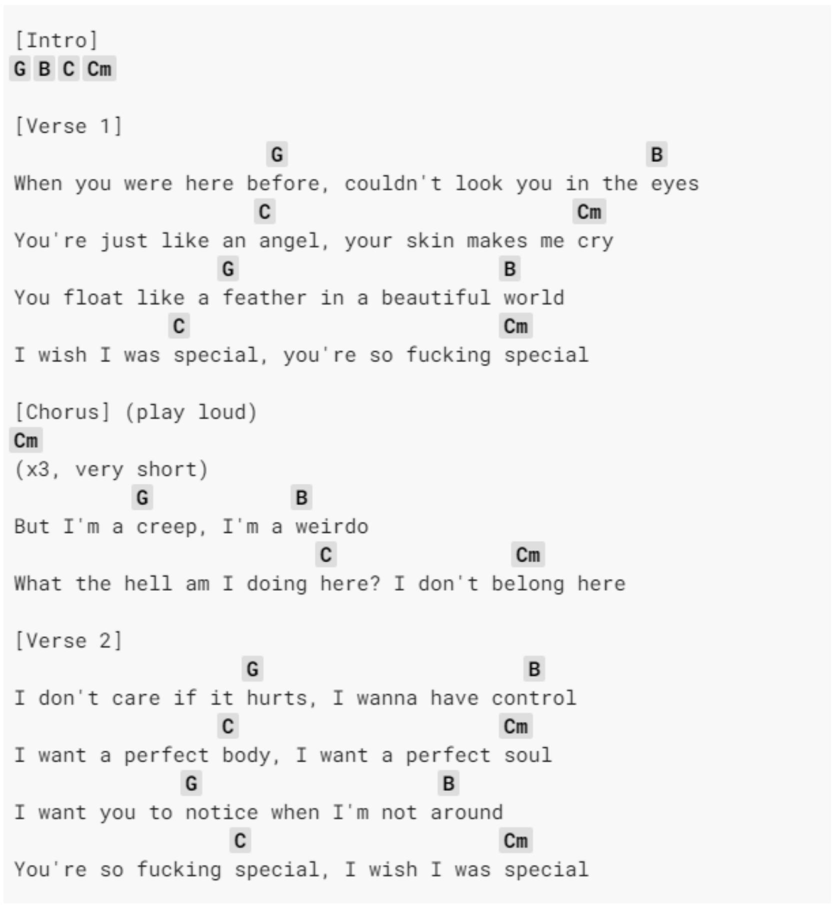
5.12.2 Audio Analysis

Since the sound of an audio played a huge factor in achieving results. It was important to compare each audio to see what worked best for the current state of the model. Therefore, some audios of different variants were used to see this through:

Test No.	Test Case	Song	Duration (MM: SS)	Output
1.	Acoustic guitar track with just chords	Creep – Radiohead	00:30	The chords were transcribed pretty accurately in terms of its pitch and sound.
2.	Guitar track with just a solo / arpeggio	Creep – Radiohead (arpeggio)	00:20	The arpeggio became muddy due to the bleed of the guitar rhythm, but the pitch was still pretty accurate.
3.	Mixture of just solo with distortion effect on guitar	Let It Be	00:30	The solo was transcribed quite poorly where there were a lot of bleed form previous notes and caused some artifacts of chords to become present.
4.	Mixture of both chords and solo(s) with more than one guitar track with low distortion on guitar	Calling After Me - Wallow	03:40	The prediction presented a lot of breeds throughout the song. This was due to having two guitars playing simultaneously.

The best result from the audio testing would be the audio of acoustic guitars that were strumming chords. While, the results from the acoustic guitars that were playing notes would cause the sound of the previous to bled through to the adjacent notes, which resulted in the model estimating the sound to have more than one note when it should not appear to be.

The result from test case number one and two, where the sound came from an acoustic of the song Creep by Radiohead. The song consisted of simple chord progressions and strumming patterns, which was suggested the reason of its accuracy, not just in terms of the pitch accuracy, but the chord shapes that it predicted. The chords of G, B, and C, and C minor was recognised by the model, of which was the actual chords of the song that was played in the audio shown in Figure 123. While the prediction had output the similar result as illustrated in Figure 124 below.



```
[Intro]
G B C Cm

[Verse 1]
                     G                              B
When you were here before, couldn't look you in the eyes
                    C                         Cm
You're just like an angel, your skin makes me cry
                 G                      B
You float like a feather in a beautiful world
             C                          Cm
I wish I was special, you're so fucking special

[Chorus] (play loud)
Cm
(x3, very short)
          G            B
But I'm a creep, I'm a weirdo
                         C                Cm
What the hell am I doing here? I don't belong here

[Verse 2]
                   G                      B
I don't care if it hurts, I wanna have control
                 C                      Cm
I want a perfect body, I want a perfect soul
              G                    B
I want you to notice when I'm not around
                  C                     Cm
You're so fucking special, I wish I was special
```

Figure 123 illustrated the chords of Creep by Radiohead.

