

BSc (Hons) in Creative Computing

Pedagogy of Music Theory and Improvisation through mobile applications

Author: Eoan O'Dea | N00162393

Supervisor: Mohammed Cherbatji

Declaration of Authorship

I hereby certify that the material, which I now submit for assessment is entirely my work and has not been taken from the work of others except to the extent of such work which has been cited and acknowledged within the text of my work.

Declaration

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

Signed: 

Date: 08/05/2021

Abstract

This document contains research in the area of Music Theory and Improvisation to determine if it can be taught through the use of mobile applications, using gamified methods of education.

The idea behind this project grew from interviewing a variety of musicians, and how they respected their instruments. Particularly comparing Pianists and Guitarists. A Pianist would often spend years learning music theory, including chords, scales, chord progressions etc. alongside learning pieces. After learning this, they usually can only play sheet music after a large amount of practice. This was noticed to be the opposite for many Guitar players, who would usually know very little about the subject of music theory, but can often improvise at a very strong level.

The project aims to bridge the gap between these two musicians, understand the gap within their knowledge and introduce the user to the dense subject of Music Theory in small, consumable pieces of information, which they can learn from by using the application consistently.

Acknowledgements

I would like to thank my supervisor, Mohammed Cherbatji. I could not have asked for a better supervisor for this project. He offered dedication, attention to detail and incredible guidance throughout. This project would not be half of what it is now if it wasn't for his invaluable experience. Working with him was an absolute pleasure.

This project would be nowhere without the large array of musicians who helped me identify and understand the presented problem, and to test the application along the way. This includes: Aoibhinn O'Dea, Clare Mohan, Christoph Busse, Johannes Hofmann, Annika Schewitz, Theresa Kolb, Maccon Keane, Daniel Henshaw and many more.

I'd also like to thank Clare Appleby for guiding me along my journey of education, and my fellow classmates, who made the whole year of remote learning much easier.

Lastly, I would like to thank my family, and in particular, my father, Paul O'Dea, for his constant support not only throughout this thesis but throughout my four years at IADT, which would not have been possible without him.

Table of Contents

1 Introduction	9
2 Research	9
2.1 Introduction	9
2.2 Problem Identification	10
2.2.1 Problem 1 - Fundamentals of Music Theory	10
2.2.1.1 Stave	11
2.2.1.2 Clefs	11
2.2.1.3 Types of Notes	12
2.2.1.4 Time Signature	13
2.2.1.5 Reading notes on a Piano	14
2.2.1.5 Reading notes on a Stave	14
2.2.1.5.1 Treble Clef	14
2.2.1.5.2 Bass Clef	15
2.2.2 Problem 2 - Teaching a Key	15
2.2.2.1 Scales and Chords	16
2.2.3 Problem 3 - Improvising	17
2.2.3.1 Chord Progressions	17
Progression 1	18
Progression 2	18
2.2.3.2 Licks	18
2.2.3.3 The Dimensions of Improvisation	19
2.2.3.3.1 Anticipation	19
2.2.3.3.2 Use of Repertoire	19
2.2.3.3.3 Emotive Communication	20
2.2.3.3.4 Feedback	20
2.2.3.3.5 Flow	20
2.3 Conclusion	21
3 Requirements	22
3.1 Introduction	22
3.2 Requirements Analysis	22
3.2.1 Existing applications	22
3.2.1.1 Duolingo (iOS, Android & Web)	23
3.2.1.2 Piano by Yousician (iOS, Android)	24
3.2.2 User Profile	26
3.2.2.1 Personas	27
3.2.2.2 Survey	27

3.2.3 Requirement Modelling	31
3.2.3.1 Requirements	32
3.2.3.1.1 Functional Requirements	32
3.2.3.1.2 Non-Functional Requirement	32
3.2.3.2 Use Case Diagram	33
3.3 System Model and System Requirements	33
3.3.1 Possible Technologies	34
3.3.1.1 MERN Stack	34
3.3.1.3.1 Advantages	34
3.3.1.3.2 Disadvantages	34
3.3.1.2 Laravel, Vue.js & MySQL	35
3.3.2.1.1 Advantages	35
3.3.2.1.2 Disadvantages	35
3.3.1.3 Flutter, GraphQL, Express.js, Node.js & MongoDB	35
3.3.2.3.1 Advantages	36
3.3.2.3.2 Disadvantages	36
3.3.2 Conclusion	36
3.4 Feasibility	37
3.5 Project Plan	37
3.6 Test Plan	38
3.6.1 Unit Testing	39
3.6.2 User Testing	39
3.6.3 System Testing	39
3.6.4 Integration Testing	39
4 Design	40
4.1 Introduction	40
4.2 Program Design	40
4.2.1 Technologies	40
4.2.2 Structure of the Technology Stack	43
4.2.3 Design Patterns	45
4.2.3 Application architecture	46
4.2.4 Database design	47
4.2.5 Data Design	48
4.3 User interface design	49
3.3.1 Wireframe	49
3.3.2 User Flow Diagram	52
3.3.3 Style guide	53
3.3.4.1 Colour	53
3.3.4.2 Typography	55

3.3.4.3 Shape	55
3.3.4.4 Sound	56
3.3.4 Environment	56
3.3.5 HiFi Prototype	57
4.4 Content Design	60
4.4.1 Music Theory Foundation Content	60
4.4.2 Music Theory Key Content	61
4.4.3 Improvising	62
4.4 Conclusion	63
5 Implementation	64
5.1 Introduction	64
5.2 Development Environment	64
5.2.1 Dart	64
5.2.2 Flutter	65
5.2.3 GraphQL for VSCode	65
5.3 Database	65
5.4 Backend	67
4.4.1 Serving Static Assets	68
4.4.2 Structure	68
4.4.3 Validation	71
4.4.4 Resolvers	71
4.4.5 Entities	73
4.4.6 Mikro ORM	75
4.4.7 GraphQL	77
4.4.8 Deployment	77
5.5 Frontend	79
5.5.1 Structure	80
5.5.2 Authentication & Routing	80
5.5.3 Models	83
5.5.4 GraphQL	84
5.5.4.1 Querying data from the server	84
5.5.4.2 Running Mutations to the server	87
5.5.5 Services	87
5.5.5 Lesson	88
5.5.6 Deployment	91
5.6 Admin Dashboard	92
5.7 Conclusion	95
6 Testing	97

6.1 Usability Testing	97
6.1.1 User Test Prerequisite	97
6.1.2 Tasks	98
6.1.3 Results	98
6.2 Unit / Integration Testing	99
6.2.1 Unit Testing	99
6.2.2 Manual Integration Testing	100
6.3 Performance Testing	100
6.3.1 Server Testing	100
6.3.2 Mobile Application testing	101
6.3.2.1 App Launch testing	101
6.3.2.2 CPU & Memory Testing	102
6.3.2.3 Memory Leak Testing	102
6.3.2.4 Broad Device Testing	103
6.3.3 Admin Dashboard testing	103
6.4 Conclusion	106
7 Conclusion	108
7.1 Project Management	108
7.2 Future Development	110
7.5 Learning Outcomes	111
7.6 Project Summary	111
7.7 Final Words	112
References	112

1 Introduction

Music theory is an extraordinarily dense subject. It can take people years to learn and understand all of its details, and more often than not, people don't understand the benefits of learning it.

This is often the case for many pianists, who learn music in a very traditional way, they often learn to read sheet music alongside learning scales, chords, key signatures without ever applying this knowledge to anything, they usually know a specific set of pieces they spent a long time learning by heart. This can be the opposite when it comes to guitarists, who would normally know very little about the subject of music theory, but can still improve at a very strong level.

This document aims to understand the gap between these two different musicians, and not only teach a user the fundamentals, but show them the reason why learning it would benefit them as a musician of any level.

This document is broken up into different chapters which focus on various steps that were taken to complete this project: 2. Research - Discusses the fundamentals of music theory, improvisation, and what is required to teach it. 3. Requirements - analysing the problem, understanding the requirements and deciding on the technology stack. 4. Design - resolves problems defined in the previous two chapters by creating the program design, user interface design & content design of the application. 5. Implementation - The process of implementing the application using the technologies mentioned in the requirements chapter. 6. Testing - Testing the application and its various components, both together and individually. 7. Conclusion - How the project was managed, Future development and final words about the project.

2 Research

2.1 Introduction

Playing an instrument is a difficult skill to develop. Depending on the instrument and your musical level, it usually involves countless years of practice before you can play a piece, or song, proficiently.

This is particularly true with someone who can play the piano. By the time they obtain a strong level, they can read sheet music, play a few pieces, and are familiar with a collection of scales and chords.

A variety of pianists, along with other musicians were interviewed to gain insight for this chapter. Something consistent amongst these interviewees was that they can read sheet music, have good knowledge of music theory, and after a lot of practice, can play a particular piece, but they cannot improvise. They are given all the necessary tools to improvise, but they are never trained to use these tools in the right way.

It takes months of preparation for a particular piece, and a pianist can usually never come up with their music instantly. This isn't necessarily their fault, it's the way they were taught music, to begin with. A similar example of this method of pedagogy is the 2nd level education system. A teacher will hand the student a large book and will tell them to memorise it. Regardless of the fact that the student understands it, if they can regurgitate it for an exam, the teacher has done their job.

This is slightly different when it comes to an (intermediate to advanced) guitar player. A guitarist can play, and usually, can improvise. They manage to learn this from experimentation. Despite not being taught music theory, scales, key signatures, sight-reading or anything else that a piano student needs to know by heart, they can improvise, and they can usually do it quite well.

This is what this research aims to solve, understanding how someone with no theoretical knowledge can improvise, and how somebody with all the theoretical knowledge can't. It is still important to know this information, particularly if you want to learn how to improvise, but we want to take the elements from how each side is taught, and subsequently understand musical pedagogy.

2.2 Problem Identification

This section discusses the problems that this research aims to address. The following problems will need to be researched in order to gain a better understanding on how the application should work and what it needs to accomplish.

- Problem 1 - Fundamentals of Music Theory
- Problem 2 - Teaching a key
- Problem 3 - Improvising

2.2.1 Problem 1 - Fundamentals of Music Theory

The first problem is music theory pedagogy. As mentioned in the introduction, many piano students are taught the fundamentals of music theory as they learn the piano, including scales, chords, sight-reading, dictation, key signatures, etc.

They have been given the necessary tools, but they are never taught how to correctly use them. This is exactly where the problem occurs. This report aims to understand why

this is the case, and how to resolve it to give the student the tools and knowhow to improvise.

Learning music theory has been described as being like learning a new language. There are rules, conventions, good practices and a general flow. (Biasutti, 2017)

If the above statement is true, this would mean a good way to learn music theory, would be to approach it in small bite-size chunks, while consistently giving the student different activities to learn.

Before the student can approach the very first key, we need to ensure they understand a few fundamentals, which includes recognising a staff, the difference between a treble and bass clef, understanding the type of notes and their values, what a time signature is, and finally reading notes on a staff.

2.2.1.1 Stave

A stave is a collection of five lines and four spaces where the notes will be placed to distinguish its pitch.

2.2.1.2 Clefs

A Clef distinguishes the left hand (Bass Clef) from the right hand (Treble clef) they are split by the note in the middle of the piano, usually referred to as “middle C”.



Fig 2.2.1 - A treble and bass clef on two staves combined

The student needs to understand what these symbols represent, as it usually tells them which hand to play with. For the most part notes on the top stave (Treble Clef), are

usually played with the right hand and vice-versa with the left hand for the bottom stave (Bass Clef).

2.2.1.3 Types of Notes

It is also important for the student to understand the type of notes they may encounter on a stave. This is an essential part of reading and understanding, as it explains how long the note should be played for.

Note	Name	Value
 <i>Fig 2.2.2</i>	Semibreve	1 - Whole note
 <i>Fig 2.2.3</i>	Minim	$\frac{1}{2}$ - Half Note
 <i>Fig 2.2.4</i>	Crotchet	$\frac{1}{4}$ - Quarter Note
 <i>Fig 2.2.5</i>	Quaver	$\frac{1}{8}$ - Eight Note

(Fig 2.2.2 - 2.2.5 sourced from Farrant, 2020)

It should be noted there are more notes, including Semiquavers, Demisemiquavers and Hemidemisemiquavers, but these are more advanced and not necessary at this stage.

2.2.1.4 Time Signature

A Time Signature is a sign that indicates the number of beats within a single bar and the note value that receives one beat. Time signatures can be set in simple or compound time.

Time Signature	2/2	3/4	4/4	6/8	C (Common Time)
Beats Per Bar	Two beats	Three beats	Four beats	Six beats	Four beats
Type of Beats	Minums	Crotchets	Crotchets	Quavers	Crotchets
Time	Simple	Simple	Simple	Compound	Simple

Simple time is any time signature where the bottom value matches the value of the note e.g. A crotchet is a $\frac{1}{4}$ note, so a 4/4 time signature has a crotchet as it's the type of beat.

Compound time is a time signature where the note division is into groups of three. You immediately know you are in a compound time when you see an 8 as the bottom value.

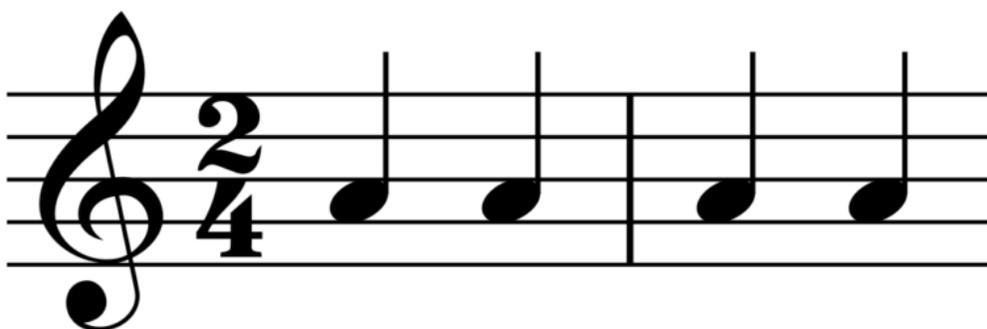


Fig 2.2.6 - A staff with 2/4 time (Aichele, 2018)

In Fig 2.2.6 above, the time signature is set to 2/4. This means a maximum of two beats, and if more notes are to be added another bar is required.

The time signature is how a musician understands the rhythm of the music and how to keep in time with the piece of music as composed.

2.2.1.5 Reading notes on a Piano

It is important for the student to firstly find the centre note on the piano, which is referred to as "Middle C". In the beginning, this can be helpful to use as a reference point when figuring out where certain notes are situated.



Fig 2.2.7 - An octave on a piano

A piano is divided up into octaves, which are a series of 12 notes (or 7 excluding the black notes) in a certain pitch. The final C in Fig 2.2.6 above, is the beginning of the next octave.

2.2.1.5 Reading notes on a Stave

Once all of the above information is understood, the student is ready to read notes on a staff. This can be overwhelming at the beginning, so it is common to develop learning techniques to help the student remember where notes are situated.

2.2.1.5.1 Treble Clef

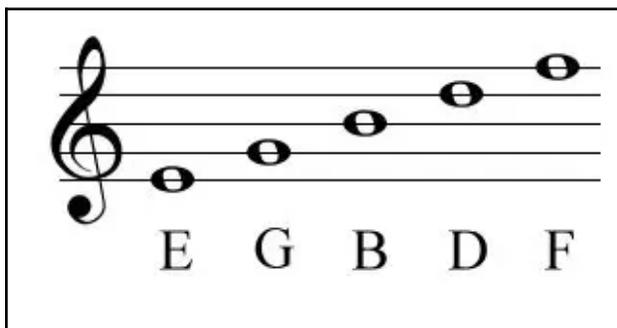


Fig 2.2.8 - The lines on a staff

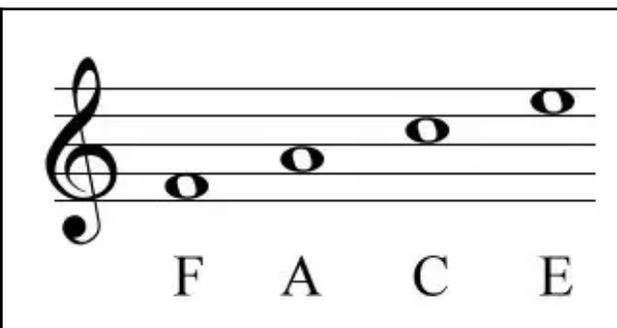


Fig 2.2.9 - The spaces on a staff

(Fig 2.2.8 - 2.2.9 sourced from Sangma, 2017)

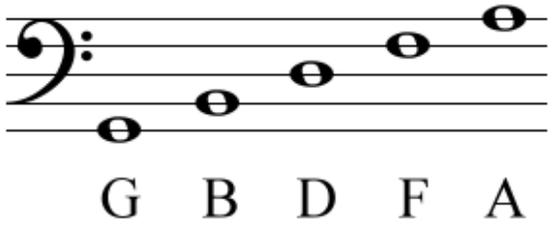
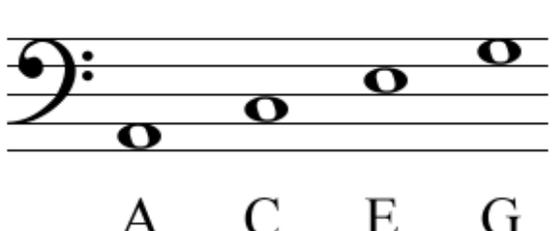
The notes placed on the lines in Fig 2.2.8 can be memorized using the following mnemonic:

Every
Good
Boy
Deserves
Fruit

The notes in Fig 2.2.9 usually don't require a mnemonic as they spell the word Face.

2.1.5.2 Bass Clef

There are also similar mnemonics to memorise notes within the bass clef.

 <p style="text-align: center;">G B D F A</p>	 <p style="text-align: center;">A C E G</p>
<p><i>Fig 2.2.10 - Lines in the bass clef</i></p>	<p><i>Fig 2.2.11 - spaces in the bass clef</i></p>
<p>Good Boys Do Fine Always</p>	<p>All Cows Eat Grass</p>

(Fig 2.2.10 - 2.2.11 sourced from How To Play The Piano, n.d.)
(mnemonics from key-notes n.d.)

2.2.2 Problem 2 - Teaching a Key

In music, a key is a set of notes that correspond to a scale. Any piece, improvised or not, will always follow this structure. The point of the structure is if you start somewhere, you should always finish there, it is described as a song's home (Dummies, n.d.).

The reason behind this structure is because, if you follow it, the sound created is pleasing to your brain. If you don't, the sound is unpleasant. For that reason, almost all music follows this structure.

For this section, the key of C will be explained, as it is the first key a student will learn on the piano.

A musical key will tell several things about a song; what sharps and flats are used (the black notes on a piano) and the scale it is based on. The first note of the scale would be the home note, and all of the other notes would have a relation to this note.

Each note within an octave on a piano has a different key, and they become more complex the higher the position in the octave.

2.2.2.1 Scales and Chords

A scale is moving through an octave in a particular key. In the case of the C Major key, it has no sharps or flats, so the movement is step-by-step from the initial C note on the piano, to the next one.

A scale will always begin on the key it is in. For example, to play an A minor scale, begin on the A note. An E melodic minor scale will start on the E key.

This document will not cover melodic minor scales, as they are advanced components of music theory, but they are mentioned in the paragraph above to prove the structure will always remain the same as one digs deeper into advanced music theory concepts.

In the case of C, the Major and Minor scales go as follows:

#	C Major	C Minor	Technical Name
1	C	C	Tonic
2	D	D	Supertonic
3	E	E Flat	Mediant
4	F	F	Subdominant
5	G	G	Dominant
6	A	A Flat	Submediant
7	B		Diminished

		B Flat	Leading Tone
1	C	C	Tonic

(Ferrant, 2020)

Once the scale is understood, the chord of the specified key can also be easily understood. A chord comprises the 1st, 3rd and 5th of a scale. Using the table above, the chords for both C Major and C Minor can be easily understood to be as follows:

#	C Major	C Minor
1	C	C
3	E	E Flat
5	G	G

2.2.3 Problem 3 - Improvising

Once the student has learnt the necessary prerequisites e.g. the fundamentals of music theory, and a particular key, then they are ready to improvise in this key. In traditional piano pedagogy, the student wouldn't usually be introduced to improvising at such an early level.

Introducing the student to improvisation as early as possible helps them to see the purpose to the theory, and gives them more of a practical approach to learning the fundamental structure of music theory.

2.2.3.1 Chord Progressions

A chord progression is the backbone of any piece. It creates rhythm, structure and sound. There are rules to how it should be structured, but it still offers a lot of flexibility to the composer.

At this stage, the student has only learnt a single key, so they will be taught to improvise solely in this key. Later on, as the student learns a second key, they will learn to improvise in that key, and will subsequently learn to improvise between the two keys they have learnt.

Chord progressions follow a certain structure, as mentioned in section 2.2.2 about a human brain either agreeing with the sound heard or not agreeing with it. The following is the structure of major chord progressions:

#	1	2	3	4	5	6
Goes to	any	5	6	1 or 5	1	2

Using this table, a simple chord progression can be composed. Continuing in the key of C, the piece should always begin (and end) with the tonic.

Following this structure, the following chord progressions had been composed:

Progression 1

Notes	C-E-G	A-C-E	D-F-A	G-D-B	C-E-G
Chord	C Major	A Minor	D Minor	G Major	Major
Sequence	1	6	2	5	1

Progression 2

Notes	C-E-G	F-A-C	G-B-D	C-E-G
Chord	C Major	F Major	G Major	Major
Sequence	1	4	5	1

Notice that both of these progressions end with a 5 - 1 progression. This is called "Perfect Cadence".

2.2.3.2 Licks

A lick is a short musical motif used during improvisation. The purpose of a lick is to create a pattern that can (but does not have to be) used on different chord changes throughout the improvisation.

It is essential to provide the student with a collection of licks, which can be used in different types of improvised pieces.

Once the student has selected a lick, they can intertwine this with particular notes of the chord they are playing, while always trying to stay within the constraints of that current chord.

For example, you have a certain lick that starts and ends on 1 (C), but you are about to move to your second chord, 4 (F), at this point, the lick should be transposed to begin

and end on the F note, while maintaining the pattern defined when playing it the first time.

2.2.3.3 The Dimensions of Improvisation

At this point, the student has the fundamental knowledge to improvise at a basic level within the key of C Major. But before the first key of improvisation is complete, it is important to show the student the dimensions of improvisation;

- Anticipation
- Use of Repertoire
- Emotive Communication
- Feedback
- Flow

(Biasutti & Frezza, 2009)

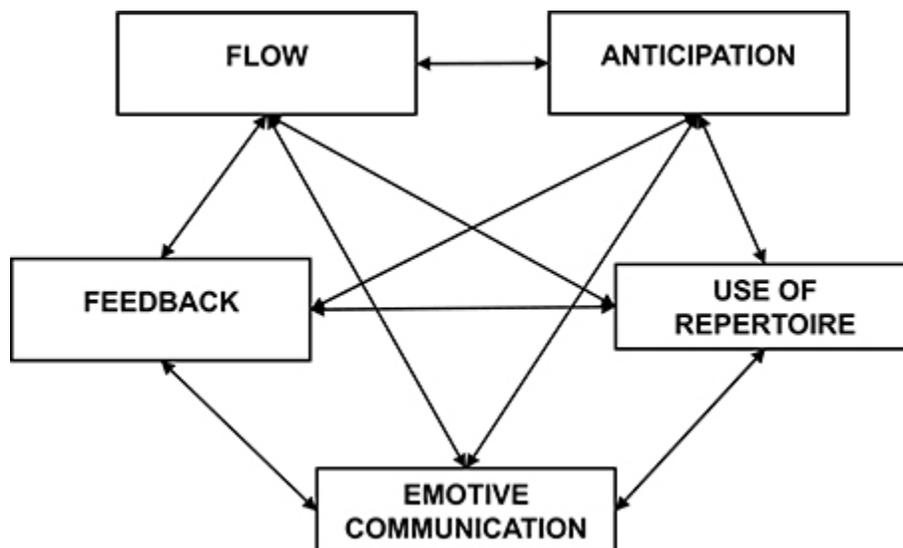


Fig 2.2.12 - The dimensions of music improvisation - Biasutti, 2017

2.2.3.3.1 Anticipation

Anticipation requires thinking in advance about rhythmic, harmonic and melodic elements of the piece while improvising. It aids the student to develop the ability to plan, and think of the structure of the improvised piece.

A possible exercise to train this ability could be to stop playing the piece on an instrument and to continue singing it.

2.2.3.3.2 Use of Repertoire

Use of Repertoire includes rhythmic and melodic features, such as licks, as described in under heading 2.3.2. Using a variety of licks helps the performer to remember these

short motifs, and broaden the selection of licks they could potentially use, and helps them develop strategies for properly selecting the right lick.

Possible exercises to develop this ability could be listening to and analysing famous musical solos, or to listen to a particular solo and sing the solo back.

2.2.3.3.3 Emotive Communication

Emotive Communication includes emotional feelings and transmission of effectiveness. The more experienced a performer is with improvisation, the more advanced the music emotions are.

Possible exercises to develop this could include getting the student to improvise while in a specific emotive state, in doing this they can find a correct sound for expressing the feeling they currently feel, which grants a process of introspection.

2.2.3.3.4 Feedback

Feedback includes developing skills that produce real-time adaptations in response to events. They are made to render the improvisation more logical and reliable. This can include both external and internal feedback. External feedback involves communication between performers, whereas internal feedback involves the performer themselves focusing on their performance.

Feedback is vital for improvising because it allows the performer to react to any changes, this could be rhythmic, melodic or anything else that the performer should be aware of, and be ready to react to.

Possible exercises to develop this skill could include getting the student to perform with their eyes closed to enhance concentration or to enhance risk-taking within a performance, like having a "conversation" between two improvisers.

2.2.3.3.5 Flow

Like speaking a language, improving requires a flow, this regards ensuring the entire performance has a natural and smooth progression to it.

Flow is a particularly difficult skill to develop as it requires all other dimensions to be well practised.

Possible exercises to develop flow could involve setting concise objectives for the student during their performance while understanding the difficulty level of these objectives.

If the objectives are too easy for the student they will find the activity boring, and if they find the objectives too difficult they will struggle to perform with a natural-sounding flow.

(Biasutti, 2017)

2.3 Conclusion

This section goes through understanding the current problem of how piano students are currently taught and comparing them to how guitar students are taught.

It investigates how a piano student is taught the fundamentals of music theory as they learn piano, but can never apply this knowledge to improvise, while a guitar player, who usually has little to no knowledge in music theory can improvise at a strong level.

This section subsequently goes on to identify the problems that may be encountered with music pedagogy, and breaks the process of teaching a student how to improve into three fundamental problems:

- Fundamentals of music theory
- Teaching a key
- Improving

These three problems are investigated in-depth, and all terminology is described along the way in the hope to educate the reader with a basic understanding of music theory.

By applying the knowledge learnt from all three sections, two Chord Progressions were developed using the structure provided.

These progressions were subsequently played by an experienced musician who had never seen them before, to prove the above information is correct.

It is believed that the musician must have a basic understanding of music theory before they are taught to improvise, so the application aims to separate music theory and improvisation lessons. Improvisation lessons will only be available to the user once they have completed the appropriate prerequisites.

3 Requirements

3.1 Introduction

The purpose of this project is to develop an application that can help the user with learning music theory using a piano. The aim here is to take inspiration from educational gamified applications such as Duolingo, which make learning something difficult fun and rewarding.

