

Euterpea Tree

**A Decentralised Tool to Aid in The Creation, Collaboration and Distribution of
Computer Music Code**

By: Christopher P.K. Morris

MSc Advanced Computer Science

Supervisor: Dr. Frank C. Langbein

School Of Computer Science and Informatics

Cardiff University



Table of Contents

Preamble	4
Abstract	4
Acknowledgements	4
Introduction.....	5
Motivation for Project	5
Pre-requisites	5
Problems to Solve with this Study from a Technical Perspective.....	5
Sections Overview	6
Aim and Objectives	6
Goal of Study	6
Achievable Project aim for Scope of this Study	6
Achievable Project Objectives for scope of this dissertation	6
Long Term Goals out of Scope for this Dissertation	7
Background Research and Investigation	8
Existing Systems	8
Decentralised Blockchain Music Distribution Applications	8
Key Start-ups Emerging in Decentralised Music Distribution.....	8
Main Problems These Start-ups are trying to Solve which my System could Integrate in a Future Version	8
Core Problems I'm Aiming to Solve with this Project.....	9
Git.....	9
Investigation Into Computer music languages and tools for use within system	10
Range of Computer Music Languages and tools Investigated.....	10
Euterpea (Haskell)	10
Investigation into technologies and architectures to build music distribution and storage system with which will form the main application	13
Main System to be Built on top of Node.js With TypeScript	13
A Centralised Approach.....	13
A Decentralised Approach	15
Investigated Decentralised Technologies	17
Arriving at a data-structure for song data	19
Problem, Approach and Development	22
Problem Definition	22
Types of Users	22
Development using Agile Scrum Framework	22
User Stories (user requirements)	23
Use Case Diagram of System	24
Approach (design)	24
Project Structure	24
Development and Implementation	26
Architectural Code Structure and Dependencies	26
Design and Structure of Command Line System	26
Gun API and Graph Design	30

Hashing and Adding Immutable Objects (Song Nodes) to Graph.....	31
Syncing of Data when Users are Offline	31
Advanced JavaScript Patterns used In Development of System	32
Analysis	35
The List Command: as a user i want to list songs on the tree, so that i can discover songs and the data (like hashes) associated with songs.....	36
The Checkout Command: as a user i want to checkout a song on the tree, so that i can play it, view the code and alter it's code before publishing	38
The Publish Command: As a user i want to publish a song to the tree so that i can share my music with others	39
Things to Consider with Performance	40
Conclusion	40
Learnings	41
References.....	42
appendices	42

Preamble

Abstract

Computer Music based languages and tools have begun to emerge over the last decade or two but there aren't many systems that aid in the creation, collaboration and discovery of artefacts in such languages. This study is an investigation and implementation into an approach for distributing such music data as code across many users creating a complete immutable, traceable and searchable history of musical ideas (as a tree structure). The system has two core parts: the chosen computer music language to produce the music and the tool that is used to distribute it among other users. I'll explain why the Haskell programming language with the library Euterpea was chosen for the music creation aspect and why a decentralised approach with the Gun.js JavaScript library was ultimately used for the networking and distribution of such files via a command line interface (CLI) based tool written in Node.js. the aim is for the final product to be similar in usage and interaction to a distributed version control system (DVCS) of musical data.

Acknowledgements

I'd like to thank all the friends, tutors and staff who have stuck by my side and been a source of encouragement and support throughout my time at Cardiff University. I'd also like to thank my mother for her patience and support during my masters, I'd also like to thank Dr Frank Langbein for his guidance and supervision during this project.

Introduction

Motivation for Project

As someone looking to enter the field of computer music it's clear that the field and artists within it producing music are fairly fragmented and could hugely benefit from an platform of interconnected data, first however we should formally define what the field of computer music actually is.

Computer music is the application of [computing technology](#) in [music composition](#), to help human [composers](#) create new music or to have computers independently create music, such as with [algorithmic composition](#) programs (Wikipedia, 2021)

this project looks to address this issue by developing a system where computer music programmers can publish, checkout, discover and play other users compositions (via code) creating a sort of ecosystem of linked musical data. From a high level perspective, the following are the core goals I plan to achieve with such a system:

- a platform and tool to learn about computer music/music theory concepts and aid in the creation/sharing/distribution of such learnings/songs easily among users
- need for an easy-to-use command line based tool similar to Git in usage to facilitate in the sharing, distribution and discovery of such song files
- inspire a community of programmers to contribute to a collective database of music (and vise-versa; encouraging musicians to learn how to code via use of tool and platform)
- create with confidence: knowing you can prove you are the original author of the songs you publish and retain credit by tracing further iterations of songs back to their original source

Pre-requisites

Some of the following concepts are beneficial but not essential in reading and understanding this study:

- Basic JavaScript and programming knowledge
- Basic understanding of Decentralisation (dApp's) and networking
- Data integrity, uniqueness and hashing
- Data structures and directed acyclic graphs (DAG's)
- Blockchain ideas and concepts such as ledgers
- Unified Modelling Language (UML) diagrams/notation and understanding of architectural diagrams
- Basic understanding of music theory

Problems to Solve with this Study from a Technical Perspective

There are certain technical requirements that need to be in place so that computer music programmers are comfortable with sharing and storing their code/music with the world via such a system.

- prevent tampering of song data and ensure its immutability so other users can't claim credit
- research and create a scalable data structure of song data that is fully traceable and easily traversable that keeps a complete history and timeline of all musical data produced in the system
- abstract away the complexities of discovering other users on the network and sharing/synchronising data with them
- research, pick and embed a suitable and elegant computer music coding language to be used in the creation of song files and data within the system
- data longevity and prevention of data loss; it's paramount in such a system that precious musical data and material survives as long as possible and can withstand computer faults such as server crashes and data corruption.