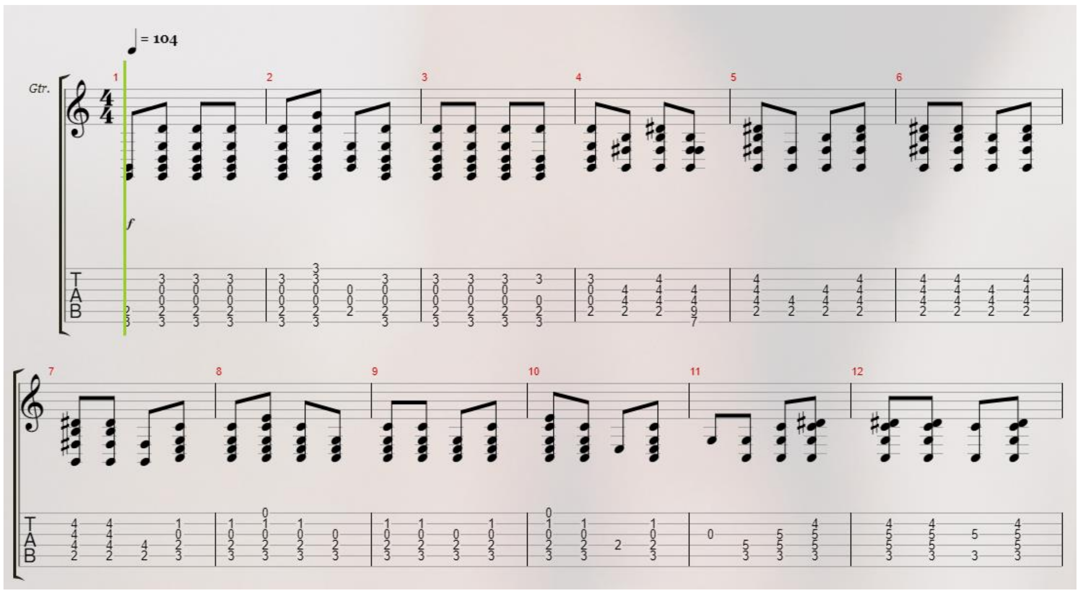
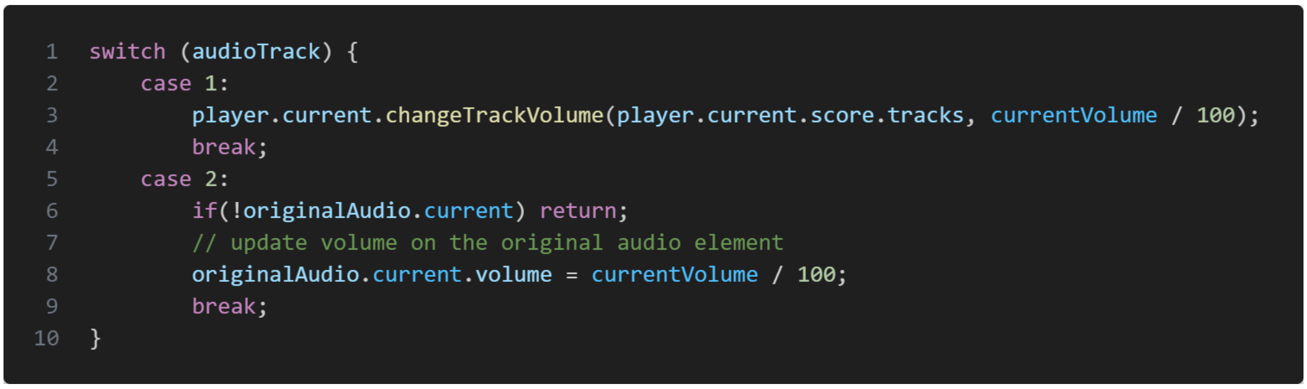


Figure 124 illustrated the chords output by the prediction model.

Although the accuracy of the notes and chords seemed poor, the sound and pitch of each chord/note was there. If there were more time to really refined the model, there would definitely step up the accuracy of the model. Additionally, more dataset of variation of guitar sounds such as normal electric guitar sound, or distorted guitar sound would tremendously increase the model's ability to distinguish the bleeding of sounds.

5.12.3 The final adjustment

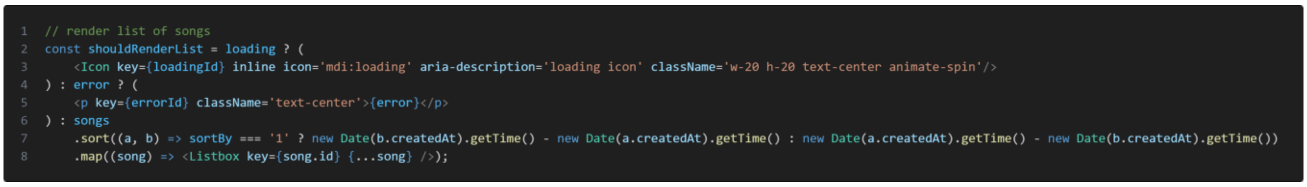
The final adjustment was made to fix some of the bug caused by the changing the volume of the audio, which caused the both MIDI and actual audio to played. This was easily fixed by checking the active audio player and mute the one was not in used as shown in Figure 125.



```
switch (audioTrack) {
    case 1:
        player.current.changeTrackVolume(player.current.score.tracks, currentVolume / 100);
        break;
    case 2:
        if(!originalAudio.current) return;
        // update volume on the original audio element
        originalAudio.current.volume = currentVolume / 100;
        break;
}
```

Figure 125 illustrated the switch case of the audio track.

The sorting of the song was not working previously as the sorting took in a callback to sort the values. It needed to take into account the format of the date object, which the author first did not implement. The Date instance could be easily created and passed into the argument of the callback as shown in Figure 126.



```
// render list of songs
const shouldRenderList = loading ? (
    <Icon key={loadingId} inline icon='mdi:loading' aria-description='loading icon' className='w-20 h-20 text-center animate-spin'/>
) : error ? (
    <p key={errorId} className='text-center'>{error}</p>
) : songs
    .sort((a, b) => sortBy === '1' ? new Date(b.createdAt).getTime() - new Date(a.createdAt).getTime() : new Date(a.createdAt).getTime() - new Date(b.createdAt).getTime())
    .map((song) => <Listbox key={song.id} {...song} />);
```

Figure 126 illustrated the sorting function.

6 Testing

6.1 Introduction

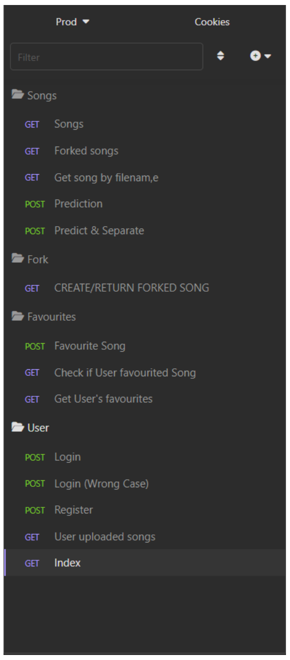
Testing served as a chapter that validate the final application via user inputs and general functional testing to see if there were any errors needed to be resolved.

6.2 Unit Testing

Functional testing was a method to test the software against the functional requirements. The application was tested by creating a checklist of tasks and record its actual output with the expected output. The results of functional testing could signify if certain section of the application was functional and working as intended.

6.2.1 Insomnia API Testing

When it came to testing API on the server-side application, Insomnia was utilised to ensure all API routes were working as intended and no error was thrown as the response. As shown in Figure 127 illustrated the available API routes that needed to be tested for possible errors and that the correct response was returned from the server. The API needed to be tested locally to avoid the issue of production debugging where the application needed to be re-deployed each time an error was discovered.



```
Prod ▾                Cookies
Filter
Songs
  GET   Songs
  GET   Forked songs
  GET   Get song by filenam,e
  POST  Prediction
  POST  Predict & Separate
Fork
  GET   CREATE/RETURN FORKED SONG
Favourites
  POST  Favourite Song
  GET   Check if User favourited Song
  GET   Get User's favourites
User
  POST  Login
  POST  Login (Wrong Case)
  POST  Register
  GET   User uploaded songs
  GET   Index
```

Figure 127 illustrated the API routes used for unit testing.