Learning to improvise on any instrument is an incredibly difficult task. The goal of this idea is to make it possible for anybody to start with basic music theory and build their level of knowledge and improve their skills.

From the outset, a huge emphasis was placed on performance. The application was designed to be fluid and easy to use on all devices. The primary reason being once a user encounters something slow or broken, they can lose all motivation to continue learning.

With that in mind, the following technologies were proposed to build the application:

- Frontend
 - Flutter
- Backend
 - Node.js
 - Express.js
 - TypeScript
 - GraphQL
- Database
 - MongoDB

3.2 Requirements Analysis

This section considers the requirements for the application. Similar applications are considered for inspiration. The advantages and disadvantages of similar applications are reviewed for inspiration. In addition, a survey is conducted to understand the needs of the user.

3.2.1 Existing applications

A review of similar applications was carried out to determine the functionality provided by those applications'. Some of the features were included in the project plan.

3.2.1.1 Duolingo (iOS, Android & Web)

Duolingo is an intuitive language learning application capable of teaching the fundamentals of over 37 languages.

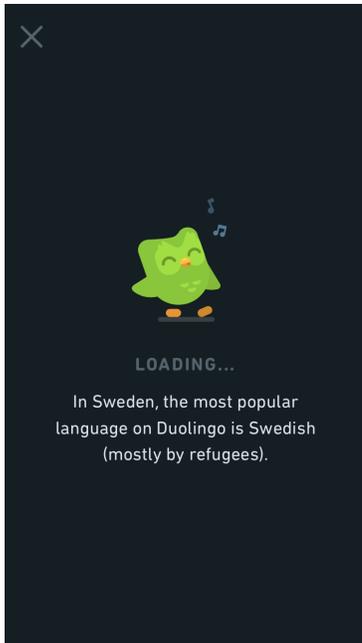


Fig. 3.1 - Loading screen provides interesting facts and cute animation



Fig 3.2 - Profile page provides tracking statistics and social options



Fig 3.3 - Skill Tree



Fig. 3.4 - Selects correct translation

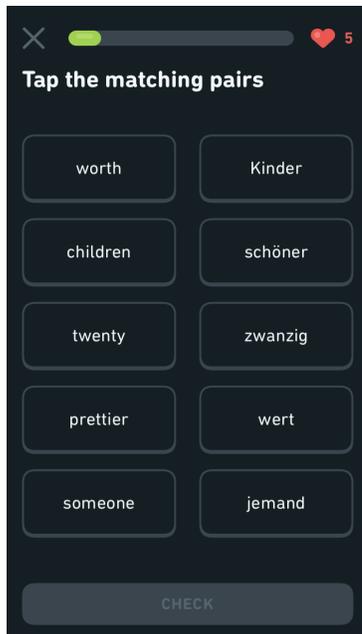


Fig. 3.5 - In-Game - Tap the pairs



Fig. 3.6 - In-Game - Translate

- Features
 - A native built application on each platform, resulting in very high performance
 - Elegant U.I., professional and friendly brand
 - Ability to learn a language either from the very beginning or from a certain level after taking an introduction test
 - Makes learning easy by rewarding the user
 - Targets skills in small minigames e.g. listening, speaking and translating as shown in Figures 4-6 above.
- Advantages
 - Very fluid and easy to use
 - Smooth animations and strong design
 - Multiple mini-games to target all types of learnings
 - Social option to compete against your friends
 - Rewards users for watching advertisements, but does not enforce it
- Disadvantages
 - If you get 5 questions wrong in the free version, you need to wait a few hours to keep going

3.2.2 Piano by Yousician (iOS, Android)

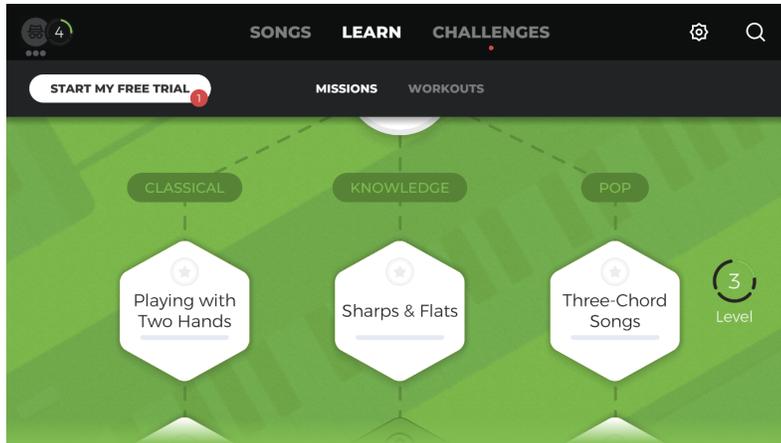


Fig. 3.7 - Learn Tab Skill tree

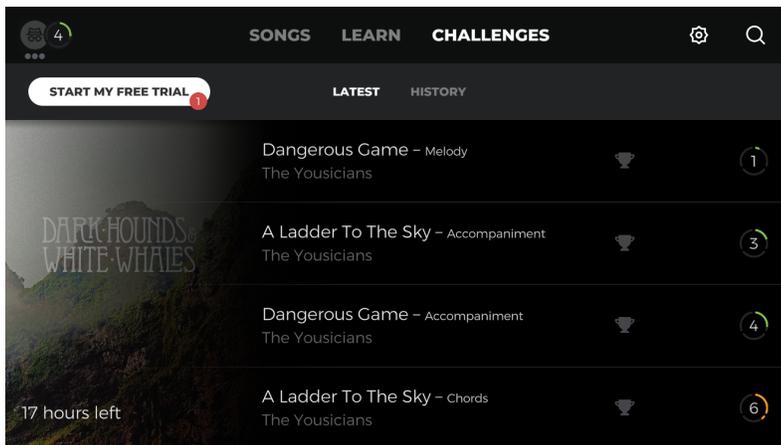


Fig. 3.8 - Challenges Tab

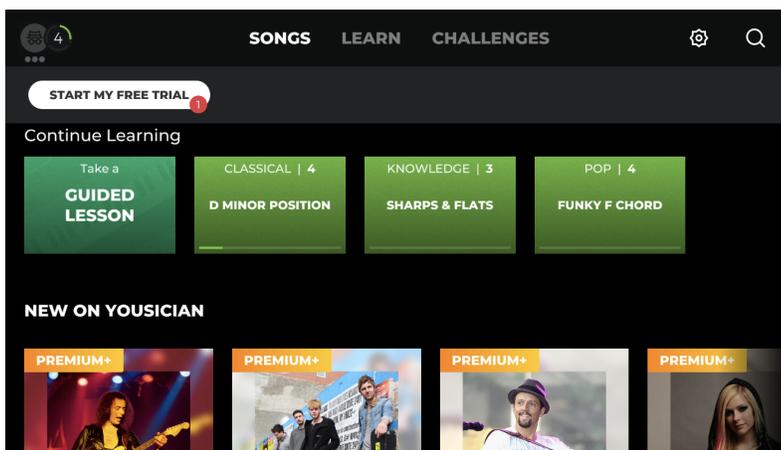


Fig. 3.9 - Songs Tab

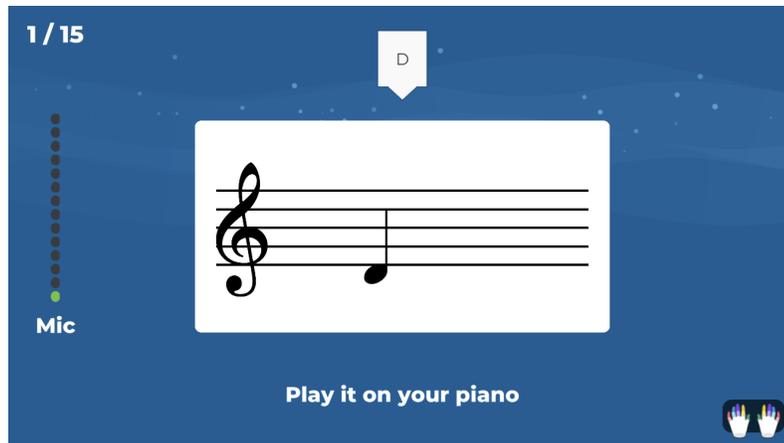


Fig. 3.10 - In a lesson

- Features
 - An application built in C# / Unity, resulting in game-like fluid speeds and animations
 - Relatively easy to use UI
 - Ability to learn the piano either from the very beginning or from a certain level after taking an introduction test
 - Makes learning easy by rewarding the user
 - Targets skills in small minigames e.g. listening, speaking and translating as shown in Figures 4-6 above.
- Advantages
 - Relatively easy to use
 - Ability to improve your music theory skills or just learn songs
 - Voice over recording talks to you and walks you through lessons
- Disadvantages
 - Quite a complex UI
 - The design feels a bit cluttered and overwhelming
 - The first thing you see as a free user is in-app purchases which generally feels a little cheap
 - You must always be by a piano to use it

3.2.2 User Profile

It is important to build a profile of the user to help understand the users' requirements.

A series of studies were carried out in an attempt to understand what would be useful to the client. Firstly a Persona was created. Then a survey was sent to over twenty people with varying skill levels in music.

3.2.2.1 Personas

A fictional character was created to understand who the relevant users are, and their needs.

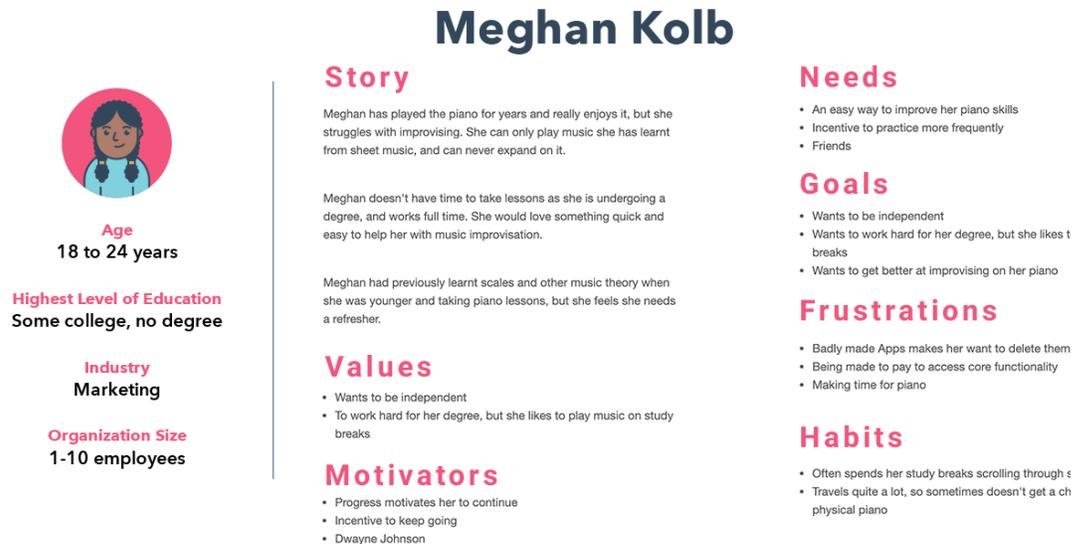


Fig. 3.2.11 - Persona of Meghan Kolb, built using [Adobe Photoshop](#)

3.2.2.2 Survey

This section will provide an overview of the decisions made based on the survey. The survey was purposely aimed at all types of musicians, even though the application is focused solely on pianists. Other musicians could still provide useful feedback.

The results of the survey were overwhelmingly positive. Twenty-one responses were received, and the feedback was extraordinarily useful.

Do you play an instrument? (Multiple Choice)

20 responses

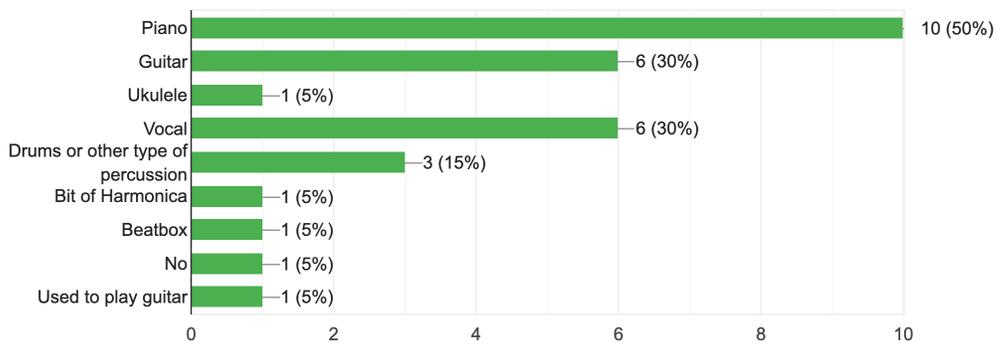


Fig. 3.2.12 - Survey Question built using [Google Forms](#)

Based on Fig 3.2.1.2, there was a good variation of musicians, the majority had played the piano (50%). Only one person who took the survey had not played an instrument.

How would you consider your knowledge of music theory? Choose one between 1 (Very basic) and 10 (Advanced)

21 responses

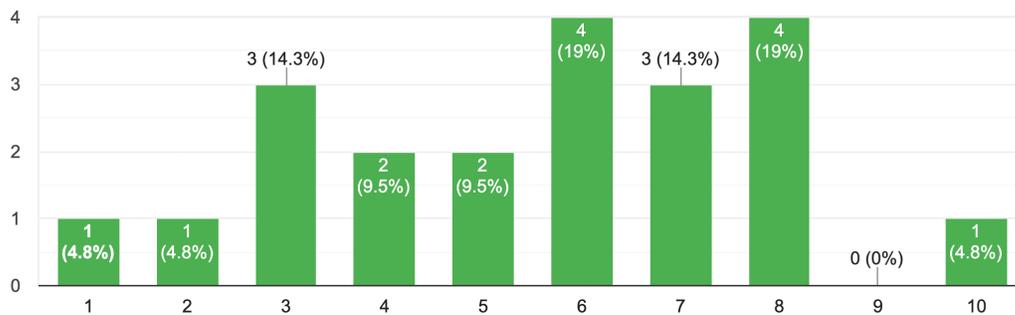


Fig. 3.2.13 - Survey Question built using [Google Forms](#)

The average level of knowledge with music theory was 5.45. The application should take into account the user's level., It should allow the user to skip ahead should the lessons be too easy.

How well would you rate your level of improvising? (Select an item from the dropdown list)
20 responses

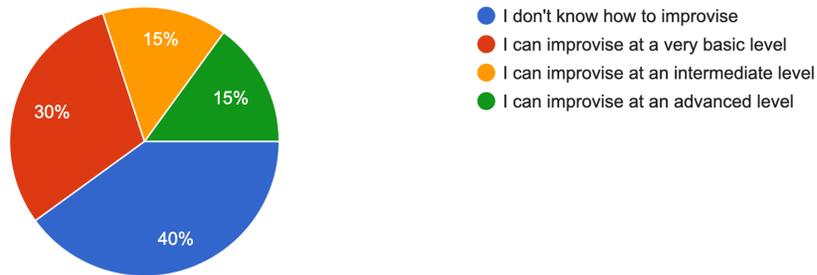


Fig. 3.2.14 - Survey Question built using [Google Forms](#)

Based on Fig 3.2.13 and Fig. 3.2.14, there was a good mix of users with varying levels of knowledge of both music theory and improvisation.

Even though a lot of users are well versed in music theory, it is clear from Fig. 3.2.14, that the majority still struggle to improvise higher than the basic level.

Do you practice a musical instrument?
20 responses

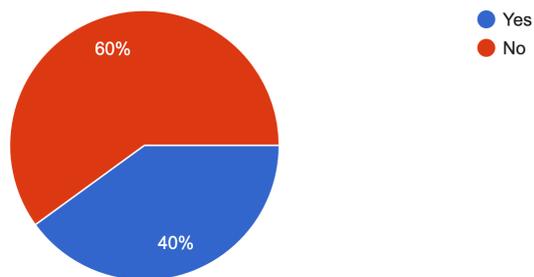


Fig. 3.2.15 - Survey Question built using [Google Forms](#)

From the responses to the above question in Fig 3.2.15, 60% of the people do not practise the instrument they play. This is a clear indication that the app requires some sort of incentive to ensure the user feels rewarded to consistently use it.

How do you normally practice?

8 responses

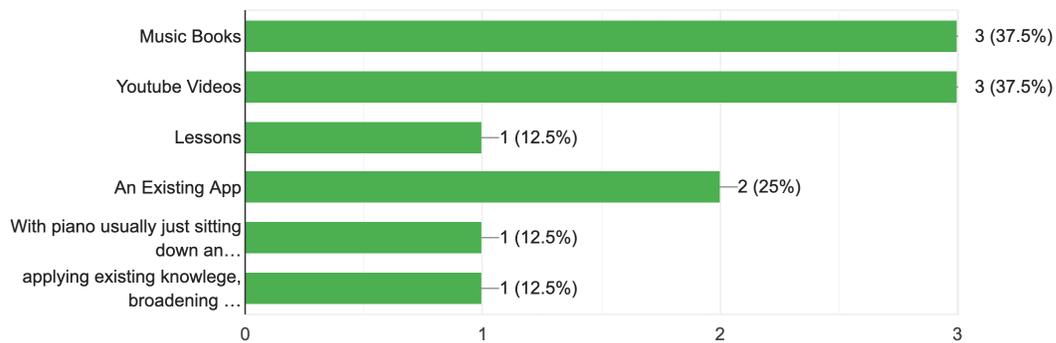


Fig. 3.2.15 - Survey Question built using [Google Forms](#)

Is there any way this method of practising could be improved?

7 responses

N/a
If it was more interactive
Achievements, if it listened to you and marked your timing etc
Some apps have bad UI or have to pay. Scheduling for practicing on an app. Such as chords on day or scales the next.
more detailed knowledge, consistency, less fear
It would be good if I knew more techniques are or finger/hand exercises to release tension and be faster switching from keys
Easier and online

Fig 3.2.16 - Survey Question built using [Google Forms](#)

Based on the responses of 2.15 and 2.16, it is clear most users are looking for some sort of help when it comes to practising in general. Music Books and Youtube Videos are popular, but they provide no feedback on improvement or mistakes. Existing apps do exist but some have either a bad user interface (UI) or encourage the user to upgrade before offering anything.

Do you think an application to help you learn music theory & improvising is useful?
20 responses

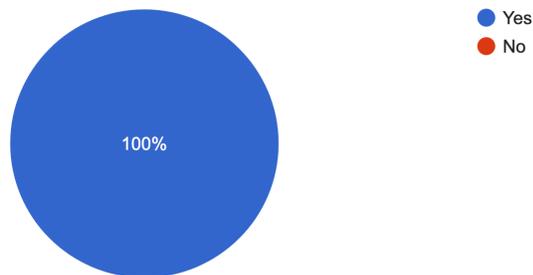


Fig. 3.2.17 - Survey Question built using [Google Forms](#)

Based on Fig 3.2.17, 100% of submissions had thought the application would be useful.

What type of functionality would you like to see?
20 responses

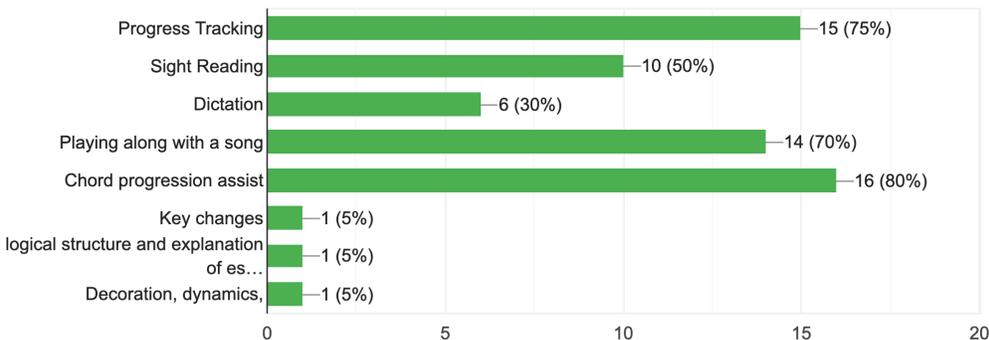


Fig. 3.2.18 - Survey Question built using [Google Forms](#)

Based on the responses of Fig. 3.2.18, the majority of users search for a practical application, based more on interactivity, with real-time feedback whilst rewarding consistency.

Link to the Survey - <https://forms.gle/jj9JHxrFZJvDMoRY7>

3.2.3 Requirement Modelling

This section discusses the functional and non functional requirements based on the research and requirements analysis that was carried out in the previous chapters.

3.2.3.1 Requirements

A series of functional and non-functional requirements was compiled non-functional to understand what the application should offer to users. The functional requirements are purely technical functionality, whereas the non-functional are more of a general idea.

3.2.3.1.1 Functional Requirements

- User Authentication
- User skill tree
- Pitch detection
- Music theory lessons
- Improv lessons
- Lessons a user can do with a piano
- Lessons a user can do without a piano
- Progress tracking to show the user how much they've improved
- Streak/reward system

3.2.3.1.2 Non-Functional Requirement

Requirement	Description
Performance	The application should be fluid, have quick response times from the server
Usability	The application should offer an easy to use navigation and have a satisfying, friendly UI
Reliability	The application should be reliable and there should be contingencies in place should the server crash.
Scalability	The application and server should be able to handle a growing number of users
Security	All data submitted to the application should be protected and stored securely
Maintainability	The applications' codebase needs to be easily maintainable for all devices it is deployed to

3.2.3.2 Use Case Diagram

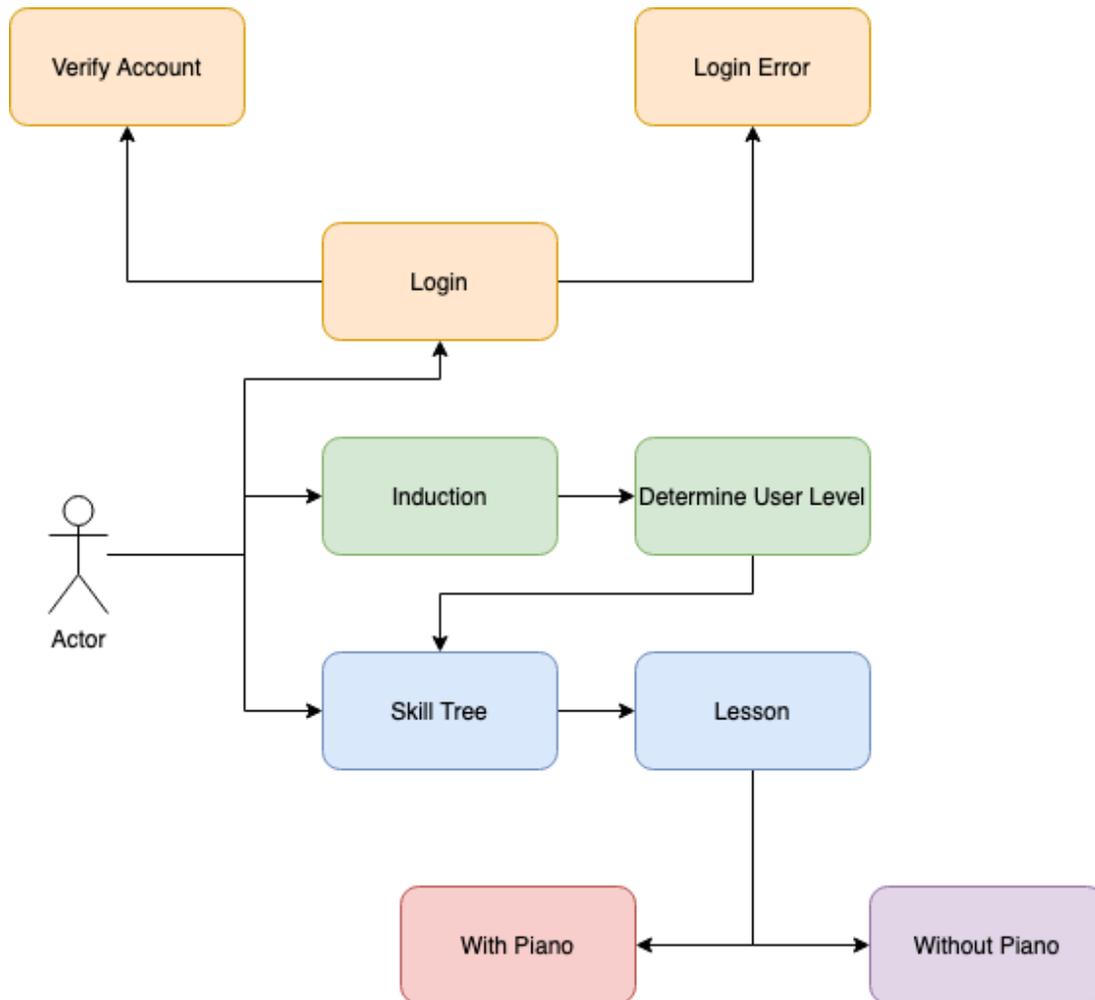


Fig. 3.2.12 - Use Case Diagram - Built using [Draw.io](https://draw.io)

3.3 System Model and System Requirements

A system model was developed to identify the primary components of the application and to consider how these components will communicate with each other. The section provides a diagram representing the system model and a discussion about its components.

3.3.1 Possible Technologies

3.3.1.1 MERN Stack

The MERN stack consists of MongoDB, Express.js, React.js & Node.js. However, if this stack were chosen, it would be an intelligent decision to swap out React.js for React Native, allowing the application to be deployed to the iOS App Store, and the Google Play Store.

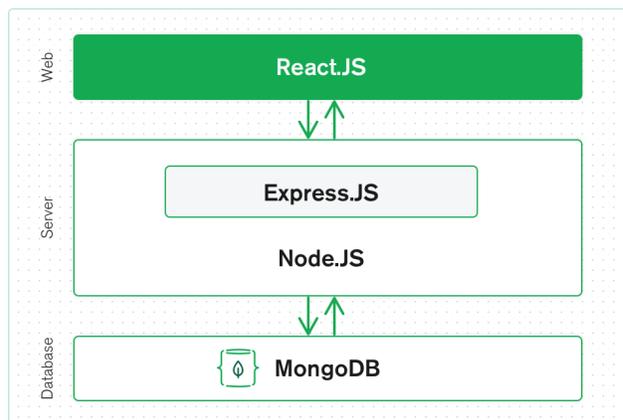


Fig 3.3.1 - The MERN Stack - [mongodb.com](https://www.mongodb.com)

3.3.1.3.1 Advantages

- Express.js and Node.js go hand in hand and are very powerful together
- The MERN stack is a very popular technology stack. There are a lot of online resources available to aid with the development.
- MongoDB is a NoSQL document database, which allows the developer to work with JSON throughout the technology stack.
- All components of this stack are open-source

3.3.1.3.2 Disadvantages

- Although non-relational databases are hugely popular, they still do not compare to relational databases. The latter is still used by large technology companies like Facebook, Google etc.

3.3.1.2 Laravel, Vue.js & MySQL



Fig 3.3.2 - Laravel, Vue.js & MySQL - Built using draw.io

Using Laravel, Vue.js & MySQL together is not only very powerful but also quite straightforward to set up. Since Laravel is built in PHP, it goes hand-in-hand with MySQL. It provides ease of use, flexibility & scalability. The same goes for Vue.js, which is a very developer-friendly JavaScript Framework.