Investigating different approaches to these problems and Finding novel solutions to such problems form the core of what makes up this research and study.

Sections Overview

In *aims and objectives* section I will discuss goals and objectives for this study and what is and isn't within scope for this study. In *background research and investigation*, I explore computer music languages, existing systems that deal with similar problems, centralised and de-centralised architectures and graph data structures. By the end of this section I arrive at a computer music language, architecture, database and data structure to use for the system. In *problem, approach and development* I clearly define the requirements of the system, the architecture, structure and design of system and go down into the lower-level implementation details of the system. In the *Analysis* section I demonstrate how well implemented functionally meets the previously defined requirements via a series of test-cases. In *conclusion* I reflect on what I've been able to achieve with the system thus far and what I envision for its future. In *learnings* I mentioned some of the various new concepts I've learnt about.

Aims and Objectives

Goal of Study

The goal of this study is to investigate, research relevant technologies and then build an MVP of a robust system where users can safely start creating and sharing song data with each-other. The goal is to get the project to a state whereby at the end of this initial study the main research, design and technological decisions have been made and local and online testing can start to begin with real users to see if such an application could be of real value to the computer music community as my hypothesis suggests.

Achievable Project aim for Scope of this Study

Implement a prototype of a system where song data can be easily shared and stored among users where you can easily list, checkout and publish song files via a familiar to use command line interface.

Implement a way for song data to be stored in an immutable tree like data structure *with a hard constrain that all songs must derive from a previously checked out song*, with this functionality in place users can start coding music from either a base template (genesis block/node) or start working/writing their composition from a user created song node thus forming a traceable tree of linked data, this functionality should work somewhat familiarly in concept to that of branching found in Git.

Achievable Project Objectives for scope of this dissertation

Within the timeframe of this dissertation project, I have had to establish what the core achievable goals are. I've narrowed them down to a set of the following objectives:

- research and choose the best suitable computer music/audio programming language/tool to be used within this system (Sonic Pi, SuperCollider, Euterpea (Haskell), Pure Data etc..., a more comprehensive list of such languages can be found at: https://en.wikipedia.org/wiki/List_of_audio_programming_languages)
- decide on the most suitable language and technology to build and write the system in
- develop a familiar and robust command line interface with *list, checkout and publish functionality* which can also handle additional flag options arguments
- experiment with libraries and technologies that could be best suited for this project and select the most suitable for this system after investigation that assist with problems in networking and data storage

- look into centralised vs decentralised approaches and which would be better suited to this project in particular
- figure out how to easily store/persist and back-up song data and keep multiple copies of it should one version or copy of the data be lost or corrupted
- look into ways to create immutable song data which can also be referenced by a computed hash of that data
- establish a scalable tree or graph data structure that can be efficiently and quickly traversed and look into techniques to ensure that nodes within the tree can't be tampered with and altered by malicious users of the system or those wishing to steal credit for previously created songs
- all song nodes/objects in the data structure must as a hard constrain contain a hash reference of a previously created song node/object
- look into ways and techniques for for users to be able to prove they are the original creator of published song data so that they can retain credit for their music
- design and lay down foundations for project codebase on a flexile and scalable architectural structure so the project can be easily further maintained and developed in the future

Long Term Goals out of Scope for this Dissertation

I plan to setup the scaffolding within the dissertation period of this project for future features and commands to be implemented that may not be within the scope of this dissertation such as:

- “play” command: this feature will allow you to shuffle through and play songs from the command line, with additional flag options for things like song sample duration and filter by artist
- “status” command: quickly and conveniently see what the last checked out song was similar to the Git status command
- “init” command: this will walk the user through configuration setup and create a JavaScript Object Notation (JSON) config file which will store details of the user such as their artist/publishing name and potentially contact information should they write a hit song. It will work very similarly in concept to node package manager's (NPM) init command and Git's init command. The idea here is that the user programmer can initiate and user the system under many different directories under different artist names she they choose, similar to some way in how we can have many different Git repositories.
- “prove” command: it will be vital that users can easily and quickly prove they are the original creators of files/songs created in the past should they become popular. Some investigation/research will be done on this feature during development of the MVP to ensure it can be easily integrated afterwards as a key feature
- “network” command: check the status of the network and how many users are online at any one time
- Ability to rate, comment and give feedback on users compositions directly within the system itself. Such a feature would allow users to list and play songs by rating for example
- If enough musical data is populated into the system overtime a learner/model (machine learning) could be trained with the gathered data to generate music and identify patterns, *the fields of computer music and artificial intelligence are becoming more intertwined and having a consistent large set of musical training data that adheres to one language/interface has been a problem that this system could potentially address* and researchers could use.
- Validate and perform checks on song files to make sure they follow they follow and adhere to an interface before allowing a user to publish to ensure data consistency

Background Research and Investigation

Existing Systems

Decentralised Blockchain Music Distribution Applications

There are currently plenty of applications in the early stages of tackling the problem of music distribution and attribution via decentralization and blockchain methods. Immutable and distributed Blockchain ledgers are mostly used as an emerging technology to handle rights management of music between creators and users, but *there are very little systems that actually aid in the discovery, creation and distribution of the code that produces the music oppose to the music itself, this is what differs my system from existing ones.* The core user of my system is the programmer not a musician or listener completely unfamiliar with coding.

As a by-product of what I plan to do with my system with how the data by constraint must be connected (reference a previous node in graph let's say) and can be tracked back to it's original source, rights and royalty percentages could be implemented/integrated similar to Vezt Inc. straight into the application in the near future, I plan to design the app with room for features like this in mind but they aren't the immediate priority for the scope of this study.