The following tables contain the resulting responses from each endpoint used for testing the application during development:

User

Test No.	Test Case	Input	Expected Output	Actual Output
1	Registration	Missing Input	Error showing missing inputs	Error showing missing inputs
2	Registration	User details	User details with success message	User details with success message
3	Login	Missing input	Error	Error
4	Login	User Credentials	User Info + Token	User Info + Token
5	User’s uploaded song(s)	Missing token	No Token Error	No Token Error
6	User’s uploaded song(s)	Token	Uploaded song(s)	Uploaded song(s)

Favourites

Test No.	Test Case	Input	Expected Output	Actual Output
1	Favouriting a song	Missing token	Error showing missing token	HTML error: list index out of range
2	Favouriting a song	Token	Successful message	Successful message
3	Check if user has favourited the song	Missing token	No token present error	No token present error
4	Check if user has favourited the song	Token	Boolean + response status	Boolean + response status
5	User’s favourited song(s)	Missing	No Token Error	No Token Error

		token		
6	User’s favourited song(s)	Token	Favourited song(s)	Favourited song(s)

Songs

Test No.	Test Case	Input	Expected Output	Actual Output
1	Get songs		List of songs	List of songs
2	Get song by filename	filename	Song	Song
3	Get song by filename	User ID + filename	Song	Song
4	Get prediction from song	No audio	Missing file error	Missing file error
5	Get prediction from song	audio	File location	File location
6	Get prediction from song with user credential	Audio + token	File location + owner	File location + owner
7	Separate song and get its prediction	No audio	Missing file error	Missing file error
8	Separate song and get its prediction	audio	File location	File location
9	Separate song and get its prediction with user credential	Audio + token	File location + owner	File location + owner

Forks

Test No.	Test Case	Input	Expected Output	Actual Output
1	Create a fork from song	Wrong song id	Invalid song id	HTML error: invalid id
2	Create a fork from song	Song id + no token	No token	No token
3	Create a fork from song	Song id + token	Song	Song

4	Get existing fork	Wrong song id	Invalid song id	HTML error: invalid id
5	Get existing fork	Original song id + no token	No token present	No token present
6	Get existing fork	Original song id + token	Forked song	Forked song
7	Update forked song	Song id + token	Song info	Song info
8	Delete forked song	Song id + token	Successful message	Successful message
9	Get forked song(s) from original song	Song id	List of forked songs	List of forked songs

In the event that there was more time to develop the API, the additional documentation made using Swagger would be effective to also test the API endpoints. This was due to Swagger being a user-friendly UI and UX that described each endpoint, while providing example input(s) and output(s) and allowed developers to test their endpoints through the visual UI.

6.3 User Testing

The user testing was performed after basic functionality of the application was fully implemented and working as intended. By doing user testing, the author was able to evaluate the user experience and whether the feature was easy for the user to utilise. To conduct the user testing, the author had created multiple tasks for the testers to follow, where the flow of the application would be tested alongside the AI prediction. The following list were the series of tasks set out to be completed by the applicant:

9. Create an account.
10. Upload an audio.
11. Find the uploaded song(s).
12. Favourite a song.
13. Edit a song.
14. Save the song.
15. Find existing forked song.
16. Logout from the site.

Although the applicant would be given the same instructions, some would be given an audio with a different way to retrieve the prediction, such as an audio that needed separation before prediction, and an audio with only guitar track. The test would aim to see the user's navigational flow to see if the application was difficult to navigate or needed changing.

6.3.1 User Testing Results

Due to a niche aspect of the application, there were a small amount of candidates selected to carried out the criteria, where the testing would be carried out on Discord as shown in Figure 128. Though the overall testing results were generally positive, as none of the candidates were having major problem with the navigational flow of the application.

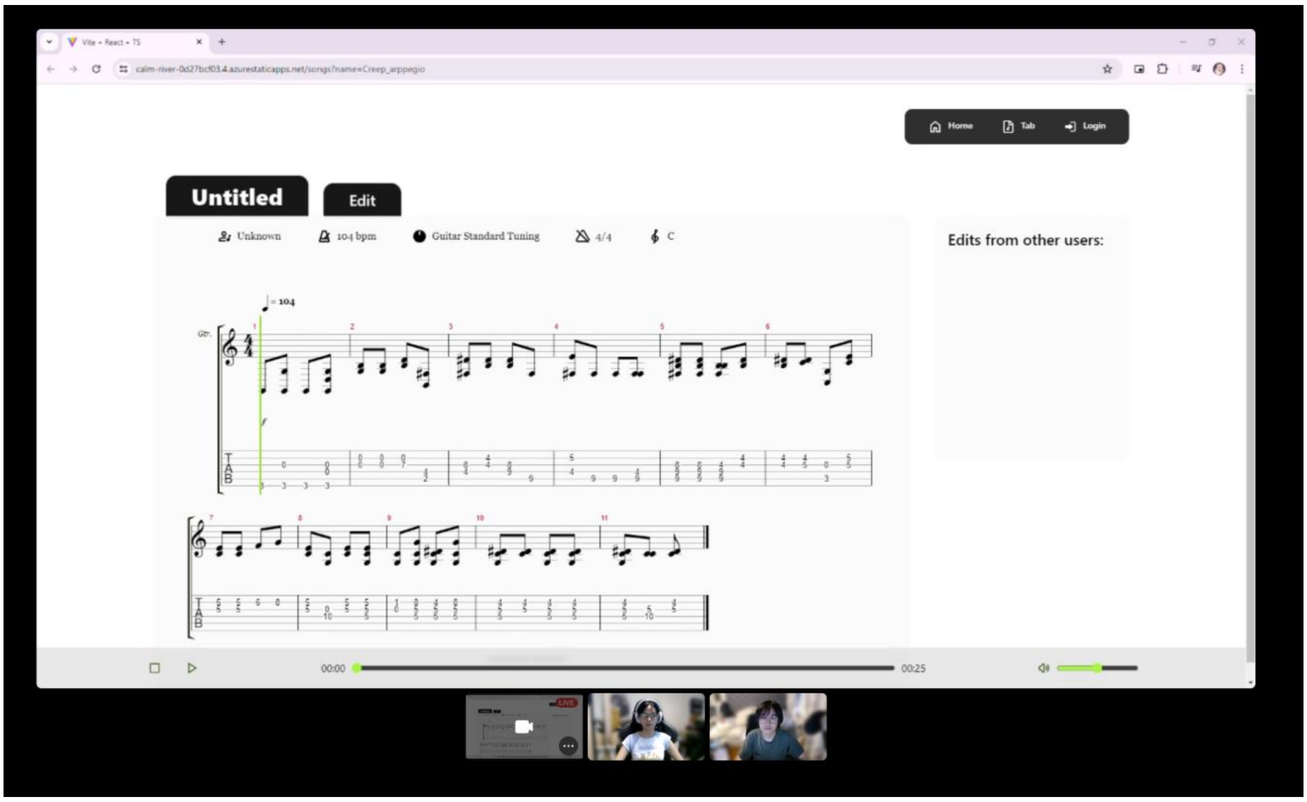


Figure 128 illustrated the process during user testing.