3.3.2.1.1 Advantages

- Very developer-friendly
- Clean Architecture
- High performance and steady reliability

3.3.2.1.2 Disadvantages

- Data must be converted from SQL to JSON format for frontend manipulation
- Vue.JS can only be deployed to the web unless it is used as a tool like NativeScript to run on a mobile device.

3.3.1.3 Flutter, GraphQL, Express.js, Node.js & MongoDB

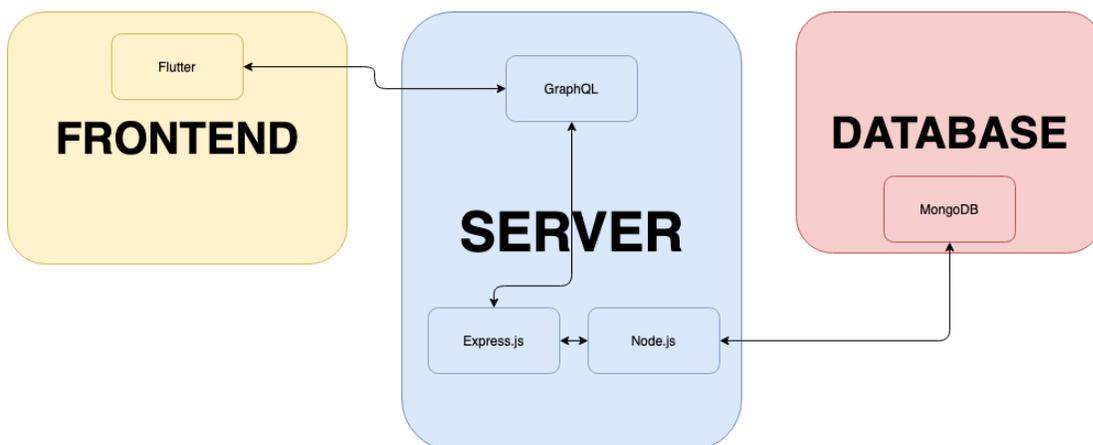


Fig 3.3.2 - Flutter, GraphQL, Express.JS, Node.JS & MongoDB - Built using draw.io

It is possible to combine the server-side of the MERN stack, with a more reasonable technology for the frontend. Since the primary requirement of this application is performance, it is a good idea to use Flutter, as it offers Native performance on Mobile Devices.

The Flutter team show their architecture layers in their documentation, the technology runs on a C/C++ engine. They also mention the embedder provided, which is written in the language appropriate for the running platform - e.g. for Android: Java and C++, and for iOS / macOS: C / Objective C++.

3.3.2.3.1 Advantages

- Easy to use coming from a background in web development
- Native device performance
- The flexible development environment and hot reloading
- Built-in component libraries (Material UI + Cupertino)
- Multiple platform deployment from one codebase
- Options to later implement web/desktop applications
- GraphQL offers lightweight queries and relatively straight forward implementation
- Very strong documentation

3.3.2.3.2 Disadvantages

- Very little documentation on Flutter, GraphQL & Node.js combination
- Flutter is still a relatively new Framework, so some documentation may be lacking

3.3.2 Conclusion

After an in-depth study of multiple technologies and technology stacks, it was decided to use Flutter as the frontend technology, and a combination of GraphQL, Express.JS, Node.JS as the backend technologies, and MongoDB as a database.

Although the developer is well versed in web technologies like React.JS & React Native, there was an inclination to try a different programming language. Considering what Flutter has to offer compared with the other languages, it presented as a clear winner.

3.4 Feasibility

The technology being implemented into this project are as mentioned in the introduction:

- Frontend
 - Flutter
 - Material UI
- Backend
 - Node.js
 - Express.js
 - TypeScript
 - GraphQL
- Database
 - MongoDB

In the development of this project, it is feasible to create such an application using the technologies stated above. There is a strong emphasis on performance from the ground up. As a result, Flutter will be used to deliver a native frontend performance and GraphQL for high-performance database querying.

The main issues are believed to be in the Frontend, building the application using Flutter which is a relatively straight forward technology, The idea to implement such functionality as pitch detection in a programming language that is unfamiliar to the developer is ambitious.

Although Flutter is very developer-friendly, it is still a new technology for the developer. In an attempt to overcome these issues, the developer enrolls in several tutorials to become proficient with the technologies.

The server will be written using TypeScript. The reason for that is that it is extraordinarily powerful to use. Although it does result in more time declaring attributes and types, It can immediately capture errors by the developer. The result is in saving time, in the long run, debugging poorly written code.

3.5 Project Plan

This project will be approached using an agile methodology. It is impossible at this stage to predict what exactly will be needed to be done each week, so the project plan is to have a weekly meeting with the project supervisor, to discuss what has been done, and what should be done for next week.

All meetings will be logged, and all tasks will be added to Trello to aid with project management.

The following is a proposed project plan but could be subject to change.

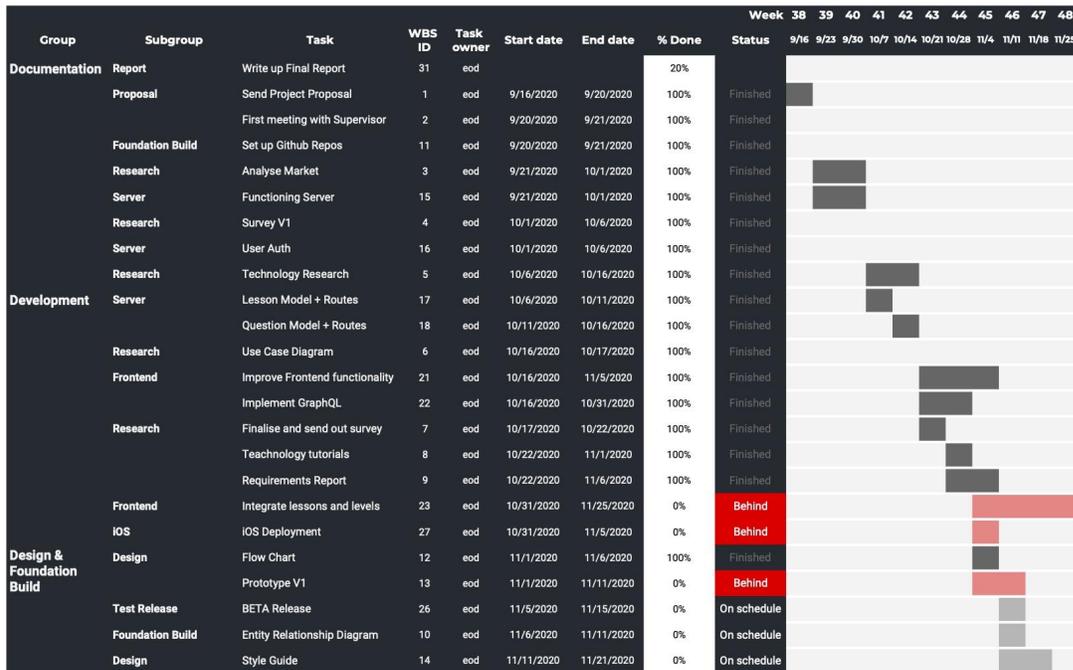


Fig. 3.5.1 - Project Plan built using [Google Sheets](#)

The above diagram shows a preview of the project plan, which was built using the Gantt project management methodology.

The full project plan can be previewed from this link:

<https://docs.google.com/spreadsheets/d/1gVTDwOJfBwC7Ms1rhW8vOj6u88FkRh5OSfAc-FHSdbA/edit?usp=sharing>

3.6 Test Plan

A series of tests will be conducted on an application of this magnitude. As the emphasis is on performance, it is important to ensure each component of the application operates at maximum efficiency.

The following are the different types of tests that will be performed on the application:

3.6.1 Unit Testing

Unit testing is a type of automated testing, usually written by the developer after writing a certain piece of code. It ensures that builds run smoothly, and aids with preventing crashes at runtime.

Unit tests must be performed on both the front and back end of the application, and it is also a good idea to set up a continuous integration system using a service like Github Actions, allowing for automatic deployment after all tests have passed.

3.6.2 User Testing

The purpose of User Testing is to get real users to test your application, without the developer interfering with their choices. A User test usually contains a list of tasks that the tester needs to complete. They should not be helped in any way, but rather observed on how they go about completing those tasks. If any of the tasks are particularly difficult for the tester to complete, they should be reviewed and changed if needed.

It is intended that user testing will take place in the design phase (with prototypes), and the implementation phase with an Alpha or BETA release of the application.

3.6.3 System Testing

The purpose of System testing is to test the entire system, including how its components communicate with each other, and the speed at which this happens.

System testing will take place in the implementation phase of the project, to ensure the entire system will perform at high speeds.

Various types of System tests will take place, including Profile Analysis on the mobile app opening speeds, using tools like XCode Instruments and Android Studio.

Network request tests will also be performed to ensure data can be retrieved quickly from the server

3.6.4 Integration Testing

Integration testing is also known as functional testing. Its primary purpose is to test certain CRUD (Create, Read, Update and Delete) services on the application. A perfect example of this would be if the user wants to update

their username. It is intended that integration testing will take place during the implementation phase of the project.

4 Design

4.1 Introduction

This chapter describes the design of the application. It aims to resolve the problems proposed in the research and requirements chapters. The primary focus of the application is to educate users in a gamified manner in music theory, and subsequently with improvising on the piano.

Due to music theory being so dense, it is essential to present the information in a consumable way to the user, so the problems presented in the research chapter can be solved. Educating the user in a gamified way means the user should be rewarded for the activities they part-take in, and they should be given a reason to return to the application and use it consistently.

The design of an application is divided into:

1. Program Design
2. User Interface Design
3. Content Design

4.2 Program Design

The program design refers to the design required to make the task of programming and coding of the application more straightforward. Due to the nature of the idea, the application must be fluid and easy to use for the user. This means all components of the application should communicate efficiently.

This chapter includes:

- What technologies were chosen
- Why the technologies were chosen
- The structure of the technologies
- The architecture of the application
- Database design

4.2.1 Technologies

The technologies are originally chosen, as defined in the requirements section, to build this application were:

- Frontend
 - Flutter
 - Material UI
- Backend
 - Node.js
 - Express.js
 - TypeScript
 - GraphQL
- Database
 - MongoDB

After further research was conducted, it was concluded that some backend technologies would need to be altered to build the server more efficiently. The technology stack was restructured as follows:

- Frontend
 - Flutter
 - Material UI
- Backend
 - Node.js
 - Apollo-Express
 - Mikro ORM
 - Type-GraphQL
 - TypeScript
 - GraphQL
- Database
 - MongoDB

Type-GraphQL, Apollo and Mikro ORM were added to the technology stack because it became clear that building a GraphQL server using Node.js & Express.js resulted in a lot of code being repeated.

When building a GraphQL server, it is important to define definitions for each type of data the server will process and store.

```
const typeDefs = `
  type User {
    id: ID
    firstName: String
    lastName: String
    password: String
    email: String
  }
`
```

Code Block 4.2.1 - A sample type definitions using GraphQL

Along with this, there is also a need to define a schema for data that will be stored in the database (MongoDB):

```
const userSchema = new mongoose.Schema({
  firstName: { type: String },
  lastName: { type: String },
  password: { type: String },
  email: { type: String },
});

const User = mongoose.model("user", userSchema);
```

Code Block 4.2.2 - A sample mongoose schema to store in a MongoDB instance

The above code blocks (4.2.1 & 4.2.2) are simplified for example, but the server architecture contains a lot of configuration, and building the entire server using this method would result in a lot of repeated code.

For that reason, Mikro ORM & Type-GraphQL were implemented, which allows the definition of both user type definitions and schema in one go:

```
@ObjectType()
@Entity()
export class User extends Base<User> {
  @Field()
  @Property()
  public firstName: string;

  @Field()
  @Property()
  public lastName: string;

  @Property()
  password: string;

  @Field()
  @Property()
  @Unique()
  public email: string;
}
```

Code Block 4.2.3 - A User entity sample, which merges a schema and type definition

The use of these technologies aims to simplify the development of the backend by aiding with defining relationships between different entities - e.g. A user could have multiple completed lessons, which can be added as an OneToMany relationship to the user entity. However, Mikro ORM & Type-GraphQL are new technologies that need to be learned to be used correctly.

```
@Field(() => [Lesson])
@OneToMany(() => Lesson, (b: Lesson) => b.user, { cascade: [Cascade.ALL] })
public completedLessons = new Collection<Lesson>(this);
```

Code Block 4.2.4 - A OneToMany relationship defined within the User entity

Type-GraphQL was also useful for the validation of data being sent to the server. As shown in Code Block 4.2.5 below, we can ensure that all data being sent to the server to add a user, for example, is in the correct format, and contains all the required information to create said user.

```
@InputType()
export class UserValidator {
  @Field()
  @IsEmail()
  @IsNotEmpty()
  public email: string;
}
```

Code Block 4.2.5 - A UserValidator sample using type-graphql and class-validator

Apollo Express would be used instead of Express.js because it integrates nicely with a GraphQL & Mikro ORM environment. It is a community-built package built with high performance in mind, along with ease of use with express.js. (apollographql, 2021, January 16)

For the Frontend of the application, a decision was made to use Flutter, along with Material UI as a component library. The reason for this is Flutter offers Native performance on Mobile Devices, resulting in a very fluid optimized application. Material UI was chosen as it is built into Flutter out of the box. It also is relatively straightforward for a developer coming from a background in Web technologies like JavaScript.

4.2.2 Structure of the Technology Stack

As mentioned in Section 4.2.1 above, the following technologies were chosen to build this application:

Frontend

Backend

Database

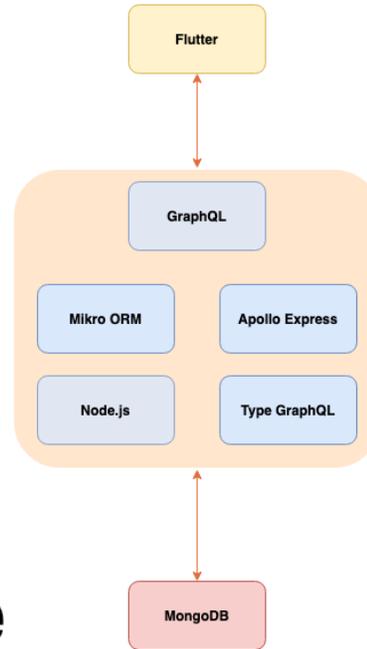


Fig 4.2.2 - Chosen Technology Stack Breakdown - built using draw.io

The front end would need to communicate with the server using a dart package named graphql_flutter. This allows the application to make GraphQL queries, mutations and subscriptions to the server. It is an open-source package built to operate similarly to Apollo's client package, which is very useful for development as a lot of the documentation and code is structured similarly to Apollo.

GraphQL supports queries, mutations and subscriptions. Queries and mutations work quite similar to how REST API would operate, but subscriptions work similarly to how a web socket would operate.

Table 4.2.1 below displays the supported GraphQL methods, what they are similar to and an example of each:

GraphQL Method	Similar to	Description	Example
Query	REST GET Request	Retrieving data from the server	Retrieve a list of lessons
Mutation	REST POST Request	Posting data to the server	Creating a new user
Subscription	Web Socket connection	Real-time information from the server	Viewing a list of users who are online in real-time

Table 4.2.1 - GraphQL methods compared to traditional data transferring methods

Having subscriptions built-in is extraordinarily useful to the developer, as, without it, one would normally need to install a package on both the frontend and backend, which would communicate with each other. This results in a much simpler way of displaying real-time data.

4.2.3 Design Patterns

The application is designed similar to the Model View Controller (MVC) Design pattern.

Instead of Models, the application uses entities to represent the data. e.g. A user would be an entity, and it would contain information like name, email password etc.

The view would be where Flutter comes in, it is everything displayed to the user where GraphQL is used to fetch this information.

Instead of a controller, a resolver is used. This is where all of the business logic for a certain user would happen. e.g. Creating, viewing, updating and deleting a user would all happen within the resolver.

4.2.3 Application architecture

Fig 2.3 below includes a diagram of the application architecture

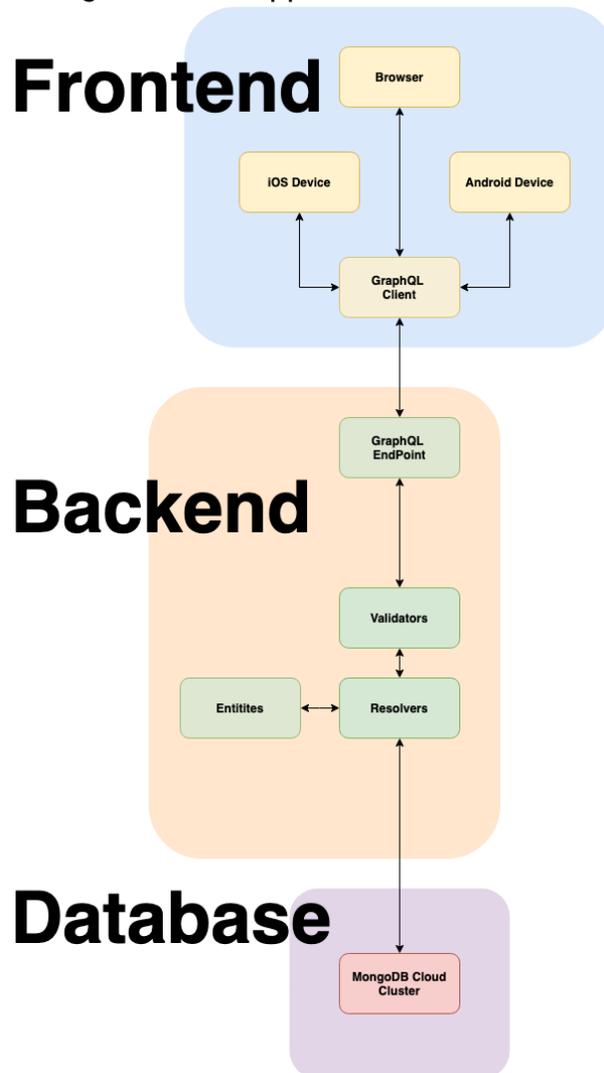


Fig 4.2.3 - Application Architecture - built using draw.io

As shown in Fig 4.2.3 above, all requests to the server can only be done via the GraphQL Client installed within a Flutter Application. This application can either be an iOS device, an Android device, or a web browser (Dependent on sufficient development time). The GraphQL Client communicates directly with the GraphQL EndPoint on the server, which in turn only exposes a single endpoint from the server.

The server will check any requests with the appropriate validator to ensure the requested data is in the correct format before executing any business logic e.g. creating a user.

After the validator passes, the resolver will create a new user using the user entity and will store that in the MongoDB Cloud Cluster, which is hosted in the MongoDB Cloud. The server communicates with the Database using a Uniform Resource Identifier (URI).

4.2.4 Database design

This section discusses the structure of the database. An Entity Relationship Diagram (ERD) was developed to understand each entity, its appropriate data types, and its relationship with other entities.

The User entity is the most complex, as it contains two arrays of foreign entities; Completed Modules - to track which modules the user has completed and Incorrect Questions - which is intended to ask the user these questions again at a later lesson, in a hope that they have learnt from their previous incorrect answer.

A Module contains a collection of lessons and an appropriate level. A User cannot partake in a module if their level is below the level of the module. It also contains a type of either Music Theory or Improvisation, which distinguishes the type of module it is.

To complete a Module, the User would need to complete all lessons associated with it. Once completed the users level will be incremented.

A Question entity is the most flexible entity, as it can be one of four enum types; Scale, Chord, Sign Reading or Chord Progression. There is also a boolean, requiresPiano, which separates questions into ones that do or do not require a piano. The reason for this is if the user is in a position where they don't have access to a piano, they can continue to partake in Lessons.

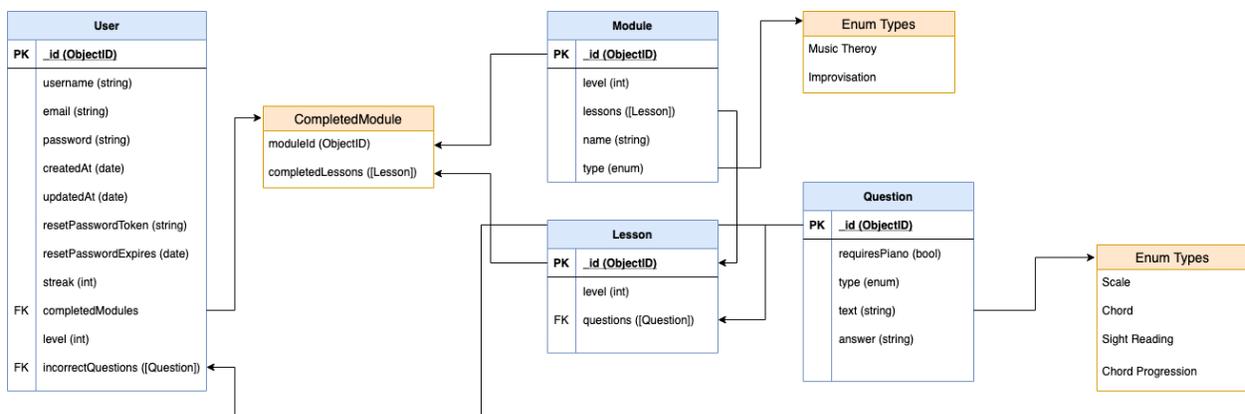


Fig 4.2.4 - Entity Relationship Diagram V5 - built using draw.io

4.2.5 Data Design

A strong benefit of GraphQL is when you query the server, you specify how much or how little data you want to be returned. Queries can be combined into one to prevent subsequent requests for similar data.

Take code block 4.2.5 below as an example:

```
{
  get {
    id
    name
    email
  }
}
```

Code block 4.2.5 - a get user graphql query

The above query requests the current logged in user on the server, by their JSON Web token (JWT). The user decrypts this token and can see which user is currently logged in.

If no token is provided, the user returns 401 Unauthorized. Otherwise, it will return the requested information above - id, name and email.

```
{
  "data": {
    "get": {
      "id": "5ff0a833e20d78000431491d",
      "name": "test",
      "email": "test@test.ie"
    }
  }
}
```

Code block 4.2.6 - Query response

Code block 4.2.6 displays the response on a successful request. One can see that it gives only the information requested, and nothing more.

This also works for relational data. For example, if one wanted to request all lessons the current logged in user has completed one could run the following query:

```
{
  get {
    id
    name
    email
    completedLessons {
      id
      level
    }
  }
}
```

Code block 4.2.7 - a get user and completed lessons graphql query

The query above in code block 4.2.7 requested the id and level of all lessons the current logged in user has completed. This will return an array of lessons within the user object.

Selecting as little or as much information as required makes the server extremely flexible for all sorts of applications to connect to it. For instance, the mobile application can request this information easily, but there could also be an admin panel that connects to it, requesting data that is shaped in an entirely different way.

GraphQL accommodates this in a very easy way. All of the queries and data schemas can be viewed in the development environment, using the GraphQL playground, which is provided and can be used to test all queries.

4.3 User interface design

3.3.1 Wireframe

A wireframe shows the content and functionality for the layout of a page. A wireframe usually does not look at typography or colour.

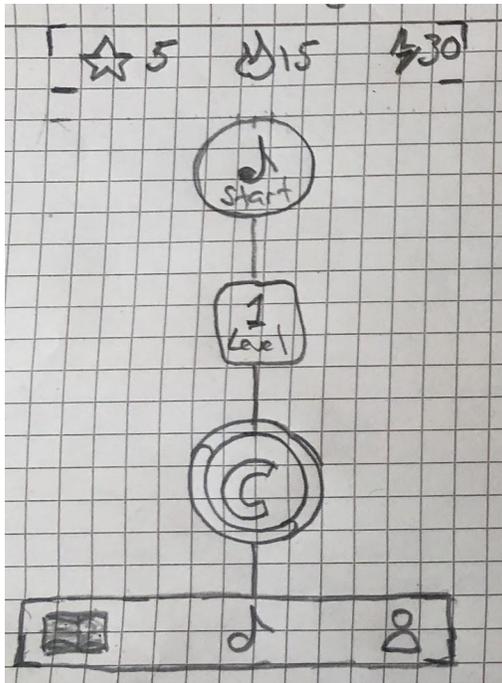


Fig 3.3.1.0 - Music Theory Screen

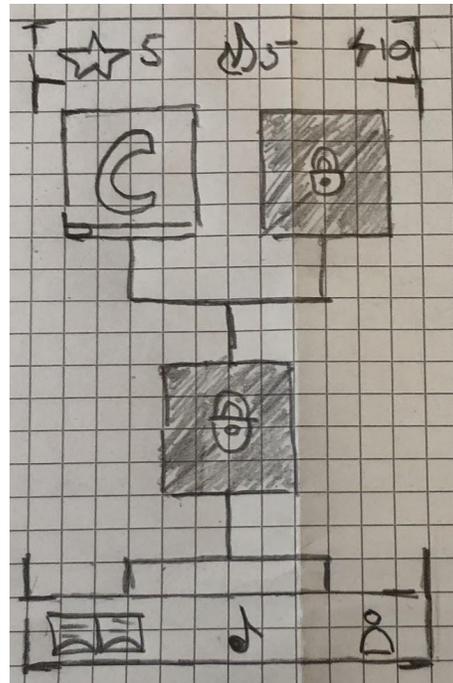


Fig 3.3.1.1 - Improv Screen

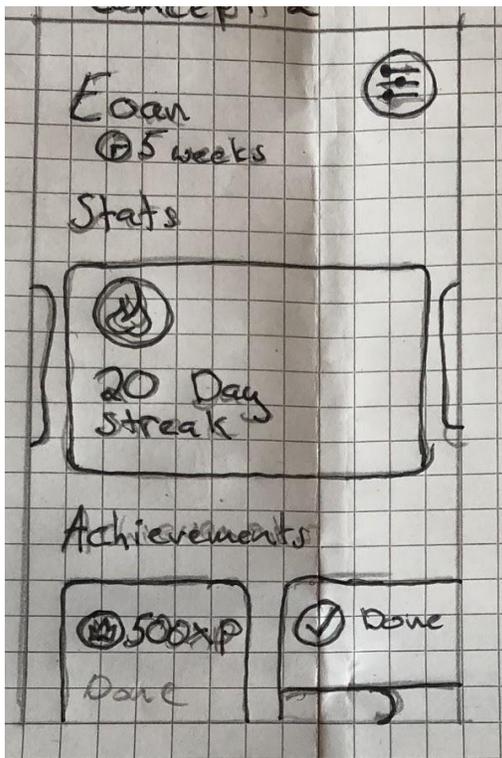


Fig 3.3.1.2 - Profile Screen

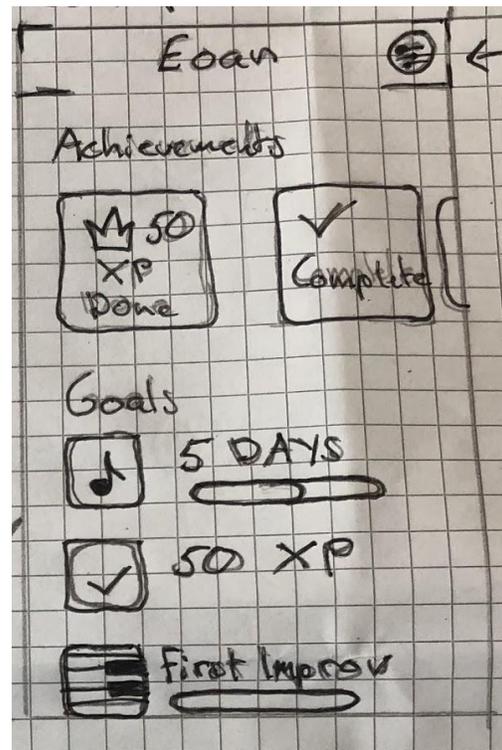


Fig 3.3.1.3 - Profile Screen on Scroll

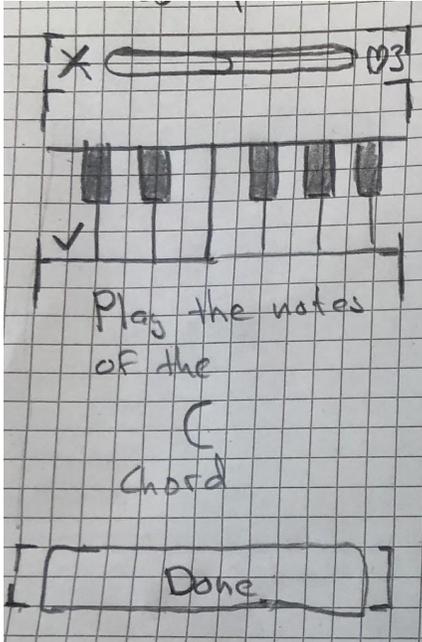


Fig 3.3.1.4 - Lesson Variation 1

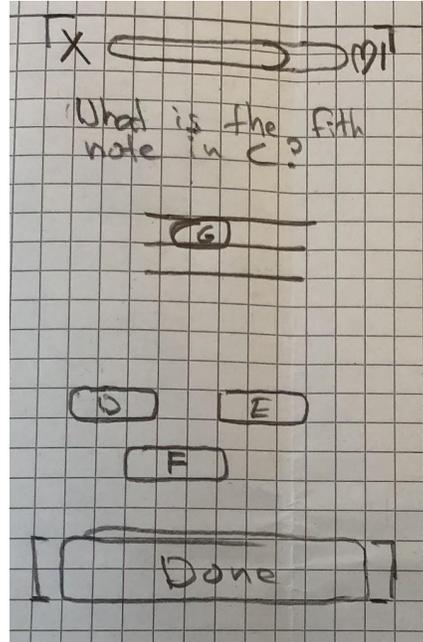


Fig 3.3.1.5 - Lesson Variation 2

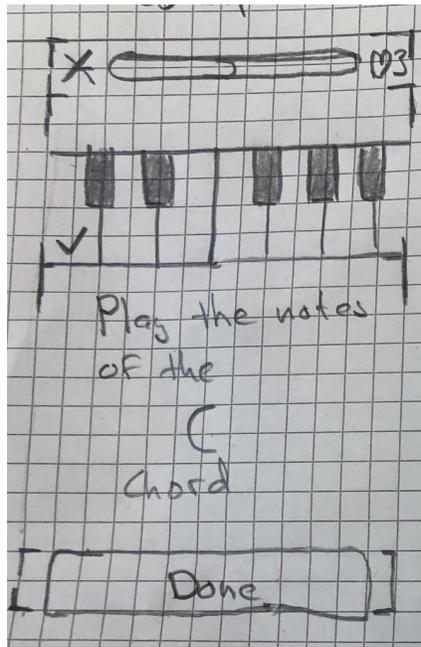


Fig 3.3.1.6 - Lesson Variation 3

3.3.2 User Flow Diagram

This shows how the user will navigate from one page to another page within the application.