Key Start-ups Emerging in Decentralised Music Distribution

- Vezt (<https://www.vezt.co/>): *there digital marketplace allows artists and songwriters to share a percentage of a song's royalties for royalty-based financing through an Initial Song Offering (ISO.) The ISO includes the date and time in which the royalty rights will be made available to the public*
- MediaChain (<http://www.mediachain.io/>): recently purchased by music streaming giant Spotify to handle problems with music rights and attribution, *The company issues smart contracts with musicians that directly state their royalty stipulations without the hassle of confusing third parties or contingencies*
- Voise (<https://www.voise.com/>): *Artists upload their content, the platform recommends music based on a user's preferences and users pay the artists (who receive almost 100% of the revenue) for their music*

Main Problems These Start-ups are trying to Solve which my System could Integrate in a Future Version

- Music attribution: Tracking music back to original source and fairly distributing royalties
- Paying artists via smart contracts, sometimes through systems such as Ethereum <https://ethereum.org/en/>
- Digital rights: Proving artist ownership of music
- Ensure data has no single source of truth and is distributed (something I want my system to have from the beginning)
- No middle-man, artists receiving 100% royalties and money for their work without the need for them to go through a record-label

Core Problems I'm Aiming to Solve with this Project

Here I list some similarities in my system and key differences that differentiates it from current existing systems.

- Provide a common interface/template and platform for all programmers to code to with Haskell Euterpea for creation and publishing song data forming a sort of *educational system of musical data and knowledge*
- Low barrier of entry for programmers, additional installation of blockchain protocols and apps is not needed to get started, just download the main command line application
- *Encourage users to share their music code with others and to collaborate/branch from other existing compositions all straight from the command line.* I may need to find a way to incentivize the programmer to share their music (publish into the system) in the future which could be by them gaining royalties or by developing features into the system that would allow them to develop a fanbase of other users and external listeners (non-programmers could perhaps benefit from this system by just listening to the music the code produces from a web GUI)
- Data consistency: Establish a platform/system which programmers can populate with music data over time creating a *collective knowledge of computer music code for learning/educational purpose as well as a potential consistent dataset that researchers could access and use*
- Make music code data (and publishing artist information) is immutable and resistant to tampering
- Be notified when new music data is added/published to the system

Git

Git <https://en.wikipedia.org/wiki/Git> is essentially the worlds most popular DVCS for distribution of code, I hope to take familiar concepts to programmers such as branching and pushing and apply these ideas in a slightly different context to my system. Git is so popular among programmers I'd like the user experience of my system to have a familiar command line interface users can quickly start getting to grips with "out of the box".

Git works by creating a Merkle Tree https://en.wikipedia.org/wiki/Merkle_tree as its underlying data structure, where each node is a cryptographic hash value of it's files contents. Changes to directories can quickly be checked by checking the computed root hash of the entire Merkle tree to see if it differs from the new one. I want to adopt a familiar system of hashing song data to check for changes and tampering between users with my project.

Why not just use Git to track multiple user's song data?

If users of my system wish to locally version their song code before publishing, I'd actually encourage it but the system I plan to develop isn't a clone of git; there are things git isn't designed to do that I plan to achieve with my system. Here are some issues git would have with a use-case such as my project:

- A monolithic repository would be needed to keep all branches and data, which would also get very messy
- Users could delete and modify code in branches
- Users could potentially delete central repositories
- Users can't be notified on there machine locally of when new songs are published without explicitly checking the state of the repository via "git status"
- No enforced rules or consistent interface to what a user can commit and push

Investigation Into Computer music languages and tools for use within system

Range of Computer Music Languages and tools Investigated

I looked into a plethora of languages and tools for music creation at the beginning of this project and also investigated live coding music environments such as SuperCollider <https://supercollider.github.io/> and Sonic Pi <https://sonic-pi.net/> but realised these tools are more suited for live performance where you write, edit and delete lines of code “on the fly” to get the maximum use out of them. I wanted a language where your compositional code could live as a sort of immutable artefact over time like a musical score, where others could play and analyse it at their own time and leisure akin to how we admire and study the stored works of Beethoven or Mozart.

After researching various languages to use with the system I developed a set of criteria that had to be met: fun, expressive and transferable. After looking at notable languages such as chuck <https://chuck.stanford.edu/> and SOUL <https://soul.dev/> (a language recently created by the writer of the popular Juce c++ audio library), I realised these languages live in isolated environments and aren't really interoperable with other standard coding languages. With Euterpea users of my system could learn to be better functional (Haskell) programmers as well as musicians at the same time, both of which are of interest to me and many I've met in the programming community. Having said that, there's no reason why future bespoke versions of my system couldn't be used with other computer music languages i.e. SOUL Tree or chuck tree.

Euterpea (Haskell)

Overview

Haskell is a statically typed purely functional language, it is also lazily evaluated meaning you can effectively recursively play a song (or series of notes a function outputs) recursively forever without running into a stack overflow exception. Euterpea is a Haskell Library that can be easily installed by Haskell's main package manager Cabal. At its core the library contains type abstractions and functions for producing MIDI output.

Euterpea is a cross-platform, domain-specific language for computer music applications embedded in the Haskell programming language. Euterpea is a wide-spectrum language, suitable for high-level music representation, algorithmic composition, music analysis, working with MIDI, low-level audio processing, sound synthesis, and virtual instrument design (Quick, 2012)

Haskell programs mostly work by building up abstractions and composing functions together which lends itself well to music creation in that abstractions from Euterpea for chords (`:=:`), sequences of notes (`:+:`) and all other musical concepts can be expressed and composed together algorithmically and efficiently. In terms of music production this allows users to define auxiliary functions that perform operations such as a minor 3rd harmony or piano sweep (descending from long note durations to quick: or from a whole note 1/1 to say a 1/128). With such a language and library users could purely just write auxiliary functions for others to build on top of in their own compositions, *the idea is with time after the system is populated with enough data (Haskell code) there will be pre-existing functions and algorithms available to use for any musical need you can imagine.*

Written code examples

Here I'll share a few of my experiments and learnings with Euterpea and demonstrate the types of useful basic functions you can define such as scales, chord progressions, piano sweeps or virtually anything else. The Idea is to pick a computer music language where you can start expressing basic musical ideas and

concepts but then in the future with some study and learning you can delve into writing more complex algorithms and functions, given the time constraints I have with this study I've barely scratched the surface of Euterpea's potential compared to someone who has a more in depth knowledge of music theory and programming with it.