User 1

The first user was selected as they were currently studying a major in music studies, and due to their specialty in guitar. The user was familiar with the use of source separation application i.e. MVSEPs due to the work that they needed to do for their college. This meant that the flow of the existing application would not be a difficult task to perform.

Tasks

The user was given tasks as stated in the previous section with the caveat of using an audio of a full mixture, meaning the audio would consisted of more than a guitar track, to separate the guitar track. They were also required to create an account prior to uploading an audio, in order to favourite and edit songs before logging out from the application.

- **Uploading the audio** – though the user was able to upload the audio, they were unsure of the way to separate the mixture. It was until they dragged in the audio that they realised they needed to tick the box to separate the audio. It would be nice to later implement a some sort of help guide for the user to utilise the application.
- **Viewer** – the tab viewer was easy to utilise due to the user's familiarity with musical notation software i.e. MuseScore, and the experience in navigating around the UI was easy and struggle-free.

- **Editability** – the user said there was little difference between the edit and the original view page of the tablature. It would be worth to have a more clearer indication of the two pages. Additionally, the edit page was a little difficult to use and the ability to dynamically change beat types and other information was not present to be able to completely relied on the site to edit the tab.
- **Favouriting a song** – the ability to favourite a song was easy to spot as they could be seen from the header of each song. Thus, there were not much of a struggle to navigate and favourite the song.

User 2

The second user was picked due to their in-experience with musical instruments or software. The aims with the current candidate was to check the navigational flow and the overall feels of the site to see whether adjustments were needed to improve the current state of the application

Tasks

The user was given tasks to predict an audio that had only a guitar track. Once the user had been navigated to the viewer, the majority feature of the viewer would be tested to see their usage. Although, the user would be required to edit the tablature, the user would be asked to perform simple edit to the song as they were not familiar with the flow of the application.

- **Uploading the audio** – the user had easily navigated to the input to drop the audio as it was located in an easy-to-spot location.
- **Viewer** – It took some time before the user figured out that the viewer was navigable through the musical score, and not only through the audio player. Though after a while, the user was able to pick up the pace and figured out the functionalities of the tab viewer.
- **Editability** – the user had definitely experienced a learning curve in this section of the application as they did not have prior knowledge of any similar software that utilised the editability.
- **Favouriting a song** – due to numerous applications with the similar feature to favourite an item, they were pretty accustomed to how the feature worked.

6.4 Conclusion

To conclude, there were several tests conducted to provide an overview of the application to see what needed changing and debugging. Unit testing was studied to allow the application to be in its best state before deploying to the cloud, to ensure that it met the functional requirements and all of the actual outputs corresponded to the expected output. The user testing phrase then allowed the author to gain a better insight into the outsiders' perspectives to test the application's functionality and accessibility. The result of the tests was used to make necessary changes to the UI and UX, where bugs and major patches would be applied to the final application.

7 Project Management

The project has been managed through Notion with an addition of Kanban system to log the progress of the application and to check what the stage I am in terms of the project. The section of project management shows the phases of the project, going from the project idea through the requirements gathering, the specification for the project, the design, implementation, and testing phases for the project. It also discussed Notion as a tool which assist in project management and productivity.

7.1 Project Phases

7.1.1 Proposal

The project proposal was set to around the beginning of December in the previous year. Albeit the fast-paced schedule where ample time was given once the idea had been finalised, there were a couple of assignments and classes occurring concurrently and a meeting was held to help generate ideas of which would be interesting for the students and challenging to utilise unfamiliar technology. The initial idea was to develop a project revolving around the usage of AI such as a Large Language Model that could help diagnose early symptoms of possible diseases. However, the idea would require a ton of knowledges in the medical fields to gather information for different diseases and its respective symptoms. As a result of AI and Professional Development classes taken during the course of the college curriculum, which especially contributed to help the author realised an idea for an application that returned predictions of different chords and notes played in a song. As the problem would be interesting, and all the while tackling a MIR problem that could possibly save time to recreate guitar instrument from scratch. After some discussion with the project coordinator and the allocated supervisor, it was determined that it would be a very interesting project to work with and allowed the author also challenges to help acquire new skills and learn new technologies.

7.1.2 Requirements

Prior to the start of project sprints, extensive research needed to be written and some sort of available models needed to be sought to realise the capabilities of the project and that the final application could be based on for comparison and analysis. To fulfil the objectives of the project, the source separation was needed as a good measure of optionally separating guitar track from its mixture, the model architecture was required to transcribe guitar from audio, and the format of visualising the predictions i.e. as a music sheet or of equivalent.

The publishing of papers, books, and documentaries were thoroughly researched to find out the current trend of models and how far the AMT had traversed over the decade. Studies on a couple of possible existing applications were reviewed to inspire from and get an idea of the feasibility of the Thesis itself. And based on this research, functional and non-functional requirements were determined to set out the technologies for developing the application. Furthermore, the creation of user personas was incredibly helpful in providing additional information requirements to create the application.

7.1.3 Design

Upon the design section of the application, the early prototypes called ‘paper prototypes’ were developed as a way to visualise the appearance and functionality of each page, as well as the components that would need to be created for the application. This process only required rough estimates of each page that only feature simple boxes and rectangles of different feature with labels attached to it, and usually took less than ten minutes to sketch. This would be alluded to the development of low-fidelity prototypes where each page was refined with placeholder texts, layout adjustment, choosing tonality and saturation of the overall application. The prototypes would be brought to high-fidelity quality, where the layout of the UI and all the features would be colourised and further configured to match the final vision.

In terms of the design of the architecture, the application would be handled by a Frontend and a Backend, which take care of the tablature visualisation and the AMT model. These architectures would be developed separately and later merged together via the communication of the API that act as a middleman that returned messages to the Frontend. In the technical aspect of the design, consideration was given to how components would interact within the application, how the server would connect to the client, and how the server would connect to the database. It was necessary to determine what would be stored in the database and what was required to develop both the server and the database.

7.1.4 Implementation

The implementation started by developing the two different components needed for the overall applications. Development between the two components was managed so each component would get sufficient time for completion. A basic prototype was built for the Frontend application, where loose structure of the final application was formed to do initial testing of libraries and the looks and feels of the final application. On the other hand, the AI model was developed using Python, TensorFlow and the pre-existing called TabCNN as a starting model to train the dataset through a CNN, which allowed it to predict tablature from audio waveforms. The development of these applications was a great initial advancement that could be built upon.

The AI model that was trained locally on a local environment was working fine, but when it came to deployment a new method needed to put in place to take an audio and converted into predictions. There was a problem since the initial prototype only take in audio through a file in a folder, albeit the actual service needed to take an audio from an API request. Thus, there would be more research needed resolve this issue and publish the application as a live server.