A flow chart was created to understand how a user would interact with the application.

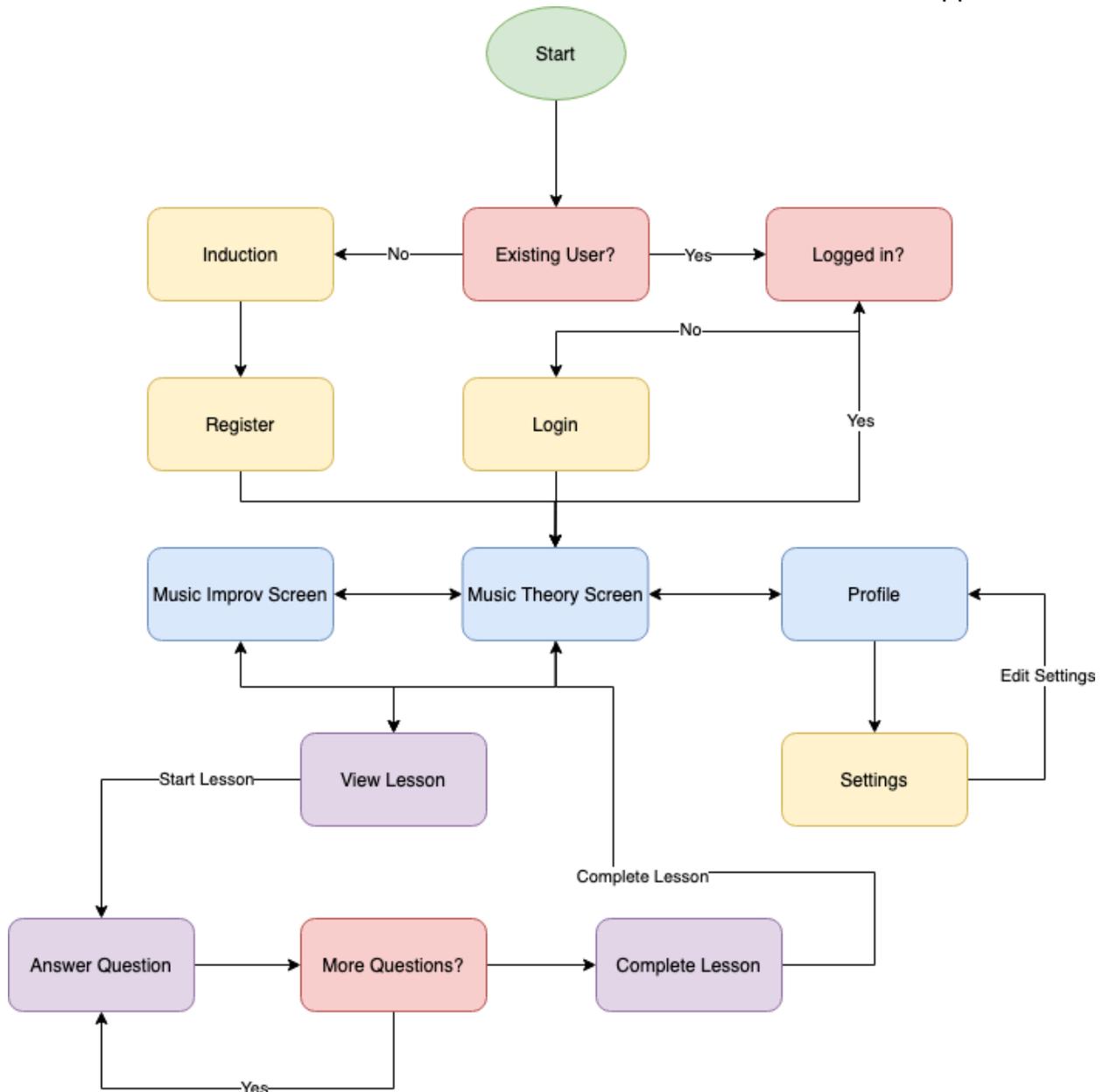


Fig 3.3.2.1 - User Flow Chart built using draw.io - built using draw.io

Fig 3.3.2.1 above portrays how a user will flow through the application. The application first checks whether the user is existing or not. This will entail some key-pair value

stored within the phone's storage to tell the application whether it has been opened before or not.

If the user is non-existent, they would be directed straight into an induction, where the user's level would be determined. This is done by asking a series of questions with varying degrees of difficulty. The intention here is to capture the user's interest immediately with a reward for answering questions, Another approach is to make the user register immediately, which may result in them losing interest.

Once completed, and the user has been successfully authenticated, they will be brought to the main screens (The blue boxes in Fig 3.3.2.1 above). From here they can see a variety of lessons between music theory and improvisation, depending on how they performed in the induction.

3.3.3 Style guide

This shows the colours, typography and layout for a single page. Often the theme for this page will be used for all pages in the app. Within this section, the colour scheme is explained and why that colour scheme has been chosen. This section also covers which fonts are being used, why they have been chosen, grids and spacing.

The application uses Material UI as a component library. There are multiple reasons for this including developer experience, the strong standard of documentation and the fact that it is built into Flutter, so it requires no additional packages to be installed into the application.

Using Material UI involves following their design guidelines on typography, colour, shape and sound.

3.3.4.1 Colour

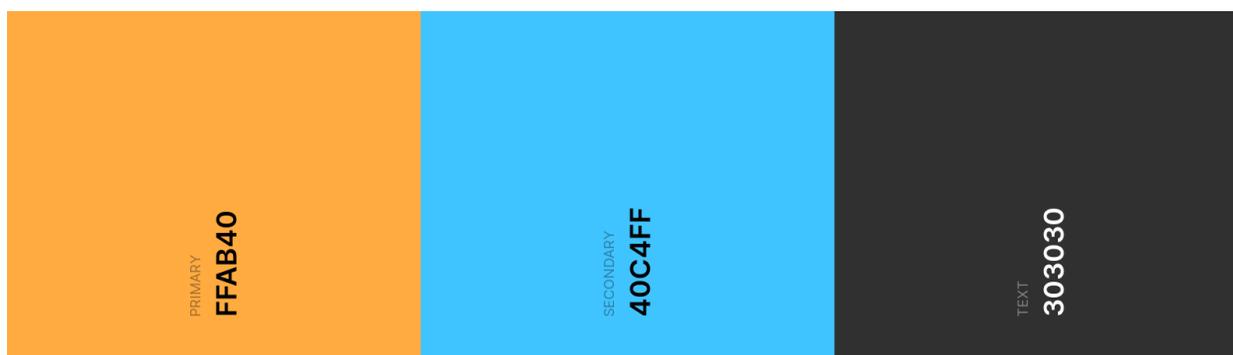


Fig 3.3.4.1 - Colour Palette V2 - Built using coolors.co

Fig 3.3.4.1 above is the selected colour palette for the application. The primary and secondary colours will be the most prominent, but following Material UI's colour

guidelines, it is suggested that to have both light and dark variations on both primary and secondary colours, which are automatically generated within the theme.

The colours were generated using Material Design's Color Tool, which gives a full selection of all of their recommended colours, their variations, and what type of text colour is legible on each.



Fig 3.3.4.2 - Primary Colour variations and legible text using [Material Designs Color Tool](#)

3.3.4.2 Typography

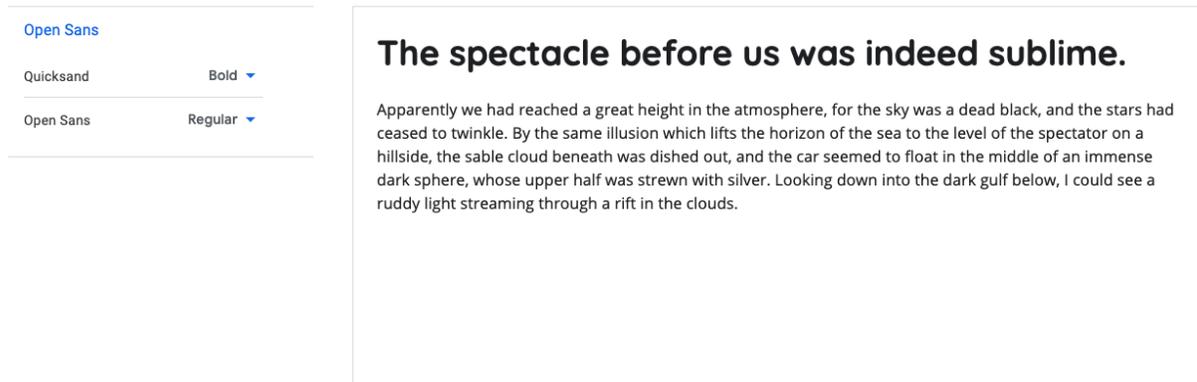


Fig 3.3.4.2 - Typography, generated using [Google Fonts](#)

As shown in Fig 3.3.4.2 above, Quicksand Bold was chosen as the heading font for the project, and Open Sans Regular was chosen as the body font.

Quicksand was chosen as it functioned well with the rest of the design of the project. The font has solid edges and reflects a friendly tone, which fitted perfectly within the educational minimalist design approach.

Open Sans Regular was chosen as the body text because Google Fonts had recommended it as a popular pairing. It was believed that the combination looked good together, and the presented information is clear, concise and modern.

3.3.4.3 Shape



Fig 3.3.4.3 - Shape as defined by [Material UI's Design guidelines](#)

Fig 3.3.4.3 displays a default Material shape. This will be used for Cards throughout the application, to present a sense of emphasis and hierarchy.

The Shape will be adjusted to have slightly rounded corners of about 4dp or 8px. It is believed that giving the shape a softer edge provides a friendlier and more educational approach to the design.

3.3.4.4 Sound

Material Design defines sound as a method to communicate information that improves the user experience (Google, 2020)

It is broken up into three sections within an application; sound design, music and voice. All of which portray brand identity in different ways.

For the application, it is intended to use sound design for actions the user takes, e.g. correctly answering a question, or completing a lesson. The aim is to associate successful actions with a specific sound, which can reinforce the meaning of the interaction, and the satisfaction of the user.

Material Design describes the sound as feedback as “Earcons” as is considered an important aspect to implement into your application, despite it often being overlooked.

It is also mentioned, however, that silence is just as important as applying sound. There are many instances when sound is not required and using too much sound can have a negative effect on the user experience. Research has shown that sound should not be implemented on frequent actions, as the user will quickly grow tired of them. (Google, 2020)

Therefore, it was concluded that sound would be applied to rewarding actions, and potentially non-rewarding actions, e.g. answering a question incorrectly, to give the user a satisfying user experience.

3.3.4 Environment

Both the frontend and the backend of the application will be developed using Visual Studio Code. The reason for this is Visual Studio Code is a flexible development environment, with support for many extensions to aid the developer in building all sorts of applications.

For the frontend, the Dart & Flutter extensions were installed, which allow you to run, debug and build a Flutter app on either an Android Emulator, iOS Simulator, or a physical device directly from Visual Studio. It also auto-formats dart code and highlights errors in real-time before compiling the application.

While running the application in development, the extension also supports hot reloads. This is extraordinarily useful as changes to the code, the application will reload these changes in real-time.

For the backend, Visual Studio Code integrates very nicely with TypeScript. This results in a very smooth development experience. Visual Studio Code presents pre-defined code-snippets, IntelliSense code completion, specific information when you hover over functions and properties and type errors before the code is even compiled.

3.3.5 HiFi Prototype

A High-Fidelity prototype was created from the Wireframe shown in Chapter 3.3.1. It was built using Origami Design, built by Facebook.

The prototype is fully interactive allowing a flow through the application, which will become useful during the development of the application.

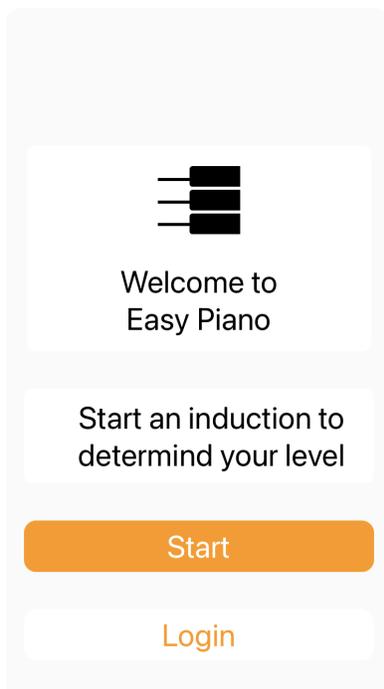


Fig 3.3.5.0 - Welcome Page

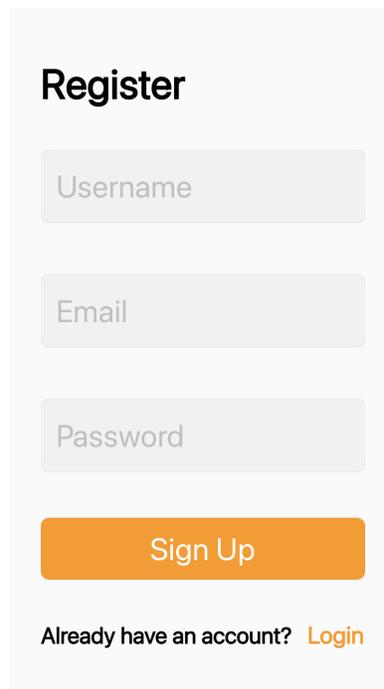


Fig 3.3.5.1 - Register Page

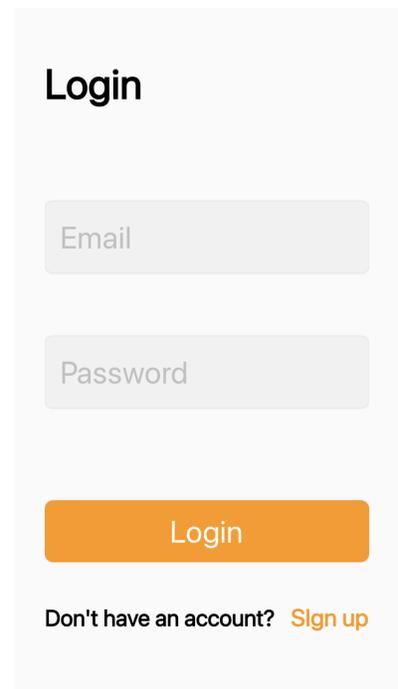


Fig 3.3.5.2 - Login Page

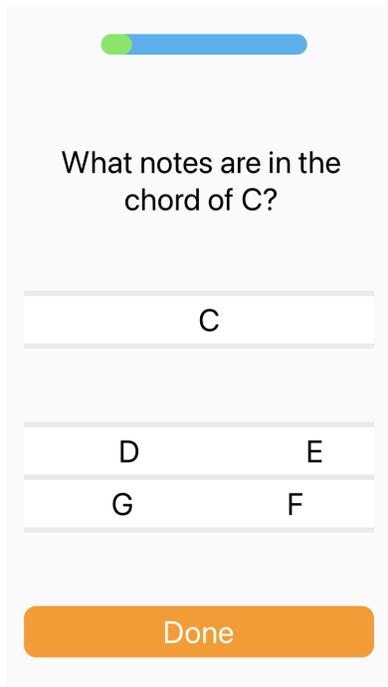


Fig 3.3.5.3 - Lesson / Induction 1

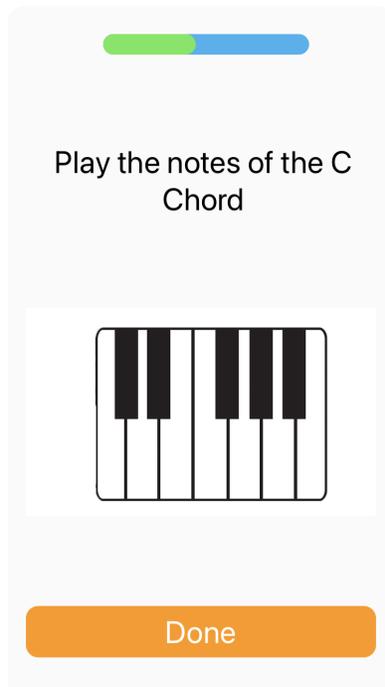


Fig 3.3.5.4 - Lesson / Induction 2

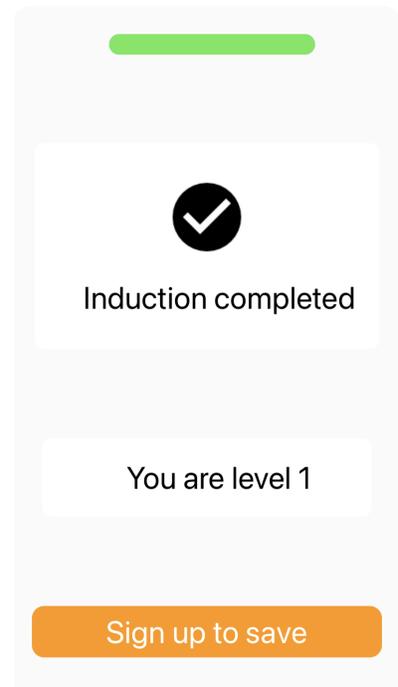


Fig 3.3.5.5 - Induction Complete

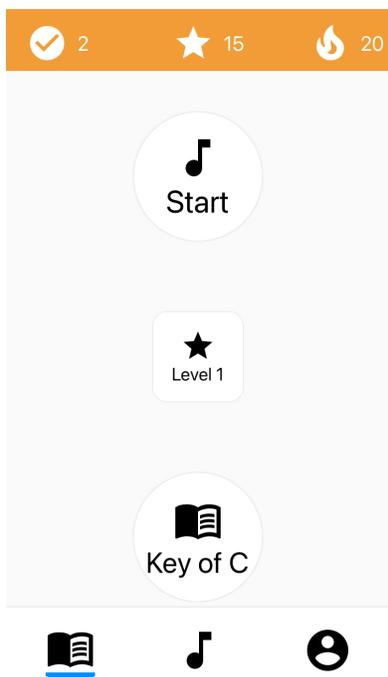


Fig 3.3.5.6 - Music Theory Screen

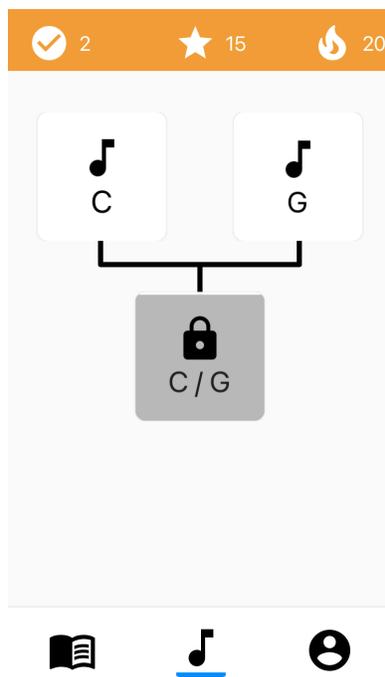


Fig 3.3.5.7 - Improv Screen

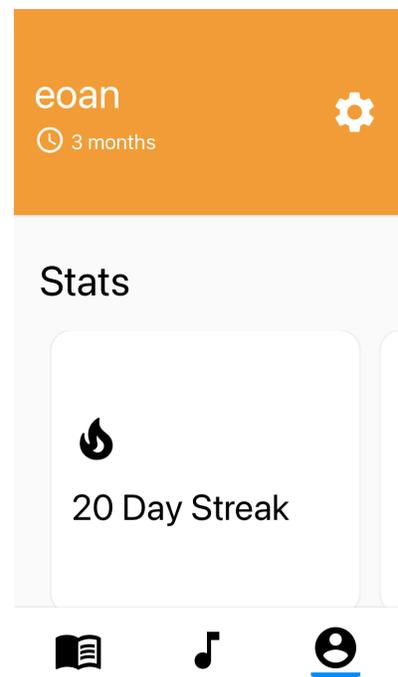


Fig 3.3.5.8 - Profile Screen

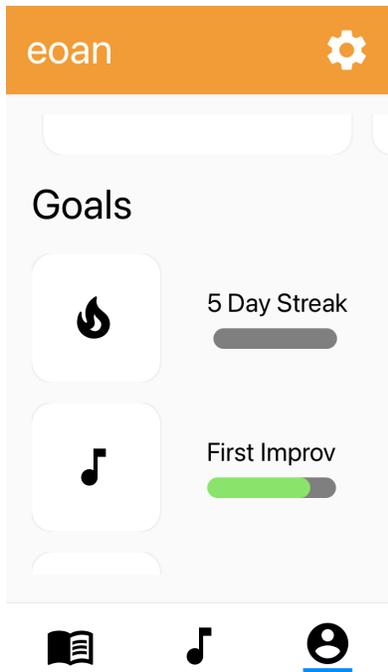


Fig 3.3.5.9 - Profile on Scroll

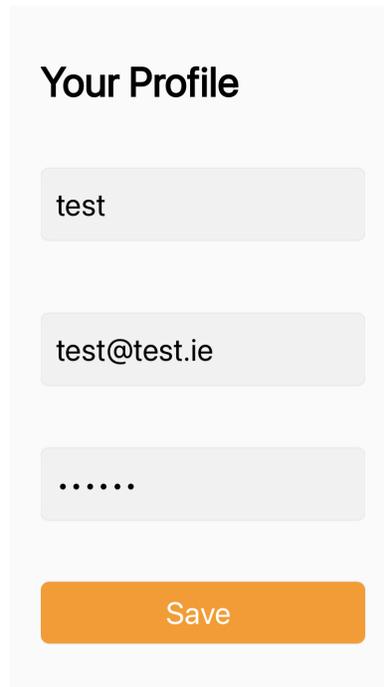


Fig 3.3.5.10 - Edit Profile

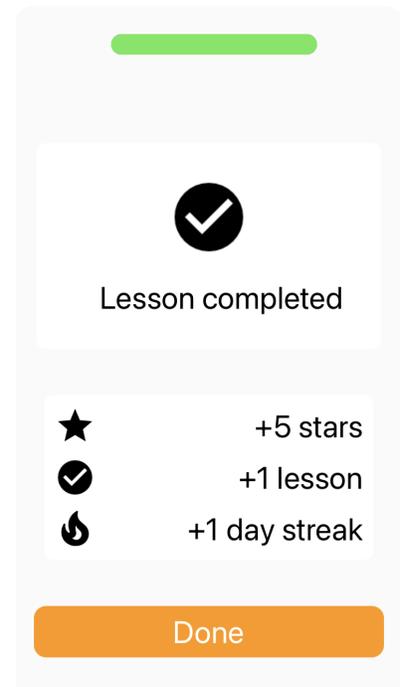


Fig 3.3.5.11 - Lesson Complete

The wireframe displays a full flow between the pages of the application. It begins by prompting the user to take an induction, to determine the level of the user. This will ask a series of questions before the user is prompted to register for the application.

Once completed, they will be prompted to register so their results will be saved. The user is then brought to the main screen, which presents three tabs:

- Theory Lessons
- Improvisation Lessons
- Profile

The theory lessons tab displays a list of lessons within a tree view. Once a certain amount of these are completed, the user will then be able to part-take in lessons in the improvisation tab.

On the profile tab, the user can see information on their current statistics, and goals that they can achieve by continuing with lessons. They also have the option to edit their profile by clicking the settings icon on the profile screen.

4.4 Content Design

The content of the application is particularly important because as mentioned in previous chapters, music theory is an extraordinarily dense subject. So the user must not be overwhelmed. The presented information must be easily digestible in small segments, and it is important not to have the goals too broad.

With this in mind, it was decided that the lessons would be broken up into:

1. Music Theory Lessons in a certain Key
2. Improvisation Lesson in a certain Key
3. Improvisation Lesson between two Keys

A user cannot part-take in an Improvisation lesson in a certain key before they complete the theory lesson for that key. This is done because the theory taught in this lesson is important and applicable to the subsequent improvisation lesson.

Once the user has completed a music theory lesson in C, for example, they will be able to do the improvisation lesson in that key. After this, the user can continue to complete a music theory lesson in another key, e.g. G. Once this is completed, and they complete the improvisation lesson in G, they will be able to do an Improvisation lesson between the two keys.

An Improvising lesson between two keys can only be completed once the user has completed the appropriate prerequisites; both the theory and improvisation lessons for both of those keys.