Outputs of functions of type "Music Pitch" can be parsed to the Euterpea defined play function, the generated MIDI output will be played by an installed MIDI synthesizer such as SimpleSynth or Fluidsynth.

```
-- 'harmonize note' function abstraction (of minor 3rd) (3 semi-tones below original note) (or steps on a piano)
hNoteMinor3rd :: Dur -> Pitch -> Music Pitch
hNoteMinor3rd d p = note d p :=: note d (trans (-3) p) -- re-usable pattern here
```

FIGURE 1 HARMONY MINOR THIRD AUXILIARY FUNCTION I WROTE WHICH CAN LATER BE USED TO DEFINE HARMONIC CHORD SEQUENCES TOGETHER FOR EXAMPLE IN YOUR COMPOSITIONS
[HTTPS://EN.WIKIPEDIA.ORG/WIKI/MINOR_THIRD](https://en.wikipedia.org/wiki/Minor_third)

```
standardTuningGuitar :: Dur -> Music Pitch
standardTuningGuitar dur = e 2 dur :+: a 2 dur :+: d 3 dur :+: g 3 dur :+: b 3 dur :+: e 4 dur
```

FIGURE 2 STANDARD GUITAR TUNING UTILITY FUNCTION (TAKES DURATION OF LENGTH OF EACH NOTE TO BE PLAYED AS ARGUMENT)

```
*Euterpea.Experiments.StandardTuningGuitar> :t standardTuningGuitar
standardTuningGuitar :: Dur -> Music Pitch
*Euterpea.Experiments.StandardTuningGuitar> :i Dur
type Dur = Rational      -- Defined in 'Euterpea.Music'
*Euterpea.Experiments.StandardTuningGuitar> █
```

FIGURE 3 HERE I'VE LOADED UP THE STANDARD TUNING OF A GUITAR FUNCTION I WROTE FROM GHCI TO TEST WHAT OUTPUT IT PRODUCES AND CHECK IT'S TYPE

```
genAllDurations :: Int -> [Dur]
genAllDurations n | n < 1 = [] -- edge case
genAllDurations 1 = [1 :: Rational] -- recursive stopping point
genAllDurations n = 1/(toRational n+1-1) : genAllDurations (n-1)

pianoSweepFastToSlow :: Music Pitch
pianoSweepFastToSlow = notesFromArrNSemiTonesHigherAtEachStep (genAllDurations 8) 1

pianoSweepSlowToFast :: Music Pitch
pianoSweepSlowToFast = notesFromArrNSemiTonesHigherAtEachStep (reverse $ genAllDurations 8) 1
```

FIGURE 4 CODE TO GENERATE FAST TO SLOW AND SLOW TO FAST PIANO SWEEP LIKE SOUNDS INCREMENTING THE MUSICAL NOTE VALUE BY A SEMITONE (1 LITERAL KEY UP HIGHER ON A PIANO) EACH TIME

```
-- As a simple example, suppose one wishes to generate a ii-V-I
-- chord progression in a particular key. In music theory, such a chord progression begins
-- with a minor chord on the second degree of a major scale, followed by a
-- major chord on the fifth degree, and ending in a major chord on the first
-- One can write this in Euterpea, using triads in the key of C major,
-- as follows:
```

```
t251 :: Music Pitch
t251 = let dMinor = d 4 wn ==: f 4 wn ==: a 4 wn
      |      gMajor = g 4 wn ==: b 4 wn ==: d 5 wn
      |      cMajor = c 4 bn ==: e 4 bn ==: g 4 bn
      in dMinor ==: gMajor ==: cMajor
```

FIGURE 5 HARD CODED II-V-I (2-5-1) CHORD PROGRESSION SEQUENCE

[HTTPS://EN.WIKIPEDIA.ORG/WIKI/II%E2%80%93V%E2%80%93I PROGRESSION](https://en.wikipedia.org/wiki/II%E2%80%93V%E2%80%93I_progression)

```
minorTriadChordCorrect :: Pitch -> Dur -> Music Pitch
minorTriadChordCorrect p d = note d p ==: note d (trans 3 p) ==: note d (trans 7 p)

majorTriadChordCorrect :: Pitch -> Dur -> Music Pitch
majorTriadChordCorrect p d = note d p ==: note d (trans 4 p) ==: note d (trans 7 p)

twoFiveOneCorrect :: [PitchClass] -> Octave -> Dur -> Music Pitch
twoFiveOneCorrect majorScale o d = let sndDegreeRootPitch = getMajorPitchClassByDegree majorScale 2
      fifthDegreeRootPitch = getMajorPitchClassByDegree majorScale 5
      fstDegreeRootPitch = getMajorPitchClassByDegree majorScale 1
      in minorTriadChordCorrect (sndDegreeRootPitch, o) d ==:
      majorTriadChordCorrect (fifthDegreeRootPitch, o) d ==:
      majorTriadChordCorrect (fstDegreeRootPitch, o) (d*2)
```