Through iterations of architecture and libraries used to develop the applications, there were a lot of difficulties that arose from using AWS as a cloud provider. Some libraries caused the deployment of these applications to be difficult due to its dependency libraries that required additional packages to be installed. This resulted in the migration of cloud provider to Azure, where the problem resolved itself, but another issue arose, where the SQL Server the author was using required a driver that was mandatory for a library to connect to the database, and the integration needed to be scrapped and migrated to MongoDB. The development was finally eased up with the revised provider and much more progress was being made.

While the development of the Front-end application was progressing smoothly, there was not much progress being made with the AI model. Though the AI model was working fine, it did not perform as initially intended. Therefore, there were a lot of attempts to improve the accuracy of the model. But unfortunately, it was not possible with the time of the development, where each application needed to be implemented concurrently. The decision was made to add an ability to edit the prediction and another model was needed to predict the chord progression to display to the user and helped them navigate through the song easier.

Towards the end of the project's Sprint, the prototype of the application was advanced to state where the initial requirements could be met. The prototype contained the intended features of the initial plan that allowed users to transcribe audio into a guitar tablature, where it could validate against the user to extract their feedback.

7.1.5 Testing

The testing of the prototype was divided into two main sections – Unit testing and User testing. Unit testing provides the author the ability to validate each component and the back-end API against different scenarios from entering a wrong input, leaving a field blank to entering the right combinations. Although, the functionalities of each API routes were fully functional, it did not consider of the entirety of the application, where it would be intertwined with the components on the Front-end application. The entire application needed to be tested to find possible errors and to ensure that the application performed as intended.

Subsequently, the User testing would be conducted to brought in the views of the people who were outside of the development. Participants were selected to take part in completing a series of tasks and were reviewed for feedback on each of the tasks. These tasks were designed to gauge the difficulties in operating the application and to see if there were any changes needed to make or improve upon. As expected, there were some issues with the UIs and the missing features such as some bugs that occurred during testing where the sorting ability was not working, both audio from the MIDI and actual audio was playing when the user changed the volume, and lastly, the . These issues were remedied and resolved during the last Sprint.

7.2 Scrum Methodologies

SCRUM was a genre of framework which allow various teams and organisation to gravely produce prototypes through adaptive solutions arise from compound problems as shown in Figure 128.

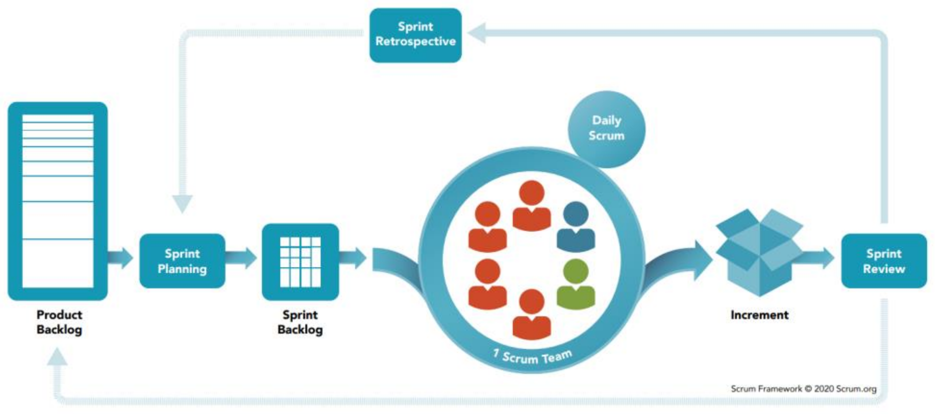


Figure 129 illustrated the diagram of scrum iteration cycle.

In order for the team to work efficiently, minimise the need for meetings and keep improving the end product, the Scrum events have implemented. These events are time-boxed where each duration has been fixed before the Sprint session started. A session may end whenever the purpose of the event has been accomplished. This process is done to ensure that no time is wasted in the process. These events are as followed:

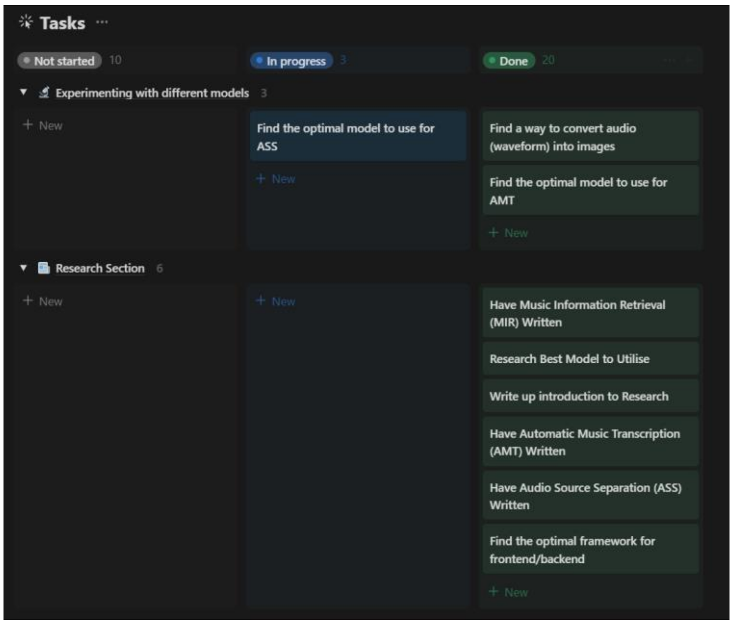
- Sprints –the time fixed in length of working to create consistency to the team.
- Sprint Planning –a prescribed goal before starting a session such as why is this Sprint valuable? What can be done this Sprint? And how will the chosen work get completed? These requirements would be increment into the Project Backlogs.
- Daily Scrum – the checking of the progress made toward the Sprint goal. The inspection then could be put in the Sprint Backlog.
- Sprint Review – to audit the outcome of the Sprint to decide whether to end the Sprint session.
- Sprint Retrospective – purpose to create ways to improve quality and effectiveness by the inspection of the previous session(s).