4.4.1 Music Theory Foundation Content

Before approaching any keys with the music theory lessons, the user will first need to complete the foundation lesson.

The foundation lesson will cover all the bases for things they need to know:

1. Describing a Stave
2. Clefs - Treble and Bass
3. The type of notes, the names and values
4. Time signatures
5. Counting and Rhythm

These lessons will involve asking users questions on these, with increasing difficulty each time. An example of this will be asking the user which notes on the stave this is:



Fig 4.4.1.1 - F letter on a Treble Clef Staff

The user will be asked which letter it is, what Clef it is on and the value of that note. Should they progress and answer questions correctly, the note name will be removed and they will be asked to show which note is where on a staff without seeing the letter below.

When it comes to rhythm, the user will be prompted to play (or clap) a single note along with the rhythm specified in a time signature. This will allow them to adapt to certain time signatures and will come to their advantage later on within the improvisation lessons.

4.4.2 Music Theory Key Content

Once the user has the foundations down, they will move on to a lesson in a particular key. This will start on the Key of C, but the intention here is to program this section within a dynamic structure, allowing additional keys to be easily added later on without having to add additional code to the project.

The structure of a key lesson will be as follows:

1. Teaching the scale of that key
2. The chord of that scale
3. Inversions of the chord

To teach the scale, it will begin with prompting the user in the structure of how a scale can be understood, and they can then be walked through a scale with the application. This means as they play a scale on the piano, the application would be able to tell the user which notes are played correctly.

The intention here has pitch detection functionality, which could understand the pitch of what note is being played by the user on their piano. It is also intended to present a "Can't Play right now" option, which will then show a piano on screen for the user, in case they are in a position where they cannot play the piano.

Once the scale has been taught, the subsequent stages should not be as difficult. The Chord consists of the 1st, 3rd and 5th notes in the scale, and a chord usually consists of three inversions, which entail playing the notes of the chord in a different order.

Root Position	C	E	G
First Inversion	E	G	C
Second Inversion	G	C	E

Table 4.4.2.0 - Chord inversions in C Major

Table 4.2.0 above displays different inversions in the C Major Chord. These are used to help smooth out motion from chord to chord, and can often have a slightly different sound than the root position chord.

4.4.3 Improvising

Once the student has learnt the foundations of music theory, and the appropriate scales and chords of a certain key, they can then unlock an improving lesson for that key.

Improvising like any other skill entails a large amount of practice before it can be executed smoothly. So it is important to take things very slow with the student.

They will first be presented with Chord Progressions in the key they are learning to improvise in, and will be taught the structure to follow when playing a chord progression. This is mentioned in the research chapter at 4.2.3.1 - Chord Progressions.

The student must have a basic understanding of this structure because the sounds resonate within a human brain as either sounding good or bad. An example of this is the Perfect Cadence, which has a very final sound to it, this is typically used to complete the progression, and a human brain would be happy to hear that. If the user was unaware of this and decided to finish on something seemingly random, the piece would sound less like music and more like a random sequence of notes being played.

With that in mind, the user will be walked through various progressions and will be taught various endings they may use to complete their pieces.

Once they become more familiar with this, the application aims to listen to the user as they play, and display various options of chords they can go to from the one they have just played, whilst keeping in time.

Continuing to practice like this would improve the user's knowledge of progressions and rhythmic ability. Both of which are essential to improve their skill of improvising.

4.4 Conclusion

This section goes through understanding the design of the application from the ground up. The application aims to educate users in a gamified manner, teaching them in small segments, which feel rewarding and satisfying for the user.

The program design describes the chosen technologies, and how they had to be changed from the requirements chapter due to additional research. It discusses the structure of the technology stack, and how the entire application will communicate. GraphQL is also discussed, including its advantages within such a large application. The different types of GraphQL methods are stated, described and a valid REST API comparison was given for ones that could be compared. The chapter then goes on to describe the design pattern of the application, and how it is similar to the MVC Design pattern in certain aspects.

The architecture of the application is discussed, and how the frontend, backend and database will communicate with each other. The server flexibility is also discussed, and its various modules including its endpoint, validation rules, resolvers and entities.

The design of the database was discussed, along with an entity-relationship diagram (ERD), which describes all data collections, and their relationships with one another. The flexible structure of GraphQL is described, and sample queries and provided to explain how efficient and accommodating it can be in many circumstances.

This chapter then goes on to describe the User Interface Design, which includes wireframes, User flow diagrams. This gives an idea of how the UI will look to the user, and how the general structure of the application will be.

After this, the style guide of the application was discussed, which includes the colours chosen, typography, shape, sound and how they follow the design guidelines specified by Material Design. Combining all of these, a HiFi prototype was created using Origami Studio, which will be used during the development of the application.

Finally, the content design was discussed, which goes through the structure of lessons, and the order in which certain lessons will be presented to the user. It is broken up into music theory lessons, improvisation lessons and finally an improvisation lesson between two keys.

The induction will determine how prepared a user is for these lessons by assigning them a level, and if they don't perform well in the induction, they will begin at the foundation music theory lesson, and work their way up to improvisation through various lessons.

5 Implementation

5.1 Introduction

This chapter discusses the implementation of individual sections within the application. It aims to resolve the problems proposed in the requirements, research and design chapters.

This chapter goes in-depth into each section of the project, which includes the server, the mobile application and the admin content management system. It also discusses the development environment, and how various development challenges were approached and resolved.

5.2 Development Environment

The project was developed using Visual Studio Code and various extensions to assist with the development of the project.

XCode was also used to run and test iOS builds, analysing build code and shipping them to TestFlight, which is Apple's BETA testing software. This will be discussed more in the deployment section.

5.2.1 Dart

A Dart extension was installed to extend the Dart programming language in Visual Studio. It provides tools for effectively editing, refactoring and running Dart code, along with an IntelliSense feature to display code and type errors before the developer runs the code.

This is particularly useful because the Dart language is type-safe. Meaning it uses a combination of static type checking to ensure the variables used always match the static types (Dart.dev, n.d.)

It is particularly useful to the developer that Dart can catch these type-safe errors before the application is even run, because developing mobile applications usually results in long compile times, so it is a much better developer experience to catch these errors before compiling the code at all.

5.2.2 Flutter

A Flutter extension was installed and used during the development of this project, allowing the developer to run, debug, and build the application directly from Visual Studio via either a Mobile Device Emulator or directly to a physical device.

This extension requires the Dart extension mentioned above as a prerequisite and offers a lot of additional functionality on top of Visual Studio's base functionality for rapid development of Flutter applications.

One feature, in particular, is the ability to trigger a hot reload while running the application in development. As mentioned above, compile times can quickly become a problem while building mobile applications, so Flutter allows you to quickly change colour for example in your code, and once you hit save it will appear immediately on the screen.

5.2.3 GraphQL for VSCode

Just like in Dart, GraphQL is strongly typed. Every entity defined needs an appropriate type to run. Although this results in more initial work to specify every detail, it pays off in the long run as the IDE can quickly spot any mistakes or type errors immediately after they are written.

The extension also offers validation, autocompletion, linting and syntax highlighting to improve the developer experience. (Harsh, 2021)

5.3 Database

The application uses MongoDB for its database, which is a NoSQL object-oriented scalable database. It stores documents in Binary JSON format (BSON) to increase efficiency and support more data types (mongoDB, n.d.-b).

The database is split up into sharded clusters, which splits read and write operations along with several nodes. The method of sharding can be compared to horizontal scaling, as it devises datasets and distributes the data over multiple servers.

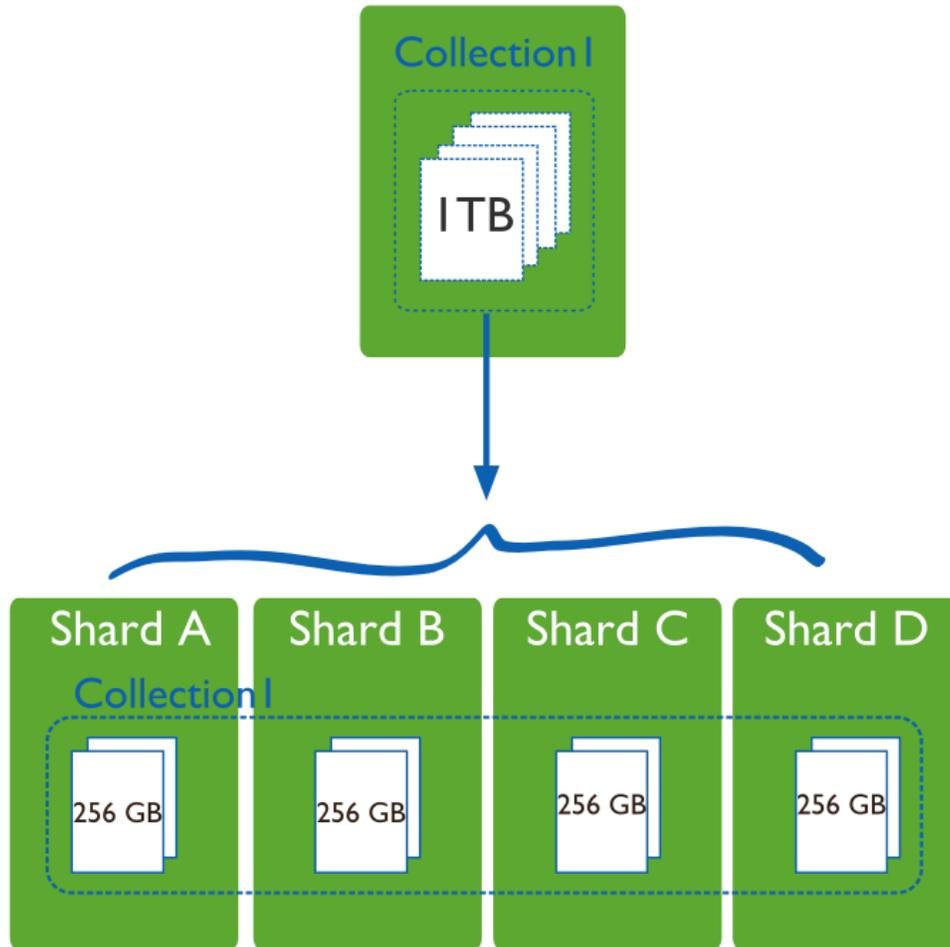


Fig 5.3.0.1 - Sharding in MongoDB (mongoDB, n.d.-a)

This can drastically reduce the overall CPU usage and RAM within a single machine, particularly on larger data sets which can quickly exhaust CPU when querying databases that store huge amounts of data. (mongoDB, n.d.-a)

As discussed in the design chapter, the database structure was defined by an ERD (Entity Relationship Diagram). There were several iterations of this diagram during the implementation of the project.

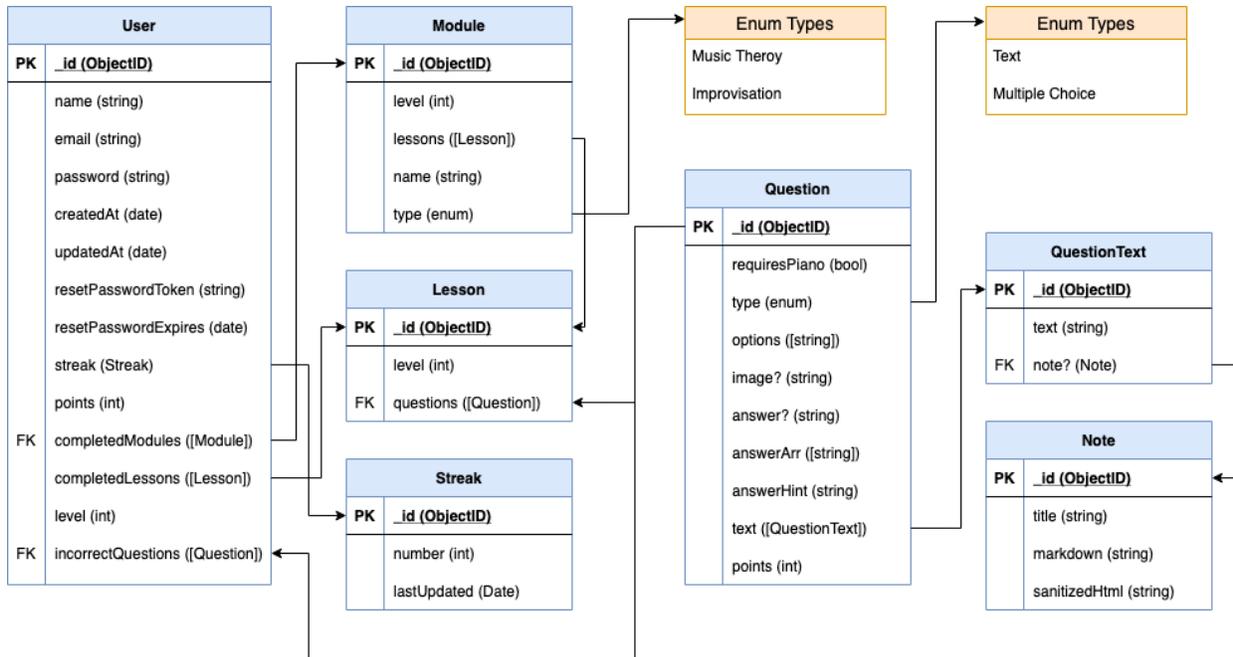


Fig 5.3.0.2 - Entity Relationship Diagram V9

The design chapter contained V5 of the ERD, but throughout the process of implementation, several iterations were made due to the increasing complexity of the server.

The relationships between these collections will be discussed in more detail in the Backend section.

5.4 Backend

The application's server is built using the following technologies:

- Node.js
- Apollo-Express
- Mikro ORM
- Type-GraphQL
- GraphQL
- TypeScript

The server contains several endpoints during a production build, and during development, it opens up an additional endpoint a GraphQL playground.

4.4.1 Serving Static Assets

The server has a static express directory which is where all images are served to the client. These images would typically be used within a certain question.

The static function is a built-in middleware to express. It transforms a particular directory into a static directory, which is defined as a directory that contains any content that can be delivered to an end-user without having to be compiled, modified or generated. (Gibb, 2016)

There are several benefits to serving static content, the content will not change which will assist in caching on the user's end devices. Serving images is one of the more power-hungry tasks for a server to do.

An image can often be much bigger than any amount of data requested by the frontend. Serving these images statically also means the server doesn't require a layer of logic to run before serving an image, it only needs to pull a file from the disk.

A particular question in the database would contain an optional field called "image". If it exists, it would fetch this asset from the server once the question is displayed on the front end. This can be seen in Fig 3.0.2 above under the Question entity.

4.4.2 Structure

The backend has been structured into small individual components. The goal here is to keep each file minimal, to only execute functions related to the folder it is in.

As the codebase grows, this becomes increasingly important with debugging and maintenance.

```
/music-theory-backend/src
|-contracts
|---validators
|----enums
|-entities
|-middleware
|-resolvers
|-utils
|---interfaces
```

Code Block 4.4.2.0 - Tree view of the backend folder structure

Code Block 4.4.2.0 above displays the structure of the server. It is divided into various subdirectories in an attempt to maintain order within the server. For example, the resolvers directory above, contains a resolver for each collection within the database - e.g. user, lesson, question etc.

Each directory will contain an index.ts file, its only job is to import every other file within that directory, and export all modules from those files. This is shown in Code block 5.2.1 below;

```
export * from "./user.resolver";
export * from "./lesson.resolver";
export * from "./auth.resolver";
export * from "./question.resolver";
export * from "./questionText.resolver";
export * from "./module.resolver";
export * from "./note.resolver";
```

Code Block 4.4.2.1 - src/resolvers/index.ts - exporting modules within index.ts

Although this makes no change in terms of functionality, it cleans up any necessary imports on other files, so all of the modules can be imported within the same statement, as shown in Code Block 4.4.2.2 below;

```
/**
 * Resolver modules
 */
import {
  UserResolver,
  LessonResolver,
  AuthResolver,
  QuestionResolver,
  ModuleResolver,
  QuestionTextResolver,
  NoteResolver,
} from "./resolvers";
```

Code Block 4.4.2.2 - src/application.ts - importing modules in one statement

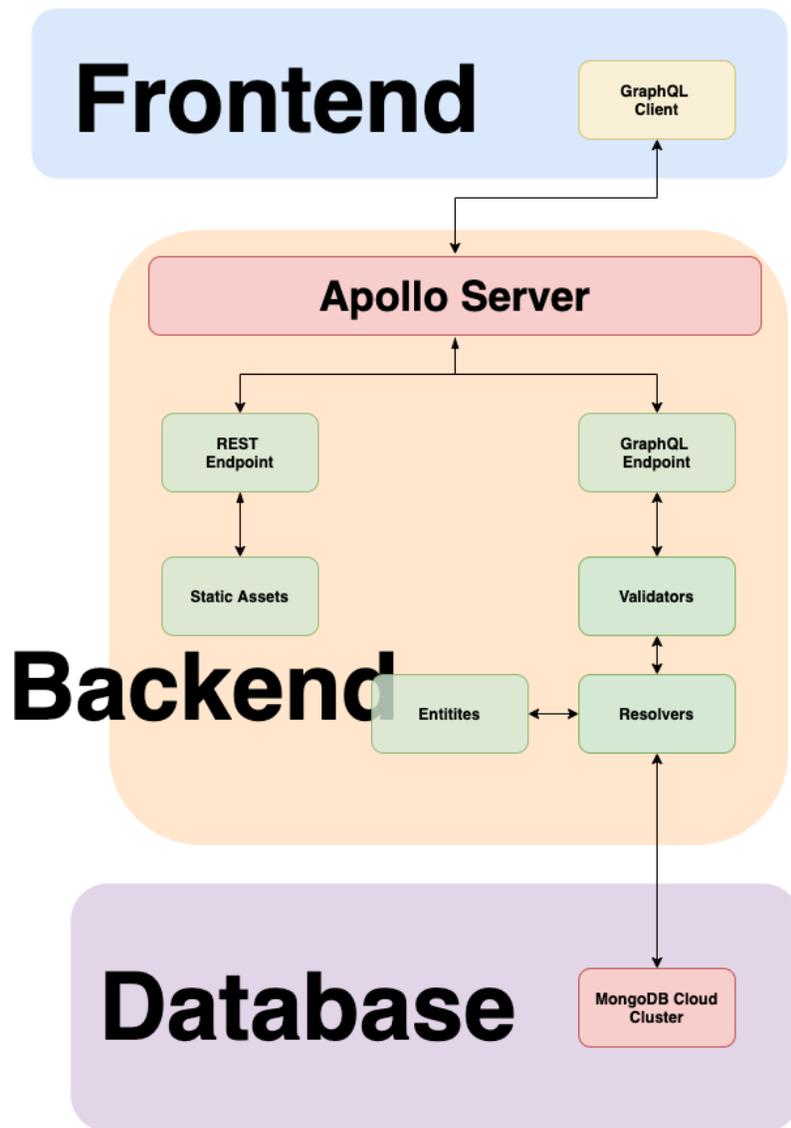


Fig 4.4.2.0 - Backend Design Pattern V2 - built using draw.io

Fig 4.4.2.0 shows a detailed architecture of the backend and the general flow of data throughout it. The Apollo Server acts as the entry point for the server, which is a community-maintained open-source GraphQL server. (apollographql, 2021)

The server will run from the application.ts file, connect to the database, register and process all entities, and wait for any incoming requests.

4.4.3 Validation

As shown in Fig 4.4.2.0 above, any requests that hit the GraphQL endpoint, will go through the process of being validated, and then will execute some business logic within the resolver.

The validator will ensure every piece of data sent along with a request is exactly what it is supposed to be.

```
@InputType()
export class AuthValidator {
  @Field()
  @IsEmail()
  @IsNotEmpty()
  public email: string;

  @Field()
  @IsString()
  @IsNotEmpty()
  @MinLength(6, {
    message: "Password must be at least 6 characters",
  })
  public password: string;
}
```

Code block 4.4.3.0 - ./contracts/validators/auth.validator.ts

Take Code block 4.4.3.0 as an example, this validator handles authentication when a user wants to sign in. It requires two fields, an email, which must be a string, and a password, which also must be a string, but also has to be at least 6 characters.

Should a user attempt to send anything to the server that is not within the confinements, the validator will reject the data, and return an error.

This keeps security tight within the server and prevents any type of NoSQL injections.

4.4.4 Resolvers

A Resolver is defined as a collection of functions that generate responses for a GraphQL query (tutorialspoint, n.d.). It handles any business logic and queries with the database.

Code block 4.4.5.0 below contains a single mutation (the other functions were removed for simplicity). This mutation expects input from the client, which uses the AuthValidator

to validate it, as mentioned in section 4.4.3 above. Once validation is passed, it attempts to find a particular user by the provided email address.

If a user is found, it will run a verify function provided by the Argon2 library, which allows us to compare the plaintext password provided by the user in the input, with the encrypted password currently stored in the database.

Once verified, an authentication JWT (JSON Web Token) is created, along with an expiration date of when the token will expire. These are both sent back to the client to be handled.

```
@Resolver(() => User)
export class AuthResolver {
  @Mutation(() => LoggedIn, { nullable: true })
  async login(
    @Arg("input") input: AuthValidator,
    @Ctx() ctx: MyContext
  ): Promise<{ token: string; expiration: number; user: User }> {
    try {
      const user = await ctx.em.findOneOrFail(User, { email: input.email });

      if (await verify(user.password, input.password)) {
        const { token, expiration } = generateToken(user.id);

        return { token, expiration, user };
      }

      throw new ClientSafeError("Incorrect Password", 403, "AUTH_ERROR");
    } catch (err) {
      throw new ClientSafeError(
        "Incorrect Email or Password",
        403,
        "AUTH_ERROR"
      );
    }
  }
}
```

Code block 4.4.5.0 - ./src/resolvers/auth.resolver.ts - the auth resolver and login mutation

Should the user make any subsequent requests that require authentication, the token will be attached to the request and is verified within the request middleware.

The server can decipher the user's unique ID from this token, so the server can understand who is making the request, and respond with relevant information.

```

@Query(() => User, { nullable: true })
async get(
  @Ctx() ctx: MyContext,
  @Info() info: GraphQLResolveInfo
): Promise<User> {
  try {
    const user = await ctx.em
      .getRepository(User)
      .findOneOrFail({ id: ctx.auth._id }, [
        "completedModules",
        "completedLessons",
        "incorrectQuestions",
      ]);

    return user;
  } catch (err) {
    throw new ClientSafeError("Not Authenticated", 403, "AUTH_ERROR");
  }
}

```

Code block 4.4.5.1 - ./src/resolvers/auth.resolver.ts - the get query

The Query in Code block 4.4.5.1 above is used to get the current user. It requires an authentication token, which has already been verified before this function is run. Should it be verified successfully, the server can find the user by ID.

If the token is not verified or is expired, the server will return a 403 Authentication Error.

4.4.5 Entities

An Entity is defined as a simple object type that is defined canonically in one implementing service (apollographql, n.d.). It represents a single collection within the database.

```

@ObjectType({ description: "Represents a user object" })
@Entity()
export class User extends Base<User> {
  @Field()
  @Property()
  @Unique()
  public name: string;

  @Field()
  @Property()
  @Unique()
  public email: string;
}

```

```

@property()
password: string;

@Field(() => [Question])
@OneToMany(() => Question, (b: Question) => b.user, {
  cascade: [Cascade.ALL],
})
public incorrectQuestions = new Collection<Question>(this);

constructor(body: UserValidator) {
  super(body);
}
}

```

Code block 4.4.5.1 - ./src/entities/user.entity.ts - The user entity

Code block 4.4.5.1 above is an example of an entity within the application. A large amount of code has been removed for simplicity. Whenever a new user is created, it will inherit all of its properties from this class.

This entity also inherits a Base entity, which defines timestamps and a primary key to the entity. This means it will have createdAt and updatedAt fields, and the updatedAt field will change every time a change is made on the entity itself.

This is done within a mutation when a user would like to register for the application.

```

@Mutation(() => User)
public async addUser(
  @Arg("input") input: UserValidator,
  @Ctx() ctx: MyContext
): Promise<User> {
  try {
    const user = new User(input);

    const hashedPassword = await hash(input.password);
    user.password = hashedPassword;

    await ctx.em.persist(user).flush();

    return user;
  } catch (err) {
    console.log("Error creating user", err);
    throw err;
  }
}

```

Code block 4.4.5.2 - ./src/resolvers/user.resolver.ts - Adding a new user within the addUser mutation

As shown in Code block 4.4.5.2 above, a new user is created using the validated input. Their password is then encrypted, followed by the newly created user being persisted and flushed.

Persisting and flushing are built-in functions with Mikro ORM, which manages the entities within the server.

The persist function is used to mark a new entity for future persisting, and the flush function is what then writes it to the database.

Another thing to notice within Code block 4.4.5.1 above, is the OneToMany relationship the User entity has with the Question entity. The incorrectQuestion field within the User object is a collection of questions that this particular user has answered incorrectly.

This is constantly updated as the user goes through various lessons, and the user will later be presented with these questions on subsequent lessons. If the user answers them correctly, they will be removed from the collection, if not they will remain there until it is answered correctly.

4.4.6 Mikro ORM

Mikro ORM is a TypeScript data-mapper Object-relational mapper for Node.js. It supports multiple drivers including MongoDB, MySQL, PostgreSQL and SQLite. It is particularly useful for both performance and the usual of GraphQL (mikroORM, n.d.).

The usage of Mikro ORM's persisting and flushing operations drastically aided the performance of the server. Persisting something means it will eventually need to be written to the database, but instead of being written in a single query, persistence will add multiple database queries in one, to reduce the overall haul of database transactions.

This is particularly useful when multiple database transactions must occur within single function, for example in Code block 4.4.6.0 below:

```
@Mutation(() => Question)
public async addQuestion(
  @Arg("input") input: QuestionValidator,
  @Arg("lessonId") id: string,
  @Ctx() ctx: MyContext,
  @Info() info: GraphQLResolveInfo
```

```

): Promise<Question> {
  const question = new Question(input);

  try {
    question.lesson = await ctx.em
      .getRepository(Lesson)
      .findOneOrFail({ id });

    ctx.em.persist(question);

    if (input.text.length > 0) {
      for (const text of input.text) {
        const newQuestionText = new QuestionText(text);

        if (text.note) {
          const note = await ctx.em
            .getRepository(Note)
            .findOneOrFail(text.note);
          newQuestionText.note = note;
        }
        newQuestionText.question = question;
        ctx.em.persist(newQuestionText);
      }
    }

    await ctx.em.flush();

    return question;
  } catch (err) {
    // Handle error
  }
}

```

Code block 4.4.6.0 - ./src/resolvers/question.resolver.ts - Adding a new question within the addQuestion mutation

Code block 4.4.6.0 above calls the persist function multiple times. Once when we assign a lesson to the question, and subsequently within the loop that checks the number of text items within the `input.text` array.

On an average question created, this would result in approximately six writes to the database. But because Mikro ORM is used, and the flush function is called at the very end of the addQuestion function, this results in all database operations combined into one.

On a small scale, this might not seem significant, however as the user base grows, and simultaneous database write operations are required amongst multiple users, this will drastically aid the performance of the server and database.

Using Mikro ORM is also beneficial for reducing duplicated code, as discussed in section 2.1 of the design chapter, the Mikro ORM entity is also used as a GraphQL schema, allowing the types for the entity to be defined once, and to be applied in multiple places.