FIGURE 6 II-V-I (2-5-1) CHORD PROGRESSION FUNCTION THAT WILL PLAY DIFFERENT SEQUENCES DEPENDING ON PASSED IN NOTE ARGUMENTS

```
*Euterpea.Experiments.StandardTuningGuitar> :l StandardTuningGuitar.hs
[1 of 1] Compiling Euterpea.Experiments.StandardTuningGuitar ( StandardTuningGuitar.hs, interpreted )
Ok, one module loaded.
*Euterpea.Experiments.StandardTuningGuitar> standardTuningGuitar qn
Prim (Note (1 % 4) (E,2)) ==: (Prim (Note (1 % 4) (A,2)) ==: (Prim (Note (1 % 4) (D,3)) ==: (Prim (Note (1 % 4) (G,3)) ==: (Prim (Note (1 % 4) (B,3)) ==: Prim (Note (1 % 4) (E,4))))))
```

FIGURE 7 THE "NEAT" THING ABOUT EUTERPEA IS THAT YOU CAN SEE THE RETURNED (RECURSIVE) SEQUENCE OF NOTES THAT FORM THE SCORE/COMPOSITION AND REASON ABOUT THEM AND CHECK THEIR TYPE

Pros

- Haskell is an extremely expressive high level language which makes it perfectly suited for the writing of music compositions with minimal code
- Euterpea library contains abstractions to get started producing sounds at the "note level" instantly out of the box without worrying about signal processing/coding (although you can code at that lower level of abstraction if you wish too)

- the Glasgow Haskell compiler interpreter (GHCI) is effective for loading up songs and quickly testing/playing them, or writing code in the interpreter to experiment with and quickly hear what sounds are produced giving the programmer instant feedback
- The Euterpea library has a wealth of resources and brilliant book available written by well-respected researchers in computer music Paul Hudak and Donya Quick called “The Haskell School Of Music” (<https://www.euterpea.com/haskell-school-of-music/>)
- Immutability: song code is deterministic with no side-effects enforcing users to write more beautiful and clear composition code
- *Because of Haskell’s typed nature If I was to embed this as the chosen language of the system the songs can be type checked and validated before being published to other users to prevent useless/messy data from entering the system, this would be achieved by running GHCI in a separate child process and compiling/checking the final output of the a song.hs file to be of type “Music Pitch” from the main system application*

Cons

- Compiled via Glasgow Haskell Compiler (GHC) which requires an install on most machines
- Doesn’t run in the browser, even though there are solutions being worked on such as Asterius:

Asterius is an experimental GHC backend targeting WebAssembly, which makes it possible to run Haskell code in your browser or in a Node.js web service (Shao, 2020)

- Requires additional software to be installed on your machine to listen to the MIDI output the code produces

Investigation into technologies and architectures to build music distribution and storage system with which will form the main application

in this section I will discuss what languages, architectures and technologies I have investigated and ultimately chosen to build the main application with.

Main System to be Built on top of Node.js With TypeScript

in the interest of time Node.js has been chosen as “base” technology to build up the application on top of due it’s maturity, compatibility with pretty much all operating systems and large eco-system of available libraries for use in just about any use-case. I’m also a fairly proficient JavaScript programmer and have worked extensively in the language and already know a fair amount of the nuances with it and patterns. As the goal of this dissertation is to set the foundations of the system in place for future development and features to be later added, the code needs to be robust, flexible and typed so I adopted typescript <https://www.typescriptlang.org/> as the primary language for development. Typescript gives me the programmer errors and warnings about types and code at compile time avoiding bugs from discretely creeping into the system which so often happens when writing projects in “vanilla” JavaScript.

A Centralised Approach

Here I will discuss what a decentralised approach and architecture would look like for the system. Given I want the core song data of the app to be something of a connected data structure that can be easily traversed and hold references between nodes I’d opt for a graph based database solution such as NEO4J <https://neo4j.com/> which I have used in prior projects which can also handle data at large scales and has a simple and effective querying language called Cypher <https://neo4j.com/developer/cypher/>. The database would be queried via express which is a node.js server-side library that handles network

requests and routing via a HTTP REST based API. I have produced a high-level architectural diagram of what the centralised system and tech-stack would look like.