7.2.1 Project Management Tools

Notion

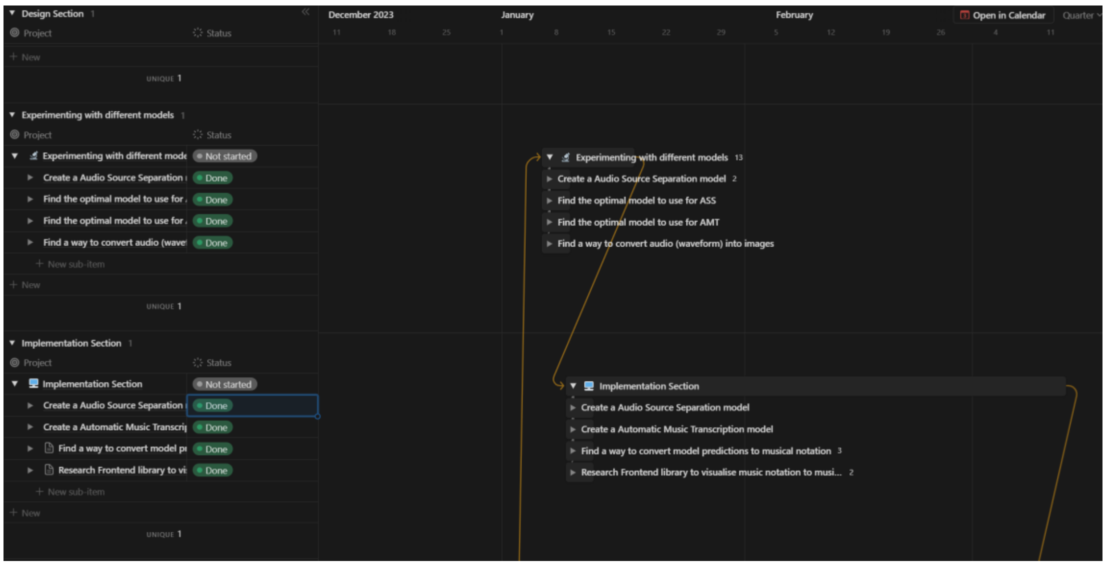
Notion allowed developers to keep track of their progresses and record them using a built-in Kanban system that let the developers arrange where they were in terms of the project development, arranging and categorising their progress in three columns as shown in

Figure 129: To-Do, Work-In-Progress, and Done. The Figure 130 also illustrated the Gantt chart that was utilised to keep track of the duration of each implementations.


Tasks

Not started 10	In progress 3	Done 20
Experimenting with different models 3		
+ New	Find the optimal model to use for ASS	Find a way to convert audio (waveform) into images
	+ New	Find the optimal model to use for AMT
		+ New
Research Section 6		
+ New	+ New	Have Music Information Retrieval (MIR) Written
		Research Best Model to Utilise
		Write up introduction to Research
		Have Automatic Music Transcription (AMT) Written
		Have Audio Source Separation (ASS) Written
		Find the optimal framework for frontend/backend
		+ New

Figure 130 shows the layout of a Kanban system in Notion.



Design Section 1

Project	Status
+ New	
UNIQUE 1	

Experimenting with different models 1

Project	Status
Experimenting with different mode	Not started
Create a Audio Source Separation	Done
Find the optimal model to use for	Done
Find the optimal model to use for	Done
Find a way to convert audio (wave	Done
+ New sub-item	
+ New	
UNIQUE 1	

Implementation Section 1

Project	Status
Implementation Section	Not started
Create a Audio Source Separation	Done
Create a Automatic Music Transcrip	Done
Find a way to convert model pr	Done
Research Frontend library to vi	Done
+ New sub-item	
+ New	
UNIQUE 1	

Experimenting with different models 13
- Create a Audio Source Separation model 2
- Find the optimal model to use for ASS
- Find the optimal model to use for AMT
- Find a way to convert audio (waveform) into images

Implementation Section
- Create a Audio Source Separation model
- Create a Automatic Music Transcription model
- Find a way to convert model predictions to musical notation 3
- Research Frontend library to visualise music notation to musi... 2

Figure 131 illustrated the Gantt chart.

7.3 Reflection

7.3.1 Your Views on the Project

It was undoubtably one of the complex projects in terms of micro-managing research, different Sprints, and fulfilling the initial requirements of the project. The development of the Front-end application was considerably satisfactory, where the majority of the Front-end skills were achieved through the years of college works in terms of the implementation of React, NodeJS, and Typescript. In addition, the learning curves of implementing the Back end with AI was quite challenging, the ability to be able to pull it off was also particularly rewarding. Being able to apply the low-level technologies and knowledge from the modules learned during the course of the study and further applied high-level knowledge to develop an AI model would undoubtedly became useful in future projects. Lastly, the exploration of AI, ML, and DevOps had allowed for a better relevancy in the current trend in the industry. As the overall project did not just complement the technical skills in programming and problem-solving, but also allowed for the valuable experiences of multi-tasking, time managing, and adaptability skills to be improved upon.

7.3.2 Completion of large software development

Though, the development of such a large-scale software had been completed by the author in the past through internship and college works. The majority of these past projects were developed through a lot of frameworks and technologies very familiar by the author. As a result, having to manage each aspect of the project while learning a whole new framework was entirely new. For instance, implementing an add-on feature that was entirely from scratch, and research different ways to develop the ML models to achieve a better accuracy result were proven to be really challenging, and a lot of adaptability were required in order to see the project through to the end. Therefore, the Gantt chart (via Notion) was put in place to keep track of the progress and allowed the author to outline functional requirements that needed to be completed first. The project had been kept tightly scheduled to make the author manage the time more efficiently, where the use of a Gantt chart template in Notion allowed each task to be timed and contributed to tracking the progress of each task. This helped the author anticipate different Sprints, whether future plans needed to be adjusted and/or re-established more proficiently.

7.3.3 Working with a supervisor

The collaboration with supervisor during the development of the project was definitely served as a great deal with steering the author in the right direction. The arrangement to attend a meeting once a week with the supervisor to discuss future plans, progresses, and struggles really helped in exposing different perspectives on the working application.

Especially at the end of the development, where the application did not managed to achieve the best possible solutions, having a supervisor had helped the author in recognising the tasks at hand, instead of worrying about future implementations and what could not be achieved any further. Even though, the supervisor was not familiar with the technologies, they would often offered different opinions and views on the potential of the application as well as the development process. To conclude, the presence of a supervisor contributed to the progress of tasks during each Sprint and also served as an encouragement to explore different methodologies to accomplish the objectives in each Sprint of the development.

7.3.4 Technical Skills

Through the development of a full-stack application, where the application was not of a monolith architecture had significantly widen the author's technical skills. The creation of a back-end architecture using Python to handle different types of functionalities in the Back end was an entirely new experience. This was due to the lack of experience in developing and using Python for software development. Though now, the author felt more comfortable with integrating and developing not just another Python application but also a new application via other languages and/or frameworks. Furthermore, the integration of Front-end and Back-end architecture of different languages had encourage the exploration of different frameworks, which would definitely be advantageous in advancing the author's own skills. Finally, the accumulation of different skills and experiences would undoubtedly contribute to achieving the author's goal of becoming a full-stack developer.

7.3.5 Conclusion

To conclude, the development of such a scale of a project, where there was high level in complexity, it was crucial to put in place a project management plan to be timely such a convoluted topic. This chapter served as a reflection on different aspects of the project, in which the author would not only look back at the work, but also appreciate the help that allowed the author to bring the project to a conclusion. Furthermore, the reflections on the skills and experiences that had been achieved since the start of the project to the encouragement of supervisor and second reader that pushed the author to overcame various challenges and produced a functional prototype that met its intended requirements.