4.4.7 GraphQL

As mentioned in section 2.5 of the Design chapter, GraphQL is very useful for specifying exactly how much or how little data required at a single point in time. Compared to traditional RestfulAPI's, which requires the application to make subsequent requests for data, a typical GraphQL query can have multiple requests of data combined into a single query.

GraphQL also has a type system, which has every entity defined already. Just like with using TypeScript, this is extraordinarily beneficial to the developer as they encounter errors before the application is even run or built.

It also offers a full playground when running the server in development mode, which allows to build and test queries, mutations and subscriptions with the server without having to have a client built and ready.

The declarative model provided by GraphQL allows the creation of a consistent, predictable API across all clients. Which is not only useful when building the frontend in Flutter, but also useful for creating an Admin Dashboard using React.js (apollographql, 2015).

The server can be queried in entirely different ways from either client and deploy those clients quickly. This is particularly useful for the admin dashboard, which will be used to continuously add modules, lessons and questions to the server.

4.4.8 Deployment

The server is deployed to Heroku, which is a developer-focused cloud Platform as a Service (PaaS). The reason why it was chosen is that it is used by developers to quickly deploy scalable applications with eases of use (Heroku, 2020).

Heroku has a command-line interface (CLI) that allows you to quickly deploy a production build. This was used frequently to deploy once the application was at a deployment testing state.

A build script was implemented on the package.json of the server, which compiles all of the TypeScript code into minified JavaScript, and runs it from the dist folder. A Procfile was also created to tell Heroku how to run the application.

```
web: node dist/src/index.js
```

Code block 4.4.8.0 - ./Procfile - The Procfile informing Heroku how to run the server

The build script runs a few prerequisite checks before a successful build is done, this script was written in the package.json, as shown in Code block 4.4.8.1 below:

```
//...  
"scripts": {  
  "prebuild": "tslint -c tslint.json -p tsconfig.json --fix",  
  "build": "tsc",  
  "prestart": "npm run build",  
  "start": "cross-env NODE_PATH=./dist/src node dist/src/index.js"  
},  
//...
```

Code block 4.4.8.1 - ./package.json - The package.json build scripts

As shown in the above Code block, four scripts are written here, the rest of the code in this file has been removed for the sake of simplicity. These four scripts are chained together using the scripts prefixed with the word “pre”. So when the “start” script is run, it first runs the “prestart”, which runs the “build” script, which first runs the “prebuild”. This is clarified in the below table:

Script	Prerequisite	Corequisite
start	prestart	Runs the production build
prestart	build	
build	prebuild	Builds the application
prebuild		Runs a static analysis on the codebase to catch errors early

Table 4.4.8.0 - Explaining the scripts and prerequisite scripts

Table 4.4.8.0 above explains the scripts and the order in which they are all called. The pre-build command, in particular, is very useful, as it will run a static analysis on the

entire code base using TSLint. It checks TypeScript code for readability, maintainability and functionality errors (Microsoft, 2015).

This is one of the large benefits of using TypeScript because the built-in powerful type system prevents difficult issues and bugs during runtime. The Type system catches them as the developer writes the code, and anything missed by the developer at the point in time will be caught by the pre-build script above.

Once the application has been successfully built locally, it can be deployed to Heroku using the CLI. Heroku will run a full build, optimize it, remove and develop dependencies that are no longer required and run the application within an isolated Linux container, which they call dynos.

The one downside to Heroku when you run an application on their Hobby plan (free plan), they will put your server to sleep whenever it has not been used for over approximately 30 minutes.

This means if somebody were to open the application and attempt to log in when the server hasn't been used for a while, they will have to wait for approximately fifteen to twenty-five seconds for the server to wake up and run before it returns to its normal operating state.

5.5 Frontend

The front end of the application is built in Flutter, along with Material UI as a component Library. Flutter is an open-source mobile SDK that can be used to build native performing applications for both Android and iOS from the same codebase (Software, 2019). It is based on the Dart programming language.

As discussed in the design chapter, it was chosen for its flexibility, performance and ease of use for the developer. Material UI was chosen as the component library because it is already built into the Flutter SDK, so no additional dependencies were required.

The application's frontend utilises the following dependencies:

- Flutter_midi
- GraphQL_flutter
- Flutter_html
- shared_preferences
- Multi_select_flutter
- Google Fonts

5.5.1 Structure

Similar to the backend, the frontend was intentionally structured into small components to keep the code maintainable and organized, particularly as the codebase grows.

```
/music-theory-frontend/lib
|-src
|---Widget
|---components
|----question
|---config
|---data
|---model
|---screens
|----lessons
|----tabs
|-----dashboard
|---services
```

Code Block 5.5.1.0 - Tree view of the frontend folder structure

Code Block 5.5.1.0 above displays the structure of the frontend, and how it is divided up into its various subdirectories depending on the components. The main entry point for the application is the main.dart file.

This file registers Material UI and its associated theme, which defines the typography using Google Fonts, the colours, and the primary routing for the application. It also registers GraphQL by wrapping the entire application in a GraphQLProvider.

5.5.2 Authentication & Routing

The main.dart initially displays a Splash screen, which displays a progress indicator to the user. While this is happening, the application checks if an authentication token exists, and if it is not expired, it will allow the user to skip the login process.

If no token exists, or the authentication token is expired, the user will need to either login or register to gain access to the rest of the application.

```
// ...
home: Splash(
  redirect: redirect,
),
routes: {
  '/welcome': (context) => WelcomePage(),
  '/login': (context) => LoginPage(),
  '/register': (context) => RegisterPage(),
  '/dashboard': (BuildContext context) => Dashboard(),
},
// ...
```

Code Block 5.5.1.1 - ./lib/main.dart - Routing within the main.dart

As shown in Code Block 5.5.1.1 above, the initial screen is the Splash screen, and it passes in a redirect as a parameter. The redirect is a string that will redirect the user to either the “/welcome” or “/dashboard” route within the routes object above, depending on if the user is authenticated or not.

The authentication token is stored within the user’s device, using the `shared_preferences` package, which persists the token to the user’s device asynchronously. This means the user can close the application and reopen it, and remain logged in provided the token is valid.

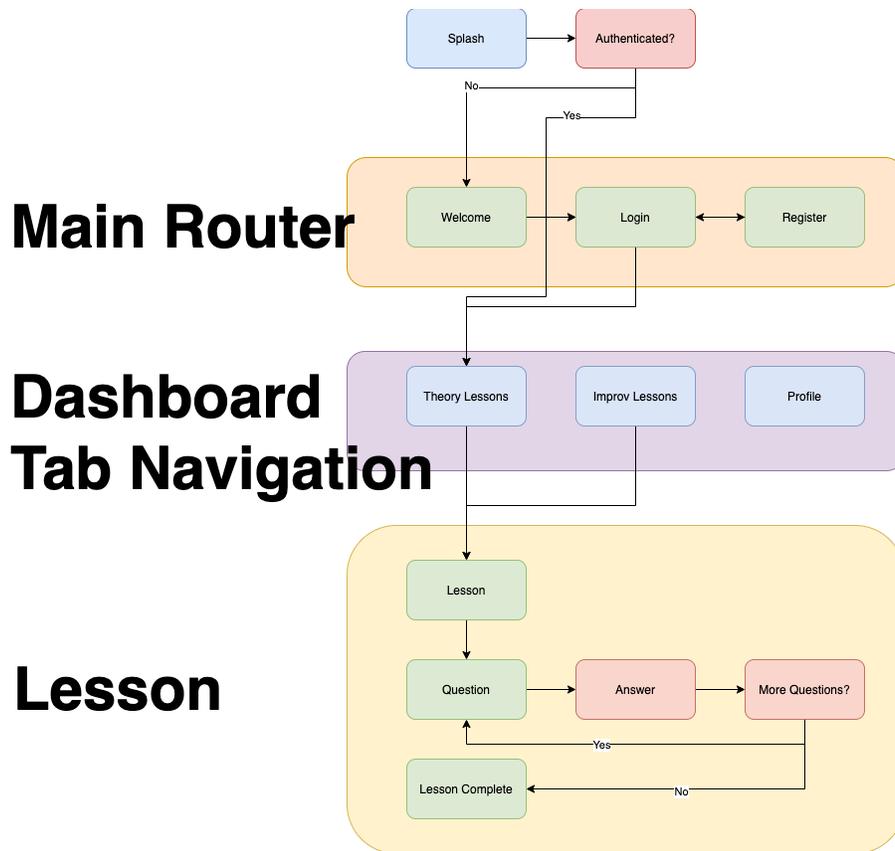


Fig 5.5.1.0 - Frontend Router Flow Chart V3

Fig 5.5.1.0 above contains a breakdown of the general routing flow throughout the entire application. It is divided into three sections:

1. Main Router
2. Dashboard Tab Navigator
3. Lesson Navigator

The main router handles the authentication for a user as described above. The initial authentication check will determine whether the user is sent to the welcome screen, or directly to the Dashboard Tab Navigator.

The Dashboard Tab Navigator allows the user to navigate between tabs, and the user is presented with a bottom tab bar to allow them to easily switch between the theory lessons, improv lessons and their profile page. The latter will display various pieces of information about their profile.

Once a user begins a lesson from either the theory or Improv screens, they are brought to the Lesson Navigator. This navigator will cycle through the various questions fetched from the server, and once all questions have been answered, will display a lesson

complete screen, to show the user how they performed in the lesson, and how many points they earned.

5.5.3 Models

Data that is fetched from the server is served in JSON (JavaScript Object Notation) format. Dart cannot understand this format, and since Dart is a strongly typed language, models need to be created for all types of dynamic data being utilized on the frontend.

These models were automatically generated using a tool called [quicktype](#), which allowed the developer to paste in a JSON response, and it automatically generated the entire model in a strongly typed language of your choice.

Code block 5.5.3.0 below represents the Module model within the frontend:

```
class ModuleItem {
  ModuleItem({
    this.id,
    this.title,
    this.level,
    this.type,
    this.lessons,
  });

  String id;
  String title;
  int level;
  String type;
  List<LessonItem> lessons;

  factory ModuleItem.fromJson(Map<String, dynamic> json) => ModuleItem(
    id: json["id"],
    title: json["title"],
    level: json["level"],
    type: json["type"],
    lessons: json["lessons"] == null
      ? null
      : List<LessonItem>.from(
        json["lessons"].map((x) => LessonItem.fromJson(x))),
  );

  Map<String, dynamic> toJson() => {
    "id": id,
    "title": title,
    "level": level,
    "type": type,
    "lessons": lessons == null
      ? null
```

```
      : List<dynamic>.from(lessons.map((x) => x.toJson())),  
    };  
  }
```

Code Block 5.5.3.0 - ./lib/src/model/ModuleItem.dart - The Module Model

The module model contains a constructor, showing all data that will be contained within a module, and two functions; fromJson and toJson. These functions convert data either from or to JSON format.

Converting data served from the server into a typed object like this gives the data more structure, and results in it being easier to maintain. This is also considered common practice in strongly typed languages like Dart (Mackier, 2019).

5.5.4 GraphQL

This application utilizes the flutter_graphql dependency, which allows it to make graphql requests to the server. To do this, the entire application needed to be wrapped in a GraphQLProvider, which is initialised GraphQL in the application entry point.

The initialize function is defined within a Config class, which is where the server URL is defined, where the connection to the server is initialized and passes the authentication token into all requests. It also is where the application is set up to handle web socket requests, but this was not fully implemented into the application.

After initialization, any graphql query or mutation requests can be made from any components that are called within the GraphQL Provider, which in this case, is all components.

5.5.4.1 Querying data from the server

Querying the server is how dynamic data is displayed on the application. An example of this is the theory page, which displays a list of theory related modules that are queried from the server.

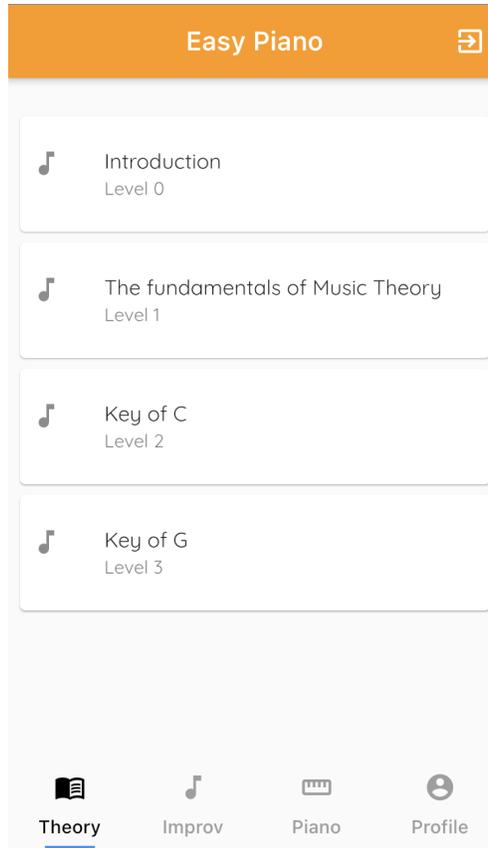


Fig 5.5.4.1.0 - The Theory tab displaying modules from the server - iOS build 1.1.1

These modules are queried from the server and are cached on the user's device to reduce subsequent network requests.

```
class _TheoryModulesState extends State<TheoryModules> {  
  @override  
  Widget build(BuildContext context) {  
    return Query(  
      options: QueryOptions(  
        // Query defined in /src/data/Module.dart  
        documentNode: gql(Module.getModules),  
        // Parameters to the query  
        variables: {"type": "THEORY"},  
      ),  
      builder: (QueryResult result,  
        {VoidCallback refetch, FetchMore fetchMore}) {  
        // Handle errors  
        if (result.hasException) {  
          return EmptyState(message: result.exception.toString());  
        }  
        // Handle loading  
        if (result.loading) {  
          return Center(child: CircularProgressIndicator());  
        }  
      }  
    );  
  }  
}
```

```

    }
    final List<LazyCacheMap> items =
      (result.data['getModules'] as List<dynamic>).cast<LazyCacheMap>();
    if (items.length == 0)
      return EmptyState(message: 'No Theory Modules Found');
    return ListView.builder(
      // Render Modules in a list
    );
  },
);
}
}

```

Code Block 5.5.4.1.0 - ./lib/src/screens/dashboard/TheoryModules.dart - A simplified version of the TheoryModule

As shown in code block 5.5.4.1.0 above, the entire component is wrapped in a Query, which is specified in the QueryOptions. Here we have specified the getModules query, which gets a module by its type.

```

class Module {
  static String getModules = ""
  query getModules(\$type: String!) {
    getModules(type: \$type) {
      id
      title
      level
    }
  }
}
"";

// ...
}

```

Code Block 5.5.4.1.1 - ./lib/src/data/Module.dart - The getModules graphql query

The getModules query will take the module-type parameter, and run it. The rest of the component is set up to either receive the data, and render it into a list of modules, display a loading indicator to the user, or handle an error and allow the user to recover from it.

Any errors throughout the application are handled using the EmptyState component. This component is designed to take a string and an action. It will display the string to the user e.g. "Error: Could not fetch modules", and an action, which will redirect the user back to the homepage. The action is optional, and if not provided, will not display action to the user.

Once the query successfully fetches modules, it will structure them using the data model explained in the previous chapter and will display the data in a list format, as shown in Fig 5.5.4.1.1 above.

5.5.4.2 Running Mutations to the server

A mutation is an action to post data to the server. It usually results in some sort of modification of data. An example of this is when the user completes a lesson.

```
class Lesson {  
  // ..  
  static String completeLesson = ""  
    mutation(\$lessonId: String!) {  
      completeLesson(lessonId: \$lessonId) {  
        id  
      }  
    }  
  "";  
}
```

Code Block 5.5.4.2.0 - ./lib/src/data/Lesson.dart - The completeLesson mutation

Code block 5.5.4.2.0 above displays the mutation that runs when a user completes a lesson. It requires the lessons' unique identifier as a parameter, and in response returns that same ID. No additional data is required on the frontend since the application already has all the data it needs, but it is mainly to tell the server that this user has finished this lesson.

Each user has a streak system attached to their account. This tracks the number of consecutive days that they participate in lessons. When a lesson is completed, the user's streak is incremented by 1, if their last lesson was within 24 hours. The server also keeps track of which lessons a user has completed, which would become useful later if some sort of rewards system was implemented into the application.

5.5.5 Services

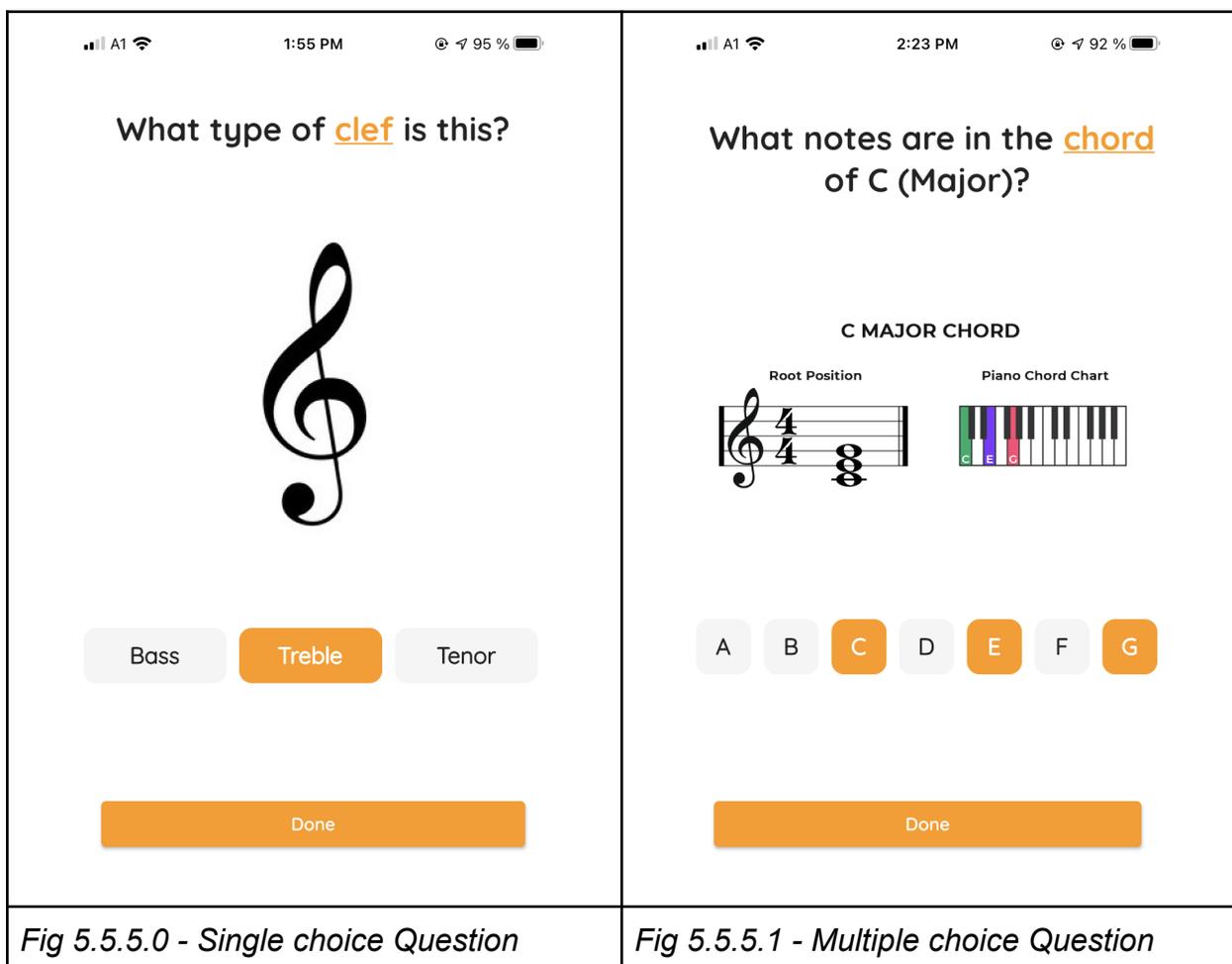
There are two services implemented into the application; The SharedPreferencesService, which manages local device storage, and the MidiUtils service, which handles the sound for playing Piano notes.

The SharedPreferencesService is used primarily for storing the authentication token, and expiration date of the token within the end user's device. This is used to allow the user to remain logged in even after closing the application.

The MidiUtils service is used to handle all Midi controls. It prepares a SoundFont file, which contains all 88 notes of the piano. After the SoundFont file has been prepared, this service is responsible for playing, pausing and unmuting.

5.5.5 Lesson

A lesson comprises a collection of questions, which is handled by the QuestionController. When a user starts a lesson, it cycles through each of these questions and renders them depending on the Question type, which is either a multiple-choice answer or a single choice answer.



As shown by Figs 5.5.5.0 & 5.5.5.1, the frontend will render the answer options based on the type of question it is given. Fig 5.5.5.0 is a single choice question, so the user can only select a single answer. But Fig 5.5.5.1 gives the user the option to select multiple options.

Multiple choice answers also play the piano note associated with the letterpressed. For example, by selecting the C option, the user will hear the C note being played on their device.

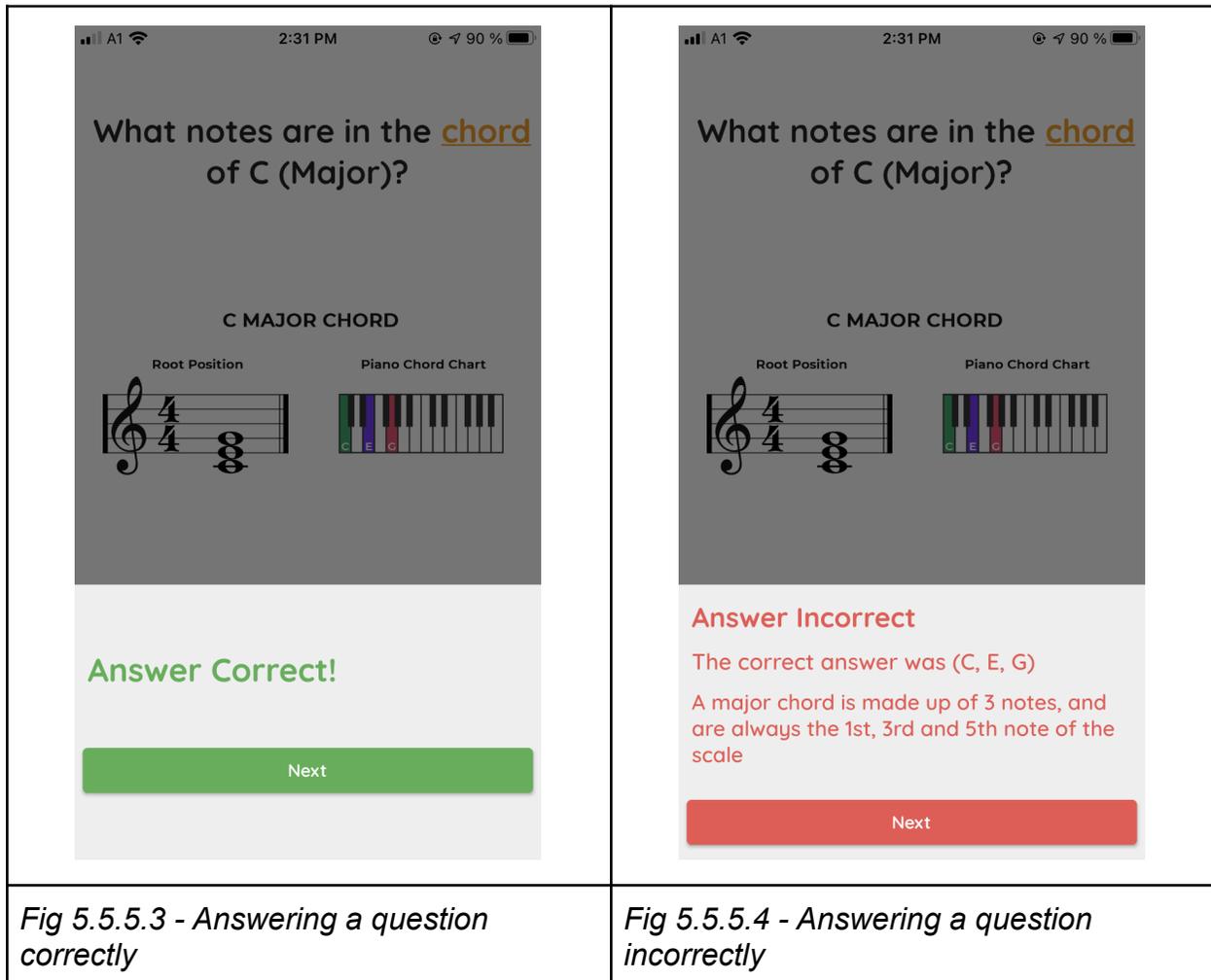
Particular words in a question will be highlighted in orange and underlined. These words have notes attached to them. The goal is if the user does not know the answer to the question, they can tap the highlighted word, and read a note that contains the answer to the question.



Fig 5.5.5.2 - Selecting the word Clef in FIG 5.5.5.0 displays the Clef note

The note is inputted in markdown, which is converted to sanitized HTML on the server when created. The frontend renders this HTML, and the images are fetched from the server as soon as the note has appeared on the screen.

If a user answers a question incorrectly, the QuestionController will keep track of this, and send a notification to the server. The server confirms the answer against the question-answer and saves the question to the user's account. When the user starts a subsequent lesson, they will be asked the incorrectly answered questions first, before being asked any new questions.



As shown in Fig 5.5.5.4, the user is shown what the correct answer was, and is also given a hint on the question. This is useful for the next time they encounter this question, they may answer and learn from their previous mistake and answer differently.



Lesson Complete

You answered 1 correct out of 2

Done

Fig 5.5.5.5 - Lesson Complete Screen

At the end of a lesson, the user is shown their full score, and the number seven-step of points they earned for that lesson.

5.5.6 Deployment

Due to the nature of Flutter, the application can be built and deployed to both iOS and Android devices from the same code base. This is extraordinarily beneficial as it drastically reduces the workload, compared to building a native application for both iOS and Android. The performance is also close to native on each device, which will be discussed in the testing chapter.

The application was deployed to Apple's TestFlight service, which allows BETA test applications before launching to the App Store. From here it was possible to grant users access to the application and test it on their device.

The process for building, validating and reviewing the application for TestFlight is very similar to publishing to the iOS App Store, so if the development process were to continue, the majority of deployment work has already been done.

The application was also deployed to Google's BETA testing system, via the Android Play Console. From here internal testing could take place where the Application could be downloaded to Android user's devices to use the application, and see how it performed.

Similar to Apple's TestFlight service, building for Android Play is similar to launching the Android Play store, so the majority of deployment work has been done should the application make it to production.

5.6 Admin Dashboard

The applications front end is designed from the ground up to be completely dynamic. This means any content added to the server will reflect in the app as new content, without having to ship another update through the iOS / Android App stores.