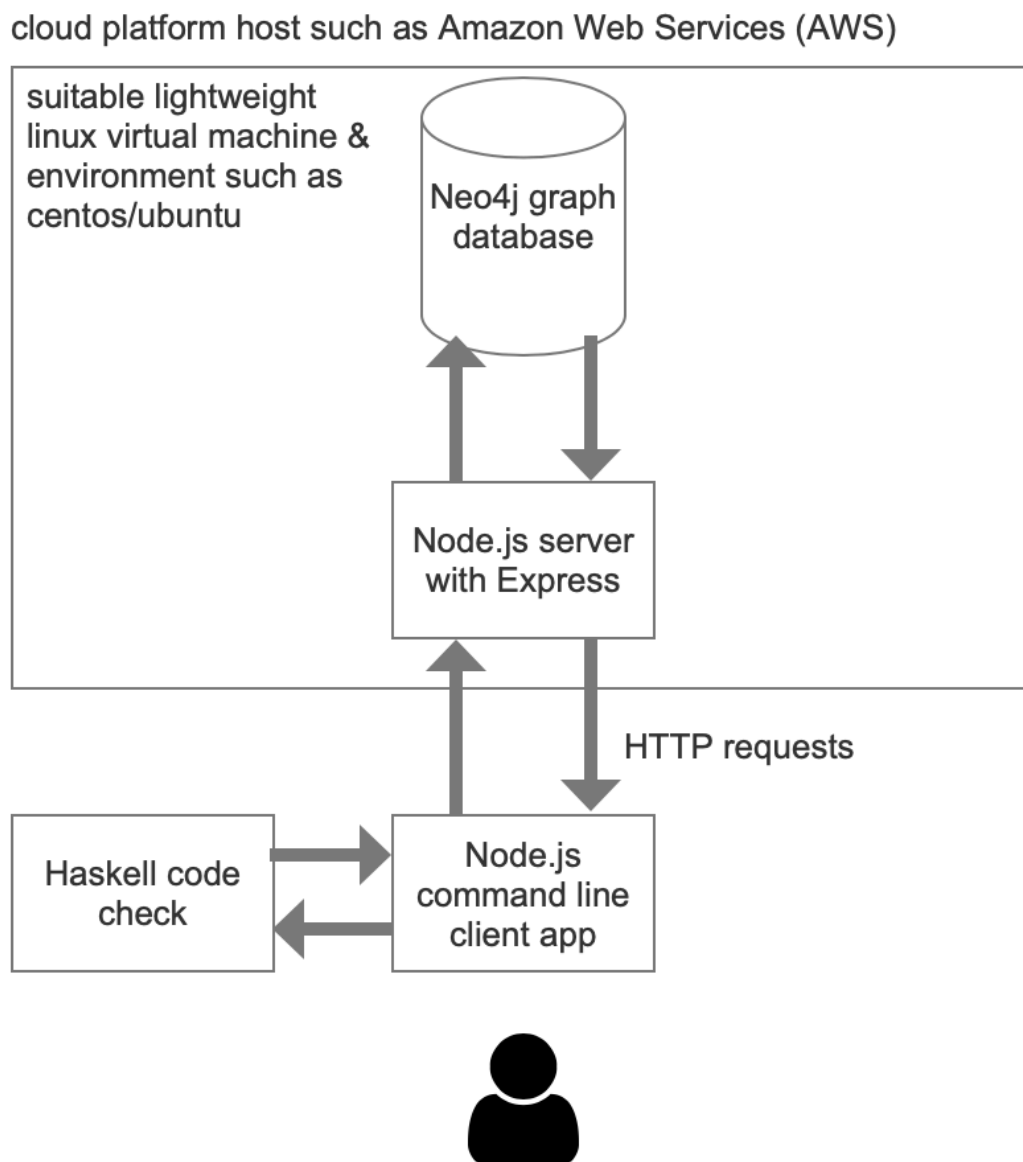


FIGURE 8 THE ARCHITECTURE AND STACK I'D USE IN A CENTRALISED VERSION OF THE SYSTEM

Advantages

- NEO4J is an established, reliable and scalable graph database technology at this point which could form the main data storage and persistence layer of the application song data
- Design and Implementation of the overall system would be more familiar to traditional full-stack web development
- A very clear and cleaner separation of concerns in established between client, server and database following a 3-tier architecture <https://www.ibm.com/cloud/learn/three-tier-architecture>

Disadvantages

- A centralised approach could be more vulnerable to attacks such a distributed denial-of-service attack (DDOS) https://en.wikipedia.org/wiki/Denial-of-service_attack preventing users from accessing their data.
- All data is held in a central location meaning a separate system to store regular backups would have to be implemented
- User has to be online to access their data unless a separate system for caching data locally is implemented

A Decentralised Approach

Here I will discuss what a decentralised approach and architecture would look like for the system.

Decentralised Applications (dApps) are becoming more prevalent now than ever with what some are calling web 3.0 in a bid to shift the ownership of data from centralised based corporations back into the full control of users.

data will be interconnected in a [decentralized](#) way, which would be a huge leap forward to our current generation of the internet (Web 2.0), where data is mostly stored in [centralized](#) repositories (Vermaak, 2022)

in the system I plan to build users should have full control and readily constant access to the musical data they produce at all times. Music is specifically such a case where users may feel uneasy storing their compositions elsewhere at a centralised repository where the data may become out of their control and will likely want access to it even when they're offline. For these reasons a decentralised application more in line with what web 3.0 is hoping to achieve may be far more beneficial for this system going forward. I've produced a high level architectural diagram of what this system would look like following a decentralised approach, and have left questions regarding the technologies that will be used for networking, storage and synchronisation of data among peers.