8 Conclusion

The project intended to develop an AI application that the guitarists and music students could utilised to quickly turn an audio of a given song into a guitar tablature. The application aimed to eliminate the long-winded process of creating a musical score/MIDI from scratch and help students to practice the song that may not be as well-known compared to the modern pop music. The findings of the project was evident in the application's ability to take an audio from a mixture or a guitar track and output predictions based on the sound waves and display the tablature on to the screen. Furthermore, the application also included an additional features such as the ability to copied the song for edits, and favourite the song for later practices, despite the existence of bugs in the current state, which showcase the possibility of its implementation.

8.1 Summary of Chapters

The research chapter really served as a crucial introduction for both the author and the overall project. It definitely contributed to the overall architecture of the application, where the investigation of different functionalities and technologies were explored to determine the feasibility of the project. It focused on the practicality of each technologies, whether they could be combined to develop an application that could surpass the current technology. The study of ML, DL, and audio processing was undoubtably crucial due to its ability to provide the important information that could be built upon to create an application that could convert an audio waveform into guitar tablature. Having the sense of the generality of these technologies would helped the author navigate the waters to find the way to visualise what worked and what needed to be substitute to make the application performed as intended.

Once the study of research had been completed, the requirements chapter would be put in place to explore existing applications and identify the current trend in the field of MIR, as well as to analyse the audience for the application. The analysis of different competitors allowed the author to gain an insight into the features offered by other applications to provide the target audience a better alternative to these established competitors. Moreover, a better insight into these preferences and requirements would tremendously contributed to determining the functional requirements of the application, as well as, incorporating them into the design of the application.

The design chapter were heavily influenced by existing applications and the requirements of the application. The chapter helped guide the overall architecture of the application, where the design patterns were incorporated into the application to put structure to the components and the architecture. In addition, the development of the application architecture and flow chart provided a visual representation of the interaction between the user and the application itself. The prototyping of different quality wireframes allowed the author to visualise what was needed to fulfil the functional requirements set forth in the requirements chapter.

Consecutively, the implementation would be carried out according to the structure set forth in the previous chapter. The tasks would be implemented in a bi-weekly manner, where each Sprint would aimed to develop the tablature front-end and AI back-end as a microservice architecture. This method would be proven to be quite effective, as the code repository could be kept neat and easy to maintain. Furthermore, the SCRUM methodology had allowed the author to monitor these micro tasks and address obstacles arose from the development of the application.

Finally, the unit testing and user testing could be conducted as a way to gauge the readiness of the application and review the feedback of the users. The unit testing of the application would be conducted throughout the process of the development. This was done to validate the expected outcomes against the actual outcomes to test the different types of error handling to prevent application from crashing the server. The user testing would commenced after the unit testing, where these users would be selected to perform a list of tasks to test all the functionalities of the application. For every task completed, the user would be given an opportunity to provide feedback on their experience using the application and their overall impression of the site. The feedback on the functionality was rather positive and a suggestion of new features were recommended to improve the application in terms of the UX and UI. These issues were addressed and resolved accordingly.

8.2 Future Improvements

Albeit the successful of the application, there were a lot of features that could be added to further enhance the application. The prediction would output chords and notes that bled from previous sound. The onsets detection could be further implemented to detect better timings, and an algorithm needed to be put in place to plot the note/chord at a better position to match the timing of the audio. A deeper dive into the audio augmentation and time to invest into the model could possibly enable the model to achieve higher accuracy. Additionally, while the edit ability was implemented the ability to add different beat type to tell how fast the note was being played could be completed to further expand the utility of the application, and offer the users the ability to fully edit the tablature and compose music online, without the need to download the tab and open a software to edit it.

Another potential feature that could also improve the learning experience of a user was the ability to use URLs to transcribe song from other streaming platforms. This feature would allow the users to quickly paste the URL from their desired platform and have it quickly transcribe to the user without the need to download it separately before feeding it into the application. Through all these possible improvements, it would definitely widen the scope and complexity of the application and leverage the user experience in surfing the site.

8.3 Personal Takeaways

The completion of the project had definitely been an accomplishment to venture into the different architectures and technologies that the author was not very familiar with. The project had accomplished its goals in developing a AI-assisted application that could quickly transform an audio into a guitar tab through the use of a micro-service architecture. The project provided a new opportunity to learn new technologies such as AlphaTab, Demucs, MIR, and Flask. These technologies were proven really handy for future projects due to its vast insight into AI, MIR, and its integration to the application, where its relevancy would be useful for future prospects.

The project had taught a variety of valuable aspects such as the importance of having a management plan to sub-divided complex topics into micro-tasks, in order to handle a large-scale application. Furthermore, the project outlined a crucial skill of problem-solving, where the adaptability to find new approaches to an idea that did not come to fruition could be significant to manage the time efficiently.

9 References

Benetos, E., Dixon, S., Duan, Z., & Ewert, S. (2019). *Automatic Music Transcription: An Overview.*

Bengio, Y., Goodfellow, I., & Courville, A. (2017). *Deep learning (Vol. 1).* MA, USA: MIT press.

Britz, D. (2015, 11 07). *Understanding Convolutional Neural Networks for NLP.* Retrieved from Denny Britz: https://dennybritz.com/posts/wildml/understanding-convolutional-neural-networks-for-nlp/

D., P., & T., T. (2017). *What is MIDI?* Tufts University.

Das, S. (2023, 03 14). *Top 10 Python Web Development Frameworks in 2023*. Retrieved from browserstack: https://www.browserstack.com/guide/top-python-web-development-frameworks

Digitization. (n.d.). Retrieved from Digital Sound and Music: https://digitalsoundandmusic.com/5-1-2-digitization/

Downie, J. (2003). Music Information Retrieval. *Annual Review of Information Science and Technology* (pp. 295-340). University of Illinois at Urbana-Champaign.

Editors, A. G. (2013, 04 25). *Acoustic Guitar Notation Guide*. Retrieved from acousticguitar: https://acousticguitar.com/acoustic-guitar-notation-guide/

fheyen. (2023, 10 31). *vite build does not work with alphaTab's worker #1289.* Retrieved from GitHub: https://github.com/CoderLine/alphaTab/issues/1289

Francisco, J. (2023, 08 02). *The Developer's Guide to Speech Recognition in Python*. Retrieved from deepgram: https://deepgram.com/learn/best-python-audio-libraries-for-speech-recognition-in-2023

Ghost, & Daneilku15. (2016, 09 3). *Adding/removing music notes dynamically? #105*. Retrieved from GitHub: https://github.com/CoderLine/alphaTab/issues/105

Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., . . . Chen, T. (2017). *Recent Advances in Convolutional Neural Networks.* Nanyang: Nanyang Technological University.