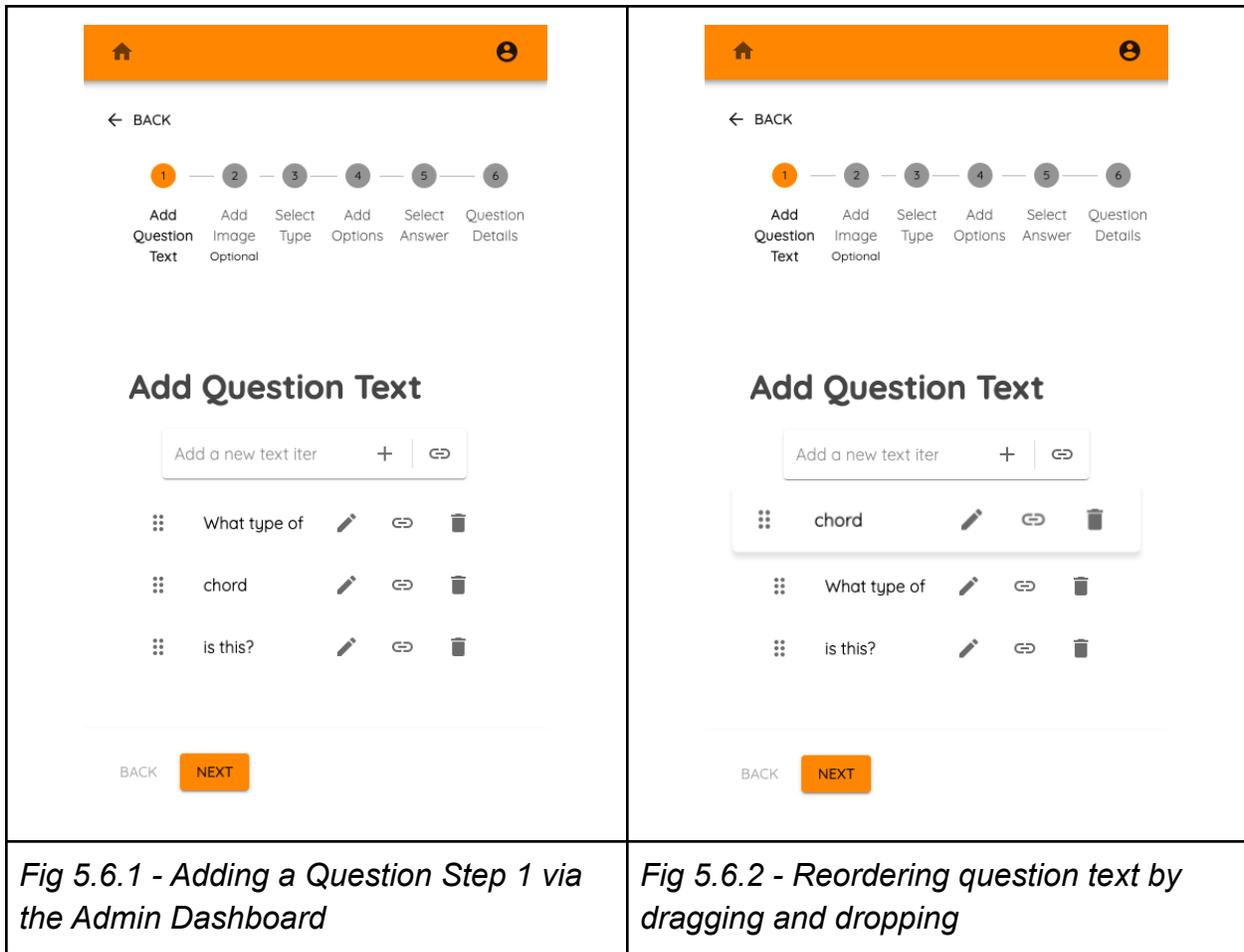
The primary reason for this is deploying a new update to either the iOS and Android store are lengthy tasks, so keeping the majority of data dynamic results in fewer updates being shipped to each App store.

Throughout the implementation of this application, the number of collections of data grew significantly, so it was decided to build an Admin dashboard, which would act as a CMS (Content Management System) for the application.

The Admin Dashboard is built using React and utilises Apollo Client, which is a comprehensive state management library that enables the management of both local and remote data using GraphQL. It integrates nicely with the Apollo Server and allows fetching, caching and modification of application data (apollographql, 2019).

The application allows the creation of modules, lessons, questions and notes, from either mobile or desktop. This is particularly useful for creating questions since it had previously become a lengthy process.

Creating a question on the admin dashboard was implemented to be a seven step process, where the user is asked for information piece by piece to prevent it from becoming overwhelming.



Question text is divided up into sections to allow the admin to attach notes to particular words. Because of this, the order of the words became very important. So a drag and drop system was implemented using the Framer Motion API, which allows the user to drag to reorder question text in an elegant user-friendly way.

When creating question text, you can optionally attach an existing or new note to this piece of text by hitting the link icon as displayed in Fig 5.0.1 above.

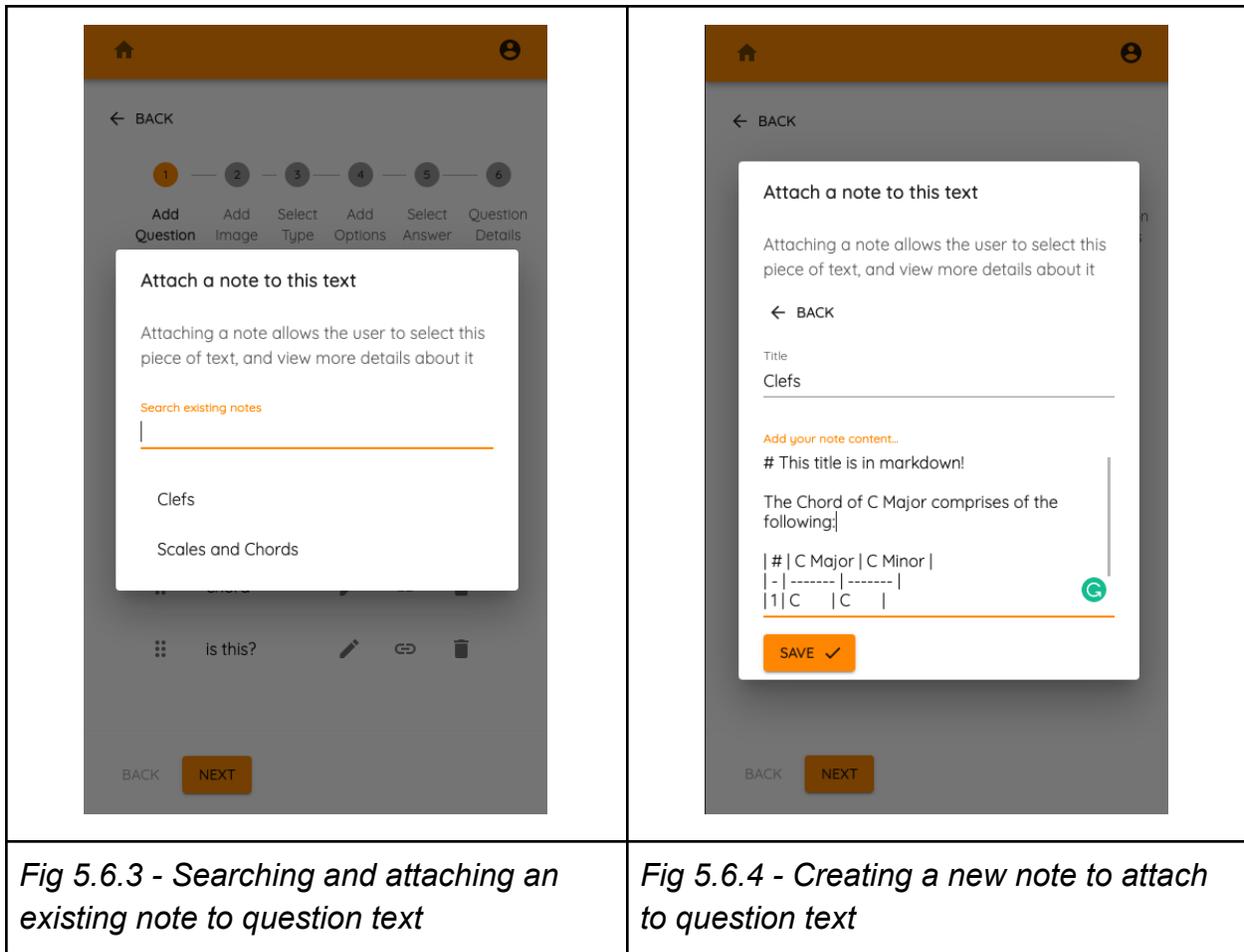


Fig 5.6.3 - Searching and attaching an existing note to question text

Fig 5.6.4 - Creating a new note to attach to question text

Once the admin selects the link icon, a dialogue appears with a list of existing notes that the admin can choose and search from. They can also create a new note from here, as shown in Fig 5.0.4. They are required to add a title, and some content, which will be rendered as markdown, so the admin can add headings, lists, tables and even images that will all be rendered in the mobile application.

Once the admin has completed all steps in creating a question, a preview is generated which mimics how the question will appear in the application.

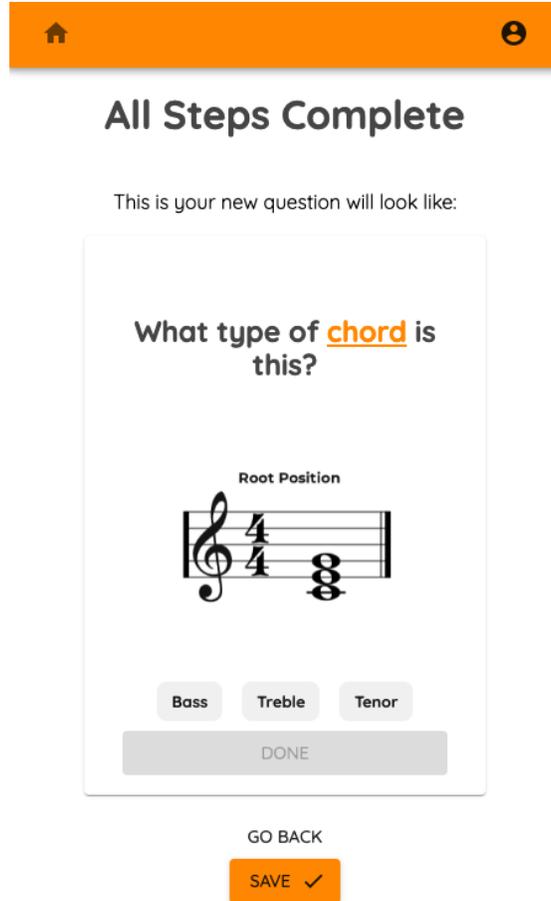


Fig 5.6.5 - Previewing a question after completing all steps

This dashboard allows the user to manage all content within the application from a user-friendly, responsive web application that can also be installed as a desktop progressive web application.

5.7 Conclusion

This chapter describes the implementation of the application. It aims to resolve the problems proposed in the research, requirements and design chapters. The implementation of this application resulted in a GraphQL Apollo Node server, a Flutter hybrid mobile application and a React web application content management system.

The project was developed using Visual Studio Code and various extensions, and the frontend Flutter application was deployed to both Apple's BETA testing software (TestFlight), and Google's Android Play BETA testing software, built into the Google Play Console.

This project utilised MongoDB as a database, which is a NoSQL object-oriented scalable database solution, which connects the GraphQL Apollo server.

The server contains two endpoints, one for serving static images to both the client and the admin dashboard, and a GraphQL endpoint, which manages all queries and mutations going to and from the server, after the data has been validated appropriately.

The server uses Mikro ORM as a TypeScript data-mapper, which handles the relationships between all entities within the server, and the server is deployed to Heroku using continuous integration via the Heroku CLI.

The application's front end was built using the Flutter SDK with the Dart Programming language. It utilises Material UI as a component library, and various other dependencies to handle sound, GraphQL, rendering HTML, local device storage etc.

The frontend was designed in a dynamic way, which allows additional content to be easily added to the application in future, without having to ship an update to the application.

A Content Management admin dashboard was created to allow an admin to log in and easily add content continuously to the system, using a user-friendly, responsive progressive web application.

6 Testing

This chapter discusses the testing of the application, in each of its sections. The aim is to uncover any bugs within the project, analyse the user experience by performing user tests, monitor the overall performance of the project and ensure the application is fully functional.

6.1 Usability Testing

Usability is described as how well a specific user can use a product/design to achieve a defined goal effectively, efficiently and satisfactorily (interaction-design.org, n.d.).

It is vital that users can easily navigate the application, no matter their level of technical know-how. The navigation may seem obvious for the developer or designer, but this can be entirely different for general users. Therefore, user testing was employed to test the general flow of the application.

The main goal of this testing is to assign the user a finite number of tasks, which they are requested to complete without any help from the developer. They are observed throughout this process to see if the application's interface presents any pain points.

The user test is an opportunity to highlight any areas of the application that are particularly difficult to navigate, any bugs the developer may have not noticed, or any additional features that would benefit the application.

6.1.1 User Test Prerequisite

Before the user took part in the user test, they were requested to fill out a consent form based on Google Forms.

The link to the form can be accessed here - <https://forms.gle/f2G6t4UB9riUDrLg6>

They were also asked the following questions:

1. On a scale of 1 - 10, what is your current knowledge of music theory?
2. On a scale of 1 - 10, how comfortable are you with technology?
3. Would you find an application for learning music theory useful?

These questions were asked to understand the user's level of music theory, their general understanding of technology and whether they would find an application like this useful.

The aim of this process was to examine various users with a variety of music theory knowhow and technical knowledge. Thus, this process offered an overview of the user-friendliness of the application for both those of higher and lesser music knowledge.

6.1.2 Tasks

For the user test, the user was requested to complete the following tasks:

#	Description	Prerequisite
1	Register as a user	
2	Log in to the application	Task 1
3	View your profile	Task 2
4	Start a Theory lesson	Task 3
5	Within a lesson, view a note associated with a question	Task 4
6	Complete a lesson	Task 4
7	Log out of the application	Task 2

Table 6.1 - User Test Tasks

After the user completed these tasks, they were asked how difficult they found each task on a scale from one to ten. This offered an insight for the development of the application, identifying any tasks that were particularly difficult to complete.

6.1.3 Results

After the test, users were asked about each task individually and requested to determine how difficult they found the task, on a scale between one and ten, where one is very easy, and ten is very difficult. This offered a greater understanding of how the current interface was to navigate for various users.

From the results of this test , an average was calculated for each user and displayed in the table below, using the results of 5 users:

#	Description	Average Score
1	Register as a user	3
2	Log in to the application	1
3	View your profile	1

4	Start a Theory lesson	1
5	Within a lesson, view a note associated with a question	2.3
6	Complete a lesson	1.6
7	Log out of the application	1.3

Table 6.2 - Average scores of tasks from the user test

In general, the results from the user test are quite strong. The most difficult task was task one, due to the user getting confused about the slow response from the server, while it was waking up. This will be addressed later in section 6.3 - Performance Testing.

6.2 Unit / Integration Testing

This section discusses the feature testing that was completed for this project. Integration testing is defined as a type of test whereby a combination of all the modules are tested as a whole. The primary purpose is to expose defects in the interaction between these modules (Guru99, 2019).

6.2.1 Unit Testing

A total of 18 individual tests took place on the application's server. Each entity was tested individually with its Create, Update and Delete functions to ensure they function as intended.

```

> music-theory-backend@1.0.2 test
> jest

PASS src/__tests__/user.test.ts
PASS src/__tests__/module.test.ts
PASS src/__tests__/lesson.test.ts
PASS src/__tests__/questionText.test.ts
PASS src/__tests__/note.test.ts
PASS src/__tests__/question.test.ts

Test Suites: 6 passed, 6 total
Tests:      18 passed, 18 total
Snapshots:  0 total
Time:       1.235 s
Ran all test suites.

```

Fig 6.2.1 - Server entity testing using Jest

6.2.2 Manual Integration Testing

Test	Expected Output	Actual Output	Status
Register	The user is redirected to the login page	The user is redirected to the login page	Pass
Login	The user is redirected to the dashboard	The user is redirected to the dashboard	Pass
Reopen app after being logged in	User can skip the login process and is redirected to the dashboard	The user is redirected to the dashboard	Pass
Begin Lesson	The user is asked the first question in a lesson	The user is asked the first question in a lesson	Pass
Answer Question Correctly	The user is shown the "Answer Correct" dialogue	The user is shown the "Answer Correct" dialogue	Pass
Answer Question incorrectly	User is shown in the "Incorrect Answer" dialogue, along with an answer hint	User is shown in the "Incorrect Answer" dialogue, along with an answer hint	Pass
Complete Lesson	The user is redirected back to the dashboard	The user is redirected back to the dashboard	Pass

Table 6.1 - Manual testing results

6.3 Performance Testing

6.3.1 Server Testing

As mentioned in the implementation chapter, the applications' server is hosted on Heroku. Rolling out an update to the server can be done via the Heroku built-in CLI which pushes all objects to a Heroku container, and runs an appropriate pre-build test to ensure the code can compile and run successfully.

The server is hosted on the free (Hobby) plan with Heroku, which puts the server to sleep after approximately 30 minutes of inactivity. As a result of this, the first interaction with the server from sleep requires some idle time before the server can respond to a request.

An endpoint response time test was run on every query & mutation present on the server using Postman. This allowed us to measure the response time to each function available through GraphQL on the server.

Test	Total Time (ms)	Average Time (ms)
Endpoint Test from Sleep	20980	599
Already awake	8579	245

Table 6.3.0 - Results of endpoint testing using Postman

As shown in the above table, the average response time was over double the average response time when the server has to run these requests from sleep. This suggests that there could be issues in the future should the application be built for production. At that point, it would be necessary for the developer to either switch to a paid plan on Heroku or choose to host the server somewhere else.

6.3.2 Mobile Application testing

Various performance tests took place using XCode instruments and Android studio. Using XCode Instruments, it was possible to measure the performance of the application at various aspects of its lifecycle.

6.3.2.1 App Launch testing

App Launch is described as the time from when the user taps the app icon, to the first interaction by the user on the application (Collino, 2020). The longer it takes, the less likely a user is to stay on the application.

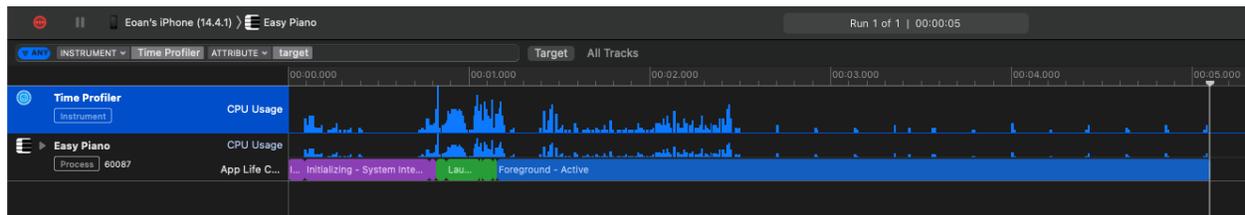


Fig 6.3.0 - Analysing the app launch using XCode Instruments

As shown in Fig 6.3.0, the app launch took a total of 5 seconds. The main component in this is the Foreground - Active section, which alone took approximately 3.84 seconds.

This test has been run twice, once when a user is already logged in, which will skip the welcome screen and head straight to the dashboard page, and send a fetch request for

a list of modules. The second test was run when a user was not logged in, so it simply displays the welcome screen.

Although the first test required more network activity, both tests were almost identical from the time it took to launch the application to the first interaction point.

6.3.2.2 CPU & Memory Testing

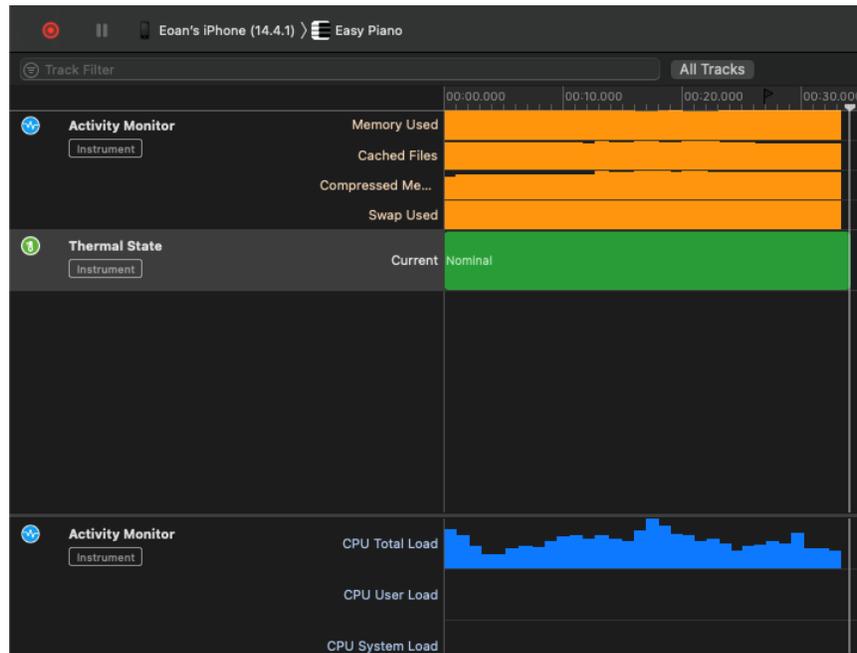


Fig 6.3.1 - Analysing activity throughout a full flow of the app

6.3.2.3 Memory Leak Testing

A memory leak is described as any portion of an application that uses memory (RAM) without eventually freeing it. If this continues to happen consistently the application would eventually crash, as its memory would be completely exhausted.

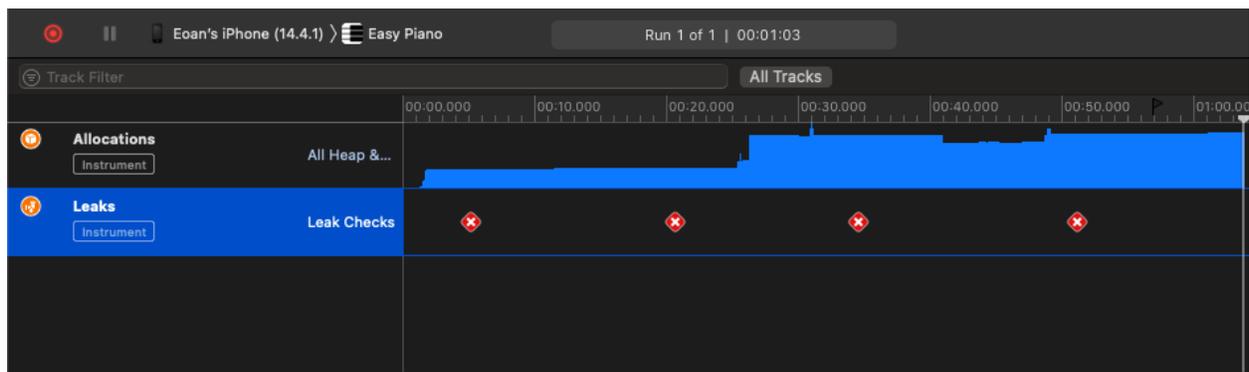


Fig 6.3.2 - Measuring memory leaks within the application

As shown in Fig 6.3.2 above, at multiple stages within the application memory leaks were caught. As shown in Fig 6.3.2 above, during a one-minute session of using the application, at four points memory leaks occurred heavily. This was primarily at points of selecting an answer while playing sound, and successfully answering a question.

These memory leaks occurred primarily in conjunction with the packages being used at this point in the application. As a result, the developer should analyse the currently written code and implementation of these packages, to ensure they are used and managed appropriately.

6.3.2.4 Broad Device Testing

The mobile application had been deployed to both iOS & Android's internal testing systems. Particularly in the Android market, there are a lot of devices to tailor for, and many of these devices are different in unique ways. This means the mobile application can run millions of devices, ranging from high end to low-end users (flutter.dev, n.d.).

Specific tests were performed to test the usability and performance of the application on various iOS and Android devices, noticeably the application was fluid and high performing on all devices it was tested on.

However, automated testing was performed on the Google Play Console, reporting a low frame rate on Nokia 1 devices, which were released in February 2018. The test reported 20% of frames being frozen, and a device launch time of 2110 ms.

6.3.3 Admin Dashboard testing

Performance testing took place using Google Lighthouse, which is built into the Chrome inspection tools. It tests Performance, Accessibility, Best Practices, SEO & Progressive web application support.

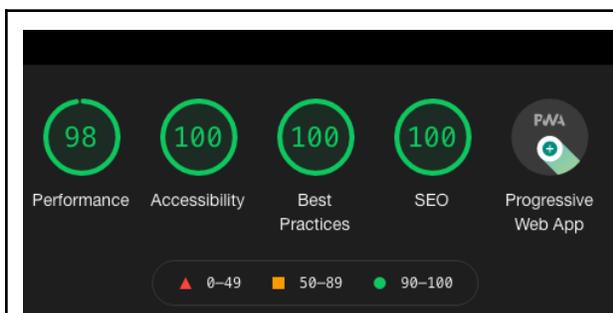


Fig 6.3.3 - Lighthouse audit desktop

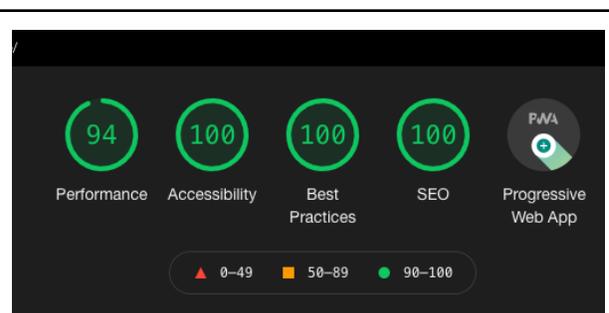


Fig 6.3.4 - Lighthouse audit mobile

These audits give an idea of how websites will perform, whether they follow the majority of web development standards, and due to ReactJS bundles, whether they are well optimized. Off the bat the application has very strong performance.

The application also has progressive web application (PWA) support, which allows the application to be downloaded as a desktop application, or an application straight from the home screen on a mobile device.