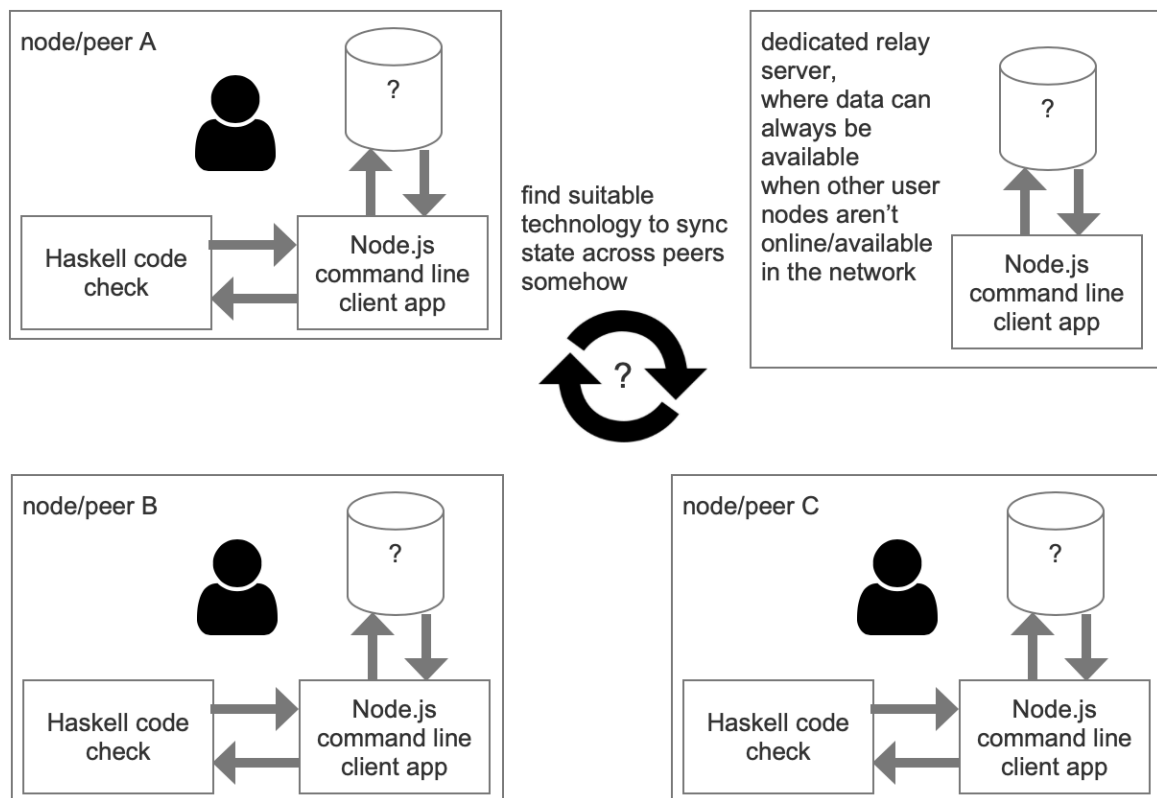


FIGURE 9 ROUGH DESIGN OF A DECENTRALISED APPROACH, THERE ARE QUESTIONS MARKS TO INDICATE WHERE TECHNOLOGIES NEED TO BE INVESTIGATED TO ACHIEVE THESE SYSTEM FUNCTIONS

Advantages

- Offline first: from moment you download system you can start publishing songs locally to be synced to the network later
- The problem of securely storing data is solved as each user will have their own copy of it should one user or the relay server lose it
- More than one source of truth: it becomes harder to maliciously alter song data of system, as each user holds an independent copy of it you can check the data of other peers on the network and see if it matches via a voting system similar to that used to ensure the validity of ledgers in blockchain protocols. A Merkle tree algorithm https://en.wikipedia.org/wiki/Merkle_tree could be deployed to achieve something like this to compute a root hash of entire dataset to be efficiently cross referenced with other peers in network.

Disadvantages

- A lot of decentralised libraries and technologies are new with un-finalised API's still in the experimental stage
- Decentralised networking protocols and Synchronisation of data among peers can be far more complicated than the traditional centralization HTTP protocol and methods for posting (POST) and getting (GET) data using say a RESTful based API

Investigated Decentralised Technologies

Once I decided I wanted to go the route of a decentralized application (dApp) for the previously mentioned reasons and benefits in *A Decentralized Approach* I investigated and experimented with technologies that could solve problems in data storage and synchronisation and narrowed it down to the follow two JavaScript based libraries: IPFS <https://js.ipfs.io/> and Gun.js <https://gun.eco/>.

Interplanetary File System (IPFS) and Interplanetary Linked Data (IPLD)

I followed code-along style tutorials which you can find at <https://proto.school/course/ipld> to learn about the basics of IPFS and IPLD before *writing some small experiments of my own which you can find links and descriptions of in the appendices*. IPFS is a sort of protocol for accessing resources as files and data objects: Users run their computers as nodes or peers in the network and when they went to obtain a resource they also cache or create a local copy of it on their own machines for others to access. Data is accessed and located by its cryptographic hash allowing data to *be addressed by it's content oppose to it's location* (like in HTTP) as a by-product of this data also becomes immutable, meaning different content will produce a different hashable address. These hashed identifiers are known as content identifiers or CID's which is a standard or specification that's been developed by those working on IPFS. This standard allows users to identify what the hash type of that data is (hashing algorithm used) and a codec with tells you how to interpret the data itself if it's an object type or a file. Having this standard format in place is useful as it allows you to continue to update your hashing algorithms and data structures well into the future without affecting your system. *This idea is something I could incorporate into my own system should I need to update what hashing algorithms I use.*

IPLD uses the concept of a Merkle directed acyclic graph or "Merkle DAG" to check the integrity of its linked data structures. The hash of a root node is computer from bottom up, or rather (from leaf nodes root) If a hash of a leaf lower down in the tree has changed than the hash of ancestor nodes will also be different, this is IPLD checks tampering of data structures between nodes. Git similarly also uses Merkle trees to calculate changes of its directories and file contents. *I could use such a technique in my own system to check among peers if an entire graph data structure across peers is fully up to date and synced or has been tampered with by just comparing the computed root hashes.*

Problems with IPFS and IPLD involve it being fairly new and having a convoluted JavaScript API (*JavaScript DAG object API (IPLD)*) which after playing with I would say is more in its experimental stage. I encountered problems with performing complete DAG Merkle tree traversal and finding out how to create and deploy a relay server which, both are which essential for my system, there was also *no obvious way as such to keep a track of the hashes of all nodes that have been previously produced in the system as a list making it hard to find and retrieve previously created objects.*

IPLD's way of linking different graphs and data models together is something I do really like and could allow for song data to be linked with Ethereum and Git based data potentially enabling programmers of my system to earn money. The open protocol nature of IPLD would also song data produced through the system to live on and be accessible long after the application software itself has been retired, as the data

is independently accessible to any user via `ipfs://<hash-of-data-node>` in any browser. I may re-investigate IPFS/IPLD when it's API's or more stable and mature as there are some interesting possibilities here if I were to integrate it as the key data storage and linking method in my system.

Gun.js

At its core Gun.js is essentially an offline-first *graph database* with a fairly simple unified API which abstracts away the complexities with peer to peer (p2p) networking built on top of technologies like WebRTC, WebSocket's and other transport layers. According to Mark Nadal "The whole database is considered to be the union of all peers' graphs. All peers help cache and sync data"

<https://gun.eco/docs/Introduction>

Gun works in node.js and browser based environments, the browser version stores data locally using the LocalStorage and IndexedDB API's whereas a separate database solution was created for the efficient storing of data locally in Node.js called RAD, according to Mark Nadal "RAD is a storage adapter for GUN that stores data at disk using a radix tree"

Because writing storage adapters for GUN has a bunch of nuances and performance tradeoffs, we've designed RAD to handle these nuances for you and chunk GUN's graph into traditional files that can be dumped to disk. It handles correctly merging updates into each batch, managing memory allocation on heavy load, reads across ranges of chunks, and more with a generalizable performance strategy (Nadal, 2021)

The first key problems I had with IPFS and IPLD: was the complexity and lack of information in setting up a dedicated relay server which will always be online for users to fetch graph data from should no current users be running their machines as nodes. Gun solves this problem out of the box making it easy to setup a dedicated relay server. *The second key problem I had was the difficulty in traversing the entire IPLD graph easily given the root node of the graph and getting all nodes in the graph in the order of which they were created: I did have some luck in writing code to find ways around this:*

<https://gist.github.com/zoolu-got-rhythm/06dff4394ac0277fc854f808b26bdac8> but nothing as efficient and simple as just adding all nodes to under a list like directory using the gun set API.

Once I was ready to start experimenting with the library I reached out to the creator and current lead maintainer of the Gun.js library Mark Nadal who actually responded to my questions on twitter and helped me get started with some basic sample code.

Hi Mark, I'm doing my dissertation msc project on a decentralized music creation app, essentially creating a distributed/immutable graph of data nodes that link/ref to prior data nodes (music) in graph. when a new user joins the system i then want to to pull/fetch the entire current history of the graph (all nodes). is this possible with GUN.js?

Sep 24, 2021, 4:57 AM ✓



Yupe! Jump into the chat.gun.eco and we can help with sample code. :)

Sep 24, 2021, 5:10 AM

FIGURE 10 CONVERSATION WITH MARK NADAL, MARK IS VERY ACTIVE WITHIN THE GUN.JS COMMUNITY AND CAN OFTEN BE DIRECTLY CONTACTED FOR HELP WITH THE LIBRARY

I wrote a short experiment to test guns publish and subscribe messaging model and database data modification features and found I was also able to test it easily locally on my machine by creating two separate copies of the same project and running them in which it uses multicast to communicate with each other: <https://github.com/zoolu-got-rhythm/gun-js-node-cmd-line-experiment> this confirmed the library is sufficient for my needs in developing the prototype of my system.

you can do far more complex things gun with features for security and user permissions, but this experiment demonstrated the technology was simple and effective enough to get my project up and running quickly with room to build-in more complex security and functionality later.

Arriving at a data-structure for song data

It became clear at this stage in the project that I would need a directed acyclic graph (DAG)

https://en.wikipedia.org/wiki/Directed_acyclic_graph data structure with no cycles in it. Taking inspiration from IPFS, IPLD, Git and blockchain protocols This approach allows me to check the integrity of an entire graph in the future by computing the tree's root hash via an Merkle tree algorithm (although this isn't necessarily within the scope of this dissertation, the data structure was thought about and designed with this in mind).

Imagine data objects (nodes) created across a linear timeline (which can be found out by the time-stamp at which they created) with a hard constraint that they must reference the hash of valid previous node (song data object). *This data structure essentially forms a tree or DAG where songs can be traced back to their original sources* and would encourage users to build on-top of the ideas and musical compositions that came before.

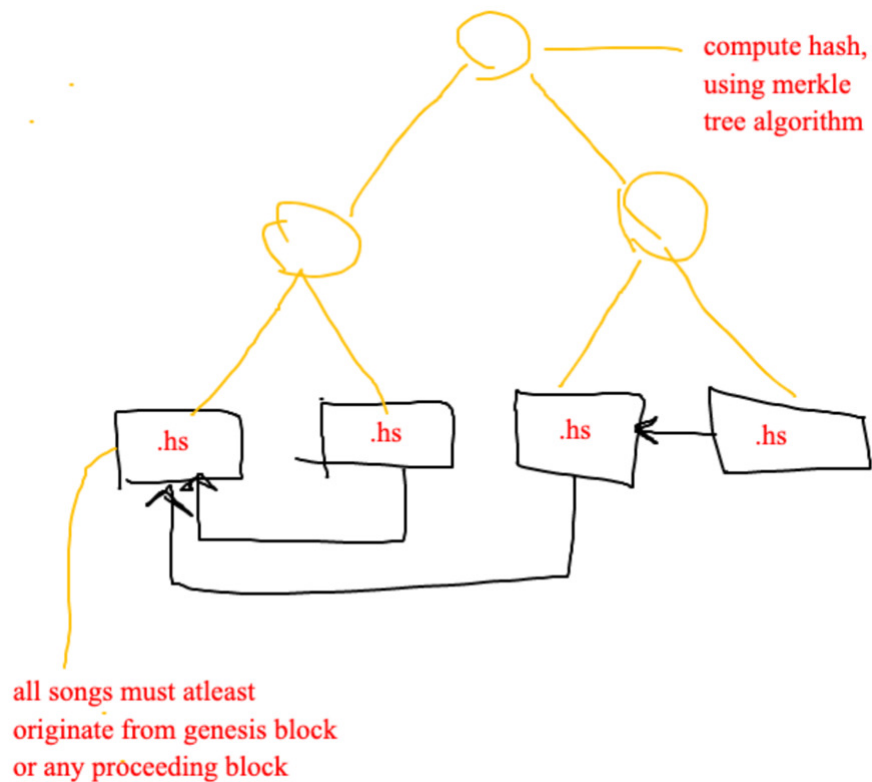


FIGURE 11 .hs FILES REPRESENT HASKELL FILES, WITH ARROWS REFERENCING PREVIOUS NODES FORMING A GRAPH, A ROOT HASH OF ENTIRE TREE COULD BE LATER COMPUTED AND SHARED/COMPARE WITH OTHER USERS USING AN MERKLE TREE ALGORITHM