Guitar Sales Statistics (US National Survey 2024). (2023, 12 31). Retrieved from Piano Dreamers: https://www.pianodreamers.com/guitar-sales-statistics/

Guzman, J. d. (2013, 06 17). *Cycfi Six Pack Hexaphonic Pickup v1.1.* Retrieved from Cycfi: https://www.cycfi.com/2013/06/cycfi-pack-hexaphonic-pickup-v1-1/

III, J. P., Mallari, J., & Pelayo, J. (2015). *Guitar as the Preferred Musical Instrument.*

Inskip, C. (2011). *Music Information Retrieval Research.*

Jadhav, Y., Patel, A., Jhaveri, R., & Raut, R. (2022). Transfer Learning for Audio Waveform to Guitar Chord Spectrograms Using the Convolution Neural Network. *Hindawi*.

Kasak, P., & Jarina, R. (2022). Evaluation of blind source separation algorithms applied to music signals. (pp. 1-8). Zilina: Department of multimedia and information-communication technology.

Kasak, P., Jarina, R., & Chmulik, M. (2020). Music Infomation Retrieval For Educational Purposes. *Department of multimedia and information-communication technologies* (pp. 296 - 301). Zilina, Slovakia: University of Zilina.

Kasak, P., Jarina, R., & Chmulik, M. (2020). Music information retrieval for educational purposes - an overview. *18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, (pp. 296-304).

Kasak, P., Jarina, R., & Chmulik, M. (2022). Music Infomation Retrieval For Educational Purposes. *Department of multimedia and information-communication technologies* (pp. 296 - 301). Zilina, Slovakia: University of Zilina.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). *Deep learning.*

Lidy, T., & Rauber, A. (2009). *Chapter XLVI Music Information Retrieval.* Vienna: Vienna University of Technology.

Liem, C., Müller, M., Eck, D., Tzanetakis, G., & Hanjalic, A. (2011). The need for music information retrieval with user-centered and multimodal strategies. *1st international ACM workshop*, (pp. 1-6).

Manilow, E., Seetharaman, P., & Salamon, J. (2020). *source-separation*. Retrieved from Open Source Tools & Data for Music Source Separation: https://source-separation.github.io/tutorial/landing.html matevzsenlab. (2023, 01 16). *Node JS and React problems #1098.* Retrieved from GitHub: https://github.com/CoderLine/alphaTab/discussions/1098

Mendes, A., & Rodrigues, O. (2023, 11 02). *Top 10 best front end frameworks in 2024*. Retrieved from imaginarycloud: https://www.imaginarycloud.com/blog/best-frontend-frameworks/

Müller, M. (2007). *Information Retrieval for Music and Motion.* Berlin: Springer.

Open Source Tools & Data for Music Source Separation. (n.d.). Retrieved from Representing Audio: https://source-separation.github.io/

O'Shea, K., & Nash, R. (2015). An Introduction to Convolutional Neural Networks., (pp. 1-10).

Parisi, G., Barros, P., Jirak, D., & Wermter, S. (2015, 02). *Real-time gesture recognition using a humanoid robot with a deep neural architecture.* Retrieved from Research Gate: https://www.researchgate.net/figure/Illustration-of-the-output-of-each-pair-of-convolution-and-max-pooling-operations-This_fig2_283646155

Payne, R. (2023, 09 22). *7 Best Python Libraries for Machine Learning and AI*. Retrieved from developer.com: https://www.developer.com/languages/python/python-libraries-for-machine-learning-ai/

Podoba, V. (2023, 09 14). *Best Python Frameworks for Web Development in 2023*. Retrieved from softformance: https://www.softformance.com/blog/python-web-frameworks/

Randel, D. M. (1986). *The new Harvard dictionary of music.* Cambridge, MA: Belknap Press.

Rangarao, P. (2021, 4 5). *Audio Processing Libraries in Python*. Retrieved from Medium: https://prabhakar-rangarao.medium.com/audio-processing-libraries-in-python-b7ee0e5690d9

Rathi, S. (2019, 08 09). *Top 8 Python Libraries for Machine Learning & Artificial Intelligence*. Retrieved from hackernoon: https://hackernoon.com/top-8-python-libraries-for-machine-learning-and-artificial-intelligence-y08id3031 react-dropzone. (n.d.). *React-Dropzone*. Retrieved from React-Dropzone: https://react-dropzone.js.org/

Stoller, D., Ewert, S., & Dixon, S. (2018). WAVE-U-NET: A MULTI-SCALE NEURAL NETWORK FOR. *19th International Society for Music Information Retrieval Conference (ISMIR 2018)*, (pp. 1-6).

Sturm, B. L., Santos, J. F., Ben-Tal, O., & Korshunova, I. (2016). *Music transcription modelling and composition using deep learning.*

Sugandhi, A. (2023, 09 05). *Best Front end Frameworks for Web Development*. Retrieved from knowledgehut: https://www.knowledgehut.com/blog/web-development/front-end-development-frameworks

Takahashi, N., Goswami, N., & Mitsufuji, Y. (2018). MMDENSELSTM: AN EFFICIENT COMBINATION OF CONVOLUTIONAL AND. (pp. 1-4). Sony Corporation.

Takahashi, N., Goswami, N., & Mitsufuji, Y. (2018). MMDENSELSTM: AN EFFICIENT COMBINATION OF CONVOLUTIONAL AND RECURRENT NEURAL NETWORKS FOR AUDIO SOURCE SEPARATION., (pp. 1-4).

Tio, D. (2019, 06 21). *Automated Guitar Transcription with Deep Learning*. Retrieved from Towards Data Science: https://towardsdatascience.com/audio-to-guitar-tab-with-deep-learning-d76e12717f81

University, L. M. (n.d.). *Perceptual attributes of acoustic waves*. Retrieved from Acousticslab: https://acousticslab.org/RECA220/PMFiles/Module05.htm

University, L. M. (n.d.). *Perceptual attributes of acoustic waves*. Retrieved from Acousticslab: https://acousticslab.org/RECA220/PMFiles/Module05.htm

yvesonline. (2021, 02 22). *How do I add librosa (python lib) to aws lambda using serverless*. Retrieved from StackOverFlow: https://stackoverflow.com/questions/66309527/how-do-i-add-librosa-python-lib-to-aws-lambda-using-serverless

10 Appendices

Front-end application: https://calm-river-0d27bcf03.4.azurestaticapps.net

Back-end application: https://backendcontainer.orangebeach-0681140a.westeurope.azurecontainerapps.io

Prototypes created via Figma: https://www.figma.com/file/QBcHKyeJAZgcUFkRack5kn/Lofi-Wireframes-v1?type=design&node-id=0%3A1&mode=design&t=7qXUdXWeZlcsxUXB-1

Kanban and Gantt Chart on Notion: https://tako-niku.notion.site/Gantt-chart-7637304cf47e4332904efb459af72755?pvs=4

Unit-testing: https://iadt-my.sharepoint.com/:u:/g/personal/n00201327_iadt_ie/ESd3Y8mvOXJPhc6Gz8Vd65MBU9YGveiga_QyYQHiddPoXw?e=PxetQY