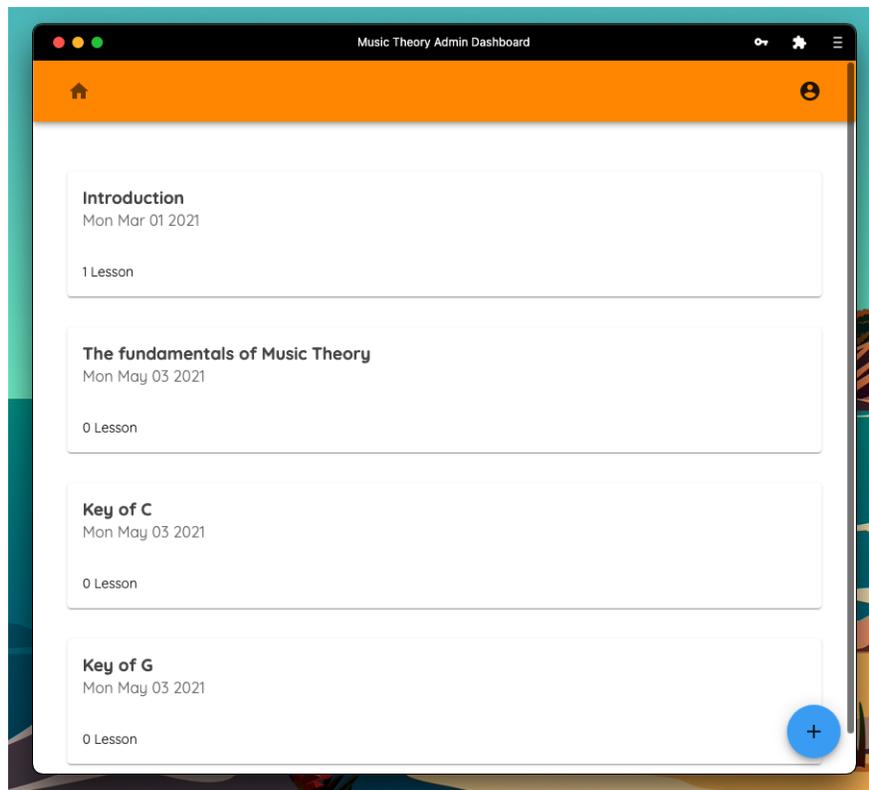


Fig 6.3.5 - Admin dashboard PWA on Mac OS X

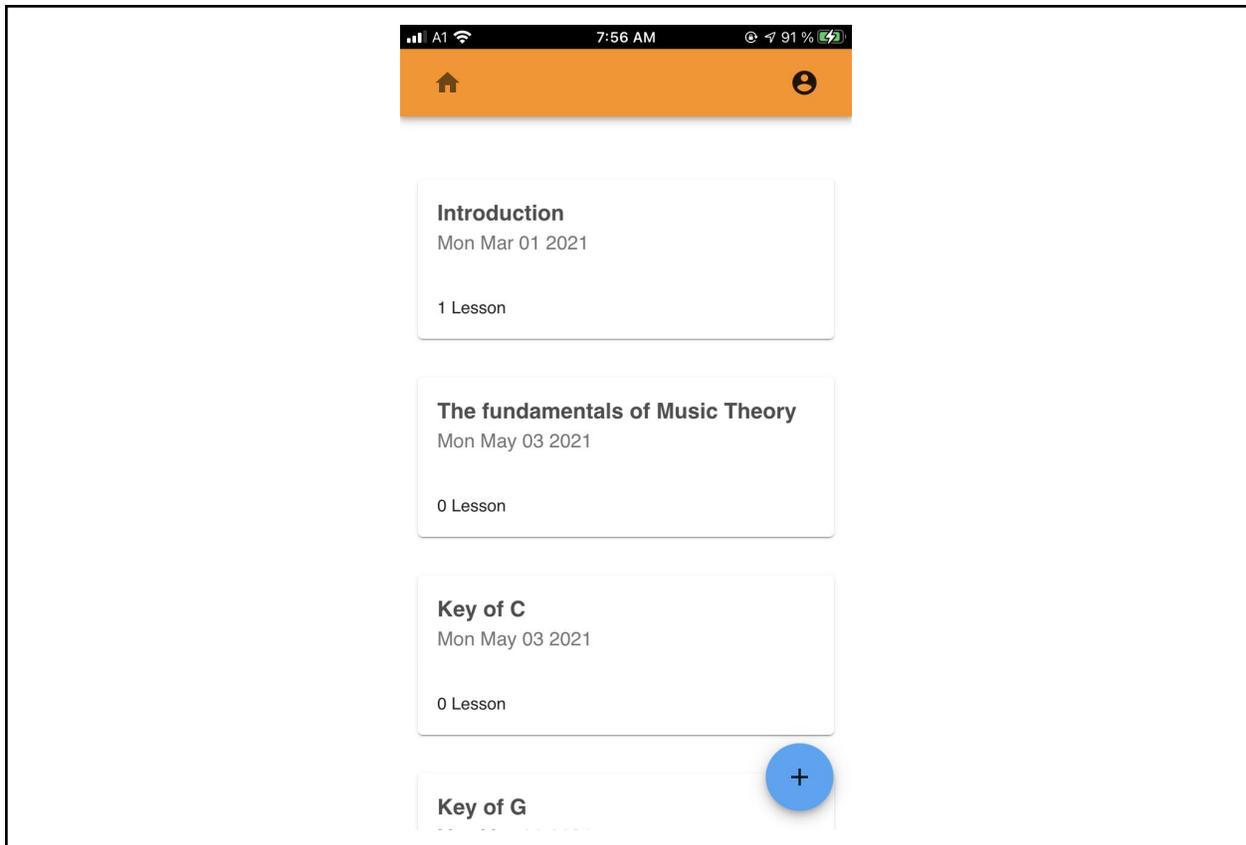
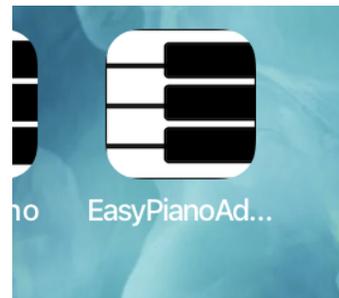


Fig 6.3.6 - Admin dashboard PWA on iOS



*Fig 6.3.7 - Admin dashboard PWA App Icon
Mac OS X*



*Fig 6.3.8 - Admin dashboard PWA App Icon
iOS*

Git was used for version control within all components of this project. For the Admin dashboard, it was used to keep changes and the application was deployed to Github Pages.

An automatic workflow was set up to enable automatic deployment when a new commit was pushed to the repository.

When a new commit is pushed to Github, it immediately runs an Action workflow, which builds the new commit, ensuring it is error and warning free, and then deploying it to Github Pages.

The workflow was written using a yml configuration file. This workflow also gets the location of the server URL from the repositories secrets, which is placed to store environment variables.

This workflow was implemented to automate the deployment & testing workflow, it also displays a badge on the README showing whether the current commit is passing testing or not.

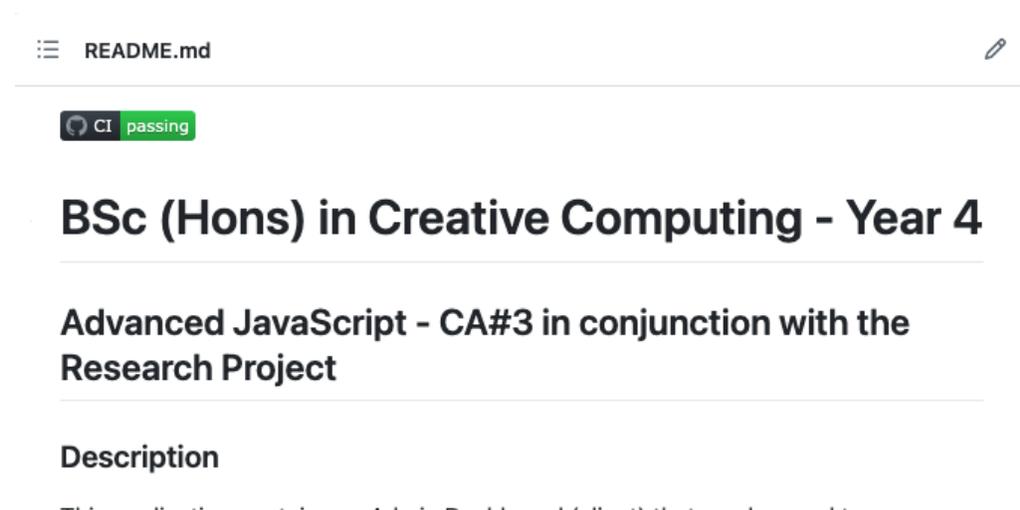


Fig 4.0.3 - README.md with Continuous integration badge

6.4 Conclusion

This chapter has discussed the testing of the application, in each of its sections. The aims to uncover any bugs within the project, analyse the user experience by performing user tests, monitor the overall performance of the project and ensure the application is fully functional, have been thoroughly observed.

Various types of usability tests took place to measure how usable, efficient and satisfying the mobile application is to end-users. User testing took place to measure how effective the user interface was to various users, and how the application performed on different devices. Based on the results of these User tests, this chapter concludes that, in general, the application has a very strong UI, and is generally user friendly.

Notably, some users that answered questions incorrectly and repeated the same lesson, learned from their mistakes, remembering which questions they answered incorrectly, remedying this the second time around. This displays the overall effectiveness of the application, showing that if a user is willing to continue to use the application, they will likely learn from the questions and resources provided.

Unit testing took place on the server to individually test the function of each entity within the database, and manual integration testing took place to measure the combination of all modules together, and how well they interact with one another.

In-depth performance testing took place on all three modules of this project. Endpoint testing took place on the server to test overall response times, both directly from waking up and from a server already running. The mobile application was tested from various aspects, including overall CPU & Memory usage while using the application, app launch testing, memory leaks and broad device testing, to check compatibility on various devices, particularly on Android.

The Admin Dashboard was also tested from various aspects. A Google Chrome Lighthouse audit was performed to check its performance, accessibility, best practices, SEO and whether it offers Progress web application support. The audit resulted in a significantly high score in each aspect, and the admin dashboard ran very fluidly as a result.

Continuous integration was implemented on the Admin dashboard, to test the system after each commit to Github. The code was tested for warnings and errors, and after passing all tests it ran a new build, and rolled out an update to Github pages, where the admin dashboard was hosted.

7 Conclusion

7.1 Project Management

This section discusses how the project was managed over 8 months. The initial proposal was presented in September, but the project commenced in October 2020 and went on until May 2021.

The main deadlines were as follows:

Week Beginning	Research Project
21/09/2020	Proposal
26/10/2020	Requirements
30/11/2020	Research
07/12/2020	Interim Presentation
18/01/2021	Design Document
08/03/2021	Beta Code
22/03/2021	Implementation Document
01/04/2021	Testing Document
08/04/2021	Final Source Code
07/05/2021	Final project documentation
18/05/2021	Final Presentation

Table 7.1 - Project deadlines

The project was planned using a Gantt chart mentioned within section 3.5 Project Plan. This Gantt chart was later converted into a Trello board, which was used throughout the project.

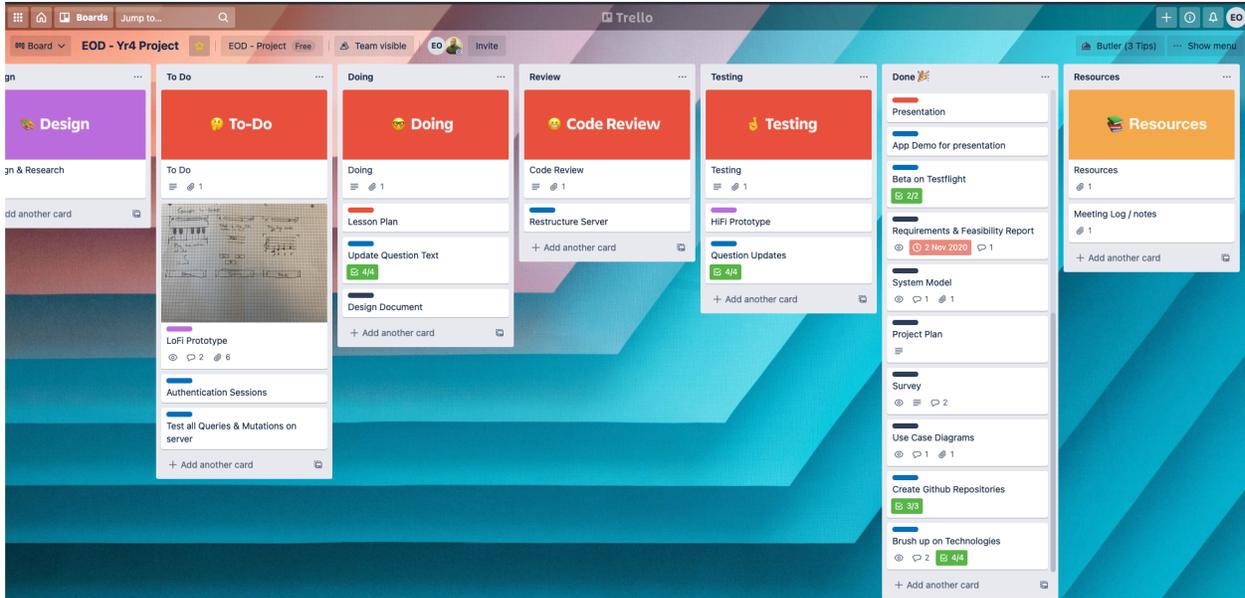


Fig 7.1 - Trello board used for project management purposes

Trello was used along with the Scrum Methodology, which advocates for various sprints, throughout the project.

Weekly meetings also took place with the project supervisor, to discuss what had been done, what needed to be done, and any problems encountered. These meetings were extraordinarily beneficial because they kept the project going in the right direction throughout the year.

Git and Github were used throughout the entire project for version control. Each component got its own Github Repository, to maintain order and structure between all of the code.

At any point there were modifications made to the code, a commit was made to the Github Repository.

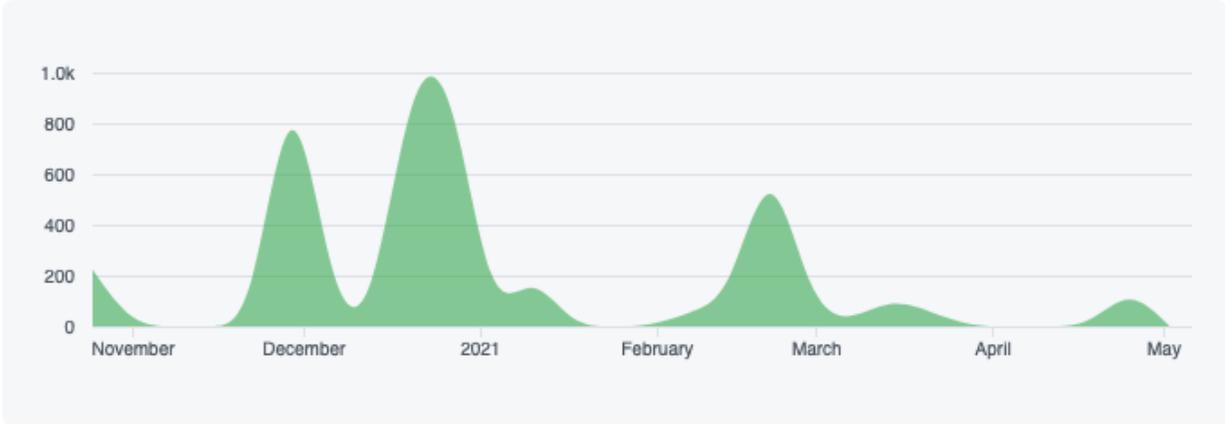


Fig 7.2 - Analysis of code additions to the backend



Fig 7.2 - Analysis of code additions to the frontend

From the Github Analysis, it was noted that a total of 102,615 lines of code were written for this project.

7.2 Future Development

This section discusses possible future developments of this project. The project began as an idea which was much more ambitious than the end result. Although the outcome is a fully functioning application, content management system and server, the research covered music theory and improvisation at a much deeper level.

Improvisation

One item in particular was the implementation of improvising through the application, which never made it to production. Although the structure was set up, and a lot of the functionality was implemented, it was never at a state to be presented to the end-user.

The research and structure has been done for this section of the application, thus, it requires implementation on the mobile application, which would include pitch detection and modification of the existing lesson flow structure.

Rewards system

A method to retain users would be to implement some sort of rewards system. The application already includes points, which are added up over time, but a type of currency could be implemented, and a store concept, where the user can purchase cosmetics for the application, would also be highly effective.

Social Aspects

Applications like Duolingo have many ways to retain users, one of which is the ability to follow friends, see when they are online and compete with them on a week by week basis in leagues.

Implementing these features would offer another method of convincing users to consistently return to the application.

The functionality to view when another user is online is also already half-implemented within the current application model. GraphQL uses a web sock system called subscriptions, and the foundations of this were implemented at the beginning of the project. It is believed that it would not be a huge overhaul to implement the rest of the functionality for this.

Monetization

Should the application begin to gain traction with users, it would be worth considering how to monetize the application. From the research, it was mentioned by the user's that they always have to pay for music learning applications, so it would be worth keeping the application free, but offering some sort of benefit to users who pay a monthly fee.

Another possibility is to show the user's advertisements or give the user the option to watch an advertisement for an additional reward.

7.5 Learning Outcomes

This project covered a broad array of topics, including researching music theory at an in-depth level, exploring the design of many mobile applications, particularly educational applications, learning the Dart Programming language from scratch, and learning GraphQL, Mikro ORM, Apollo and how it all integrates with TypeScript.

None of the technologies above were covered in the course curriculum, and thus were self-taught outside of class hours. LinkedIn Learning courses and many online articles were utilized to understand how these complex technologies work and interact with one another.

This project has taught me a huge amount about GraphQL, and building applications in Flutter. I can now confidently work in these technologies at an intermediate level as a result of this project and the extensive research put in.

7.6 Project Summary

The objective of this project was to explore and understand the pedagogy of Music Theory and Improvisation through mobile applications. The overall goal was to see whether this was possible at a basic level.

The project began as something much more ambitious. It not only aimed to cover teaching a user music theory, a highly dense subject in itself, but also how to improvise. Unfortunately, due to parsimonious constraints, the element of musical improvisation within the application was never fully implemented.

The design of the project takes inspiration from applications such as Duolingo, which turn learning a language into bite-sized chunks of gamified education. This method of educational gamification was applied to this project.

The mobile application is built using Flutter and was written in the Dart programming language. It offers high performance and fluidity and can run easily on hundreds of different devices, all from a single code base.

The server was built using Apollo, Express, Mikro ORM and Node.js all written in TypeScript. The application was designed from the ground up with high performance and uses GraphQL as an endpoint for sending and receiving data between the server and the client.

7.7 Final Words

The primary goal of this project was to see whether teaching music theory was possible through mobile applications. It aimed to offer a fluid, high performing mobile application from which a user could learn and develop knowledge in music theory.

From the user tests conducted, a highly encouraging observation noted that users who initially got questions wrong, and went back to part take in another lesson, got those same questions correct on the second time. Through reading the learning resources and answer hints provided, the users learned from their mistakes.

From this observation, I can confidently say that the goal of this project was achieved. A high performing application was built that can be deployed to millions of devices, which is capable of effectively educating the user on the subject of music theory.

References

Amazon. (2019). *About AWS*. Amazon Web Services, Inc.

<https://aws.amazon.com/about-aws/>

Apache Software Foundation. (n.d.). *Apache Kafka*. Apache Kafka. Retrieved November

30, 2020, from <https://kafka.apache.org/>

Armstrong, S. (2018, October 5). *The untold story of Stripe, the secretive \$20bn startup driving Apple, Amazon and Facebook*. Wired UK.

<https://www.wired.co.uk/article/stripe-payments-apple-amazon-facebook>

AWS. (n.d.-a). *AWS Solution Provider Program*. Amazon Web Services, Inc. Retrieved November 29, 2020, from <https://aws.amazon.com/partners/solution-provider/>

AWS. (n.d.-b). *Financial Services Case Studies - Amazon Web Services*. Amazon Web Services, Inc. Retrieved November 29, 2020, from <https://aws.amazon.com/financial-services/case-studies/>

AWS. (n.d.-c). *Global Infrastructure*. Amazon Web Services, Inc. Retrieved November 29, 2020, from <https://aws.amazon.com/about-aws/global-infrastructure/?p=ngi&loc=1>

AWS. (n.d.-d). *Storage - Amazon Elastic Compute Cloud*. Docs.aws.amazon.com. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Storage.html>

AWS. (2006). *What Is Amazon S3? - Amazon Simple Storage Service*. Amazon.com. <https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html>

AWS. (2019a). *Amazon EC2 Pricing - Amazon Web Services*. Amazon Web Services, Inc. <https://aws.amazon.com/ec2/pricing/>

AWS. (2019b). *Regions, Availability Zones, and Local Zones - Amazon Elastic Compute Cloud*. Amazon.com. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html>

AWS. (2019c). *What Is Amazon EC2? - Amazon Elastic Compute Cloud*. Amazon.com.

<https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html>

AWS For Business. (n.d.). *What are some Amazon EC2 use cases?* AWS for Business.

Retrieved November 29, 2020, from

<https://www.awsforbusiness.com/amazon-ec2-use-cases/>

Back4App. (2019, November 26). *Firestore vs Heroku|What are the Differences?*

Back4App Blog.

<https://blog.back4app.com/firebase-vs-heroku/#:~:text=Unlike%20Firestore%2C>

[%20Heroku%20developers%20can](https://blog.back4app.com/firebase-vs-heroku/#:~:text=Unlike%20Firestore%2C%20Heroku%20developers%20can)

Barr, J. (2006, August 25). *Amazon EC2 Beta*. Amazon Web Services.

https://aws.amazon.com/blogs/aws/amazon_ec2_beta/

Barr, J. (2008, May 29). *More EC2 Power*. Amazon Web Services.

<https://aws.amazon.com/blogs/aws/more-ec2-power/>

Bessie Chu. (2013, November 4). *Amazon Web Services SWOT & Competitor Analysis*.

SlideShare.

<https://www.slideshare.net/BessieChu/amazon-web-services-swot-competitor-a>

[analysis](https://www.slideshare.net/BessieChu/amazon-web-services-swot-competitor-a)

Chapel, J. (2020, April 1). *Who Is Leading Among The Big Three?: AWS vs. Azure vs.*

Google Cloud Market Comparison - DZone Cloud. Dzone.com.

<https://dzone.com/articles/who-is-leading-among-the-big-three-aws-vs-azure-vs-#>

[:~:text=Cloud%20Computing%20Market%20Share%20Breakdown&text=As%20of](https://dzone.com/articles/who-is-leading-among-the-big-three-aws-vs-azure-vs-#:~:text=Cloud%20Computing%20Market%20Share%20Breakdown&text=As%20of)

[%20February%202020%2C%20Canalys](https://dzone.com/articles/who-is-leading-among-the-big-three-aws-vs-azure-vs-#:~:text=Cloud%20Computing%20Market%20Share%20Breakdown&text=As%20of%20February%202020%2C%20Canalys)

Clark, J. (2019, November 28). *DigitalOcean vs Heroku | Which is better?* Back4App Blog.

<https://blog.back4app.com/digitalocean-vs-heroku/>

Datanyze. (n.d.). *Heroku Market Share and Competitor Report | Compare to Heroku,*

Firebase, Google App Engine. Datanyze. Retrieved November 30, 2020, from

<https://www.datanyze.com/market-share/paas-445/heroku-market-share>

Dubsmash. (n.d.). *Dubsmash - Customer Success | Heroku.* Wwww.heroku.com. Retrieved

November 30, 2020, from <https://www.heroku.com/customers/dubsmash>

Google. (n.d.). *PayPal Case Study.* Google Cloud. Retrieved December 2, 2020, from

<https://cloud.google.com/customers/paypal>

Harvey, C., & Patrizio, A. (2020, October 22). *AWS vs. Azure vs. Google: Cloud*

Comparison [2019 Update]. Datamation.com.

<https://www.datamation.com/cloud-computing/aws-vs-azure-vs-google-cloud-comparison.html>

Hatton Enterprise Solutions. (2019, February 10). *SWOT analysis - AWS.*

Wwww.hattonenterprisesolutions.uk.

<https://www.hattonenterprisesolutions.uk/blog/swot-analysis-aws/>

Heroku. (n.d.-a). *Dynos and the Dyno Manager | Heroku Dev Center.*

Devcenter.heroku.com. Retrieved December 2, 2020, from

<https://devcenter.heroku.com/articles/dynos#isolation-and-security>

Heroku. (n.d.-b). *Pricing | Heroku.* Wwww.heroku.com. Retrieved November 30, 2020, from

<https://www.heroku.com/pricing>

Heroku. (2019a). *Heroku Dynos | Heroku.* Heroku.com. <https://www.heroku.com/dynos>

Heroku. (2019b). *Heroku Security | Heroku*. Heroku.com.

<https://www.heroku.com/policy/security>

Heroku. (2019c, July 24). *About Heroku | Heroku*. Heroku.com.

<https://www.heroku.com/about>

Heroku. (2020, July 13). *Building an Add-on | Heroku Dev Center*. Devcenter.heroku.com.

<https://devcenter.heroku.com/articles/building-an-add-on>

Illsley, R. (2020, May 7). *SWOT Assessment: Amazon Web Services*. Omdia.

<https://omdia.tech.informa.com/-/media/tech/omdia/assetfamily/2020/05/07/swot-assessment-amazon-web-services/swot-assessment-amazon-web-services-pdf.pdf>

ITProToday. (2015, March 31). *Why PayPal Replaced VMware With OpenStack*. IT Pro.

<https://www.itprotoday.com/iaaspaas/why-paypal-replaced-vmware-openstack>

Li, A. (2020, February 27). *PayPal migrating "key" parts of infrastructure to Google Cloud*.

9to5Google. <https://9to5google.com/2020/02/27/google-cloud-paypal/>

Lindenbaum, J. (2009, October 14). *Announcing Huge Growth and New CEO*.

Blog.heroku.com.

https://blog.heroku.com/announcing_huge_growth_and_new_ceo

Motola, C. (2020, February 17). *How Does Stripe Work? The Beginner's Guide To Stripe*.

Merchant Maverick. <https://www.merchantmaverick.com/how-does-stripe-work/>

Ortiz, J. (2015, October 13). *AWS re:Invent 2015 Keynote | Jorge Ortiz, Manager of*

Infrastructure, Stripe. YouTube. <https://youtu.be/WmHineJiYqk>

Papertrail. (n.d.). *Papertrail - Add-ons - Heroku Elements*. Elements.heroku.com.

Retrieved November 30, 2020, from

<https://elements.heroku.com/addons/papertrail>

Redis. (n.d.). *Redis*. Redis.io. <https://redis.io/>

Redis Labs. (2012, November 9). *Six Things to Consider When Using Redis on Heroku*.

Redis Labs.

[https://redislabs.com/blog/six-things-to-consider-when-using-redis-on-heroku/#.](https://redislabs.com/blog/six-things-to-consider-when-using-redis-on-heroku/#.VtcXgpMrKL8)

VtcXgpMrKL8

Rojas, A. (2017, August 31). *A Brief History of AWS*. The Media Temple Blog.

<https://mediatemple.net/blog/cloud-hosting/brief-history-aws/>

Stripe. (n.d.-a). *Online payment processing for internet businesses - Stripe*. Stripe.com.

Retrieved December 1, 2020, from <https://stripe.com/en-gb-at>

Stripe. (n.d.-b). *Pricing & fees | Stripe*. Stripe.com. Retrieved December 1, 2020, from

<https://stripe.com/en-gb-at/pricing>

Stripe. (n.d.-c). *Security at Stripe*. Stripe.com. Retrieved December 2, 2020, from

[https://stripe.com/docs/security/stripe#:~:text=Encryption%20of%20sensitive%20](https://stripe.com/docs/security/stripe#:~:text=Encryption%20of%20sensitive%20data%20and)

[data%20and](https://stripe.com/docs/security/stripe#:~:text=Encryption%20of%20sensitive%20data%20and)

Stripe. (n.d.-d). *Stripe Connect: Use Cases*. Stripe.com. Retrieved December 1, 2020,

from <https://stripe.com/en-br/connect/use-cases>

Stripe. (n.d.-e). *Stripe Partner Program: Become a Partner with Stripe*. Stripe.com.

<https://stripe.com/en-gb-at/partner-program>

Techcrunch. (2010, December 8). *Salesforce.com Buys Heroku For \$212 Million In Cash*.

TechCrunch.

<https://techcrunch.com/2010/12/08/breaking-salesforce-buys-heroku-for-212-million-in-cash/>

Wikipedia Contributors. (2020a, November 18). *Amazon Web Services*. Wikipedia.

https://en.wikipedia.org/wiki/Amazon_Web_Services#:~:text=The%20AWS%20platform%20was%20launched

Wikipedia Contributors. (2020b, November 29). *DigitalOcean*. Wikipedia.

<https://en.wikipedia.org/wiki/DigitalOcean>

Wikisme. (2020, October 30). *Stripe SWOT Analysis: What are the biggest Strengths and*

Weaknesses of Stripe? Wikisme. <https://www.wikisme.com/stripe-swot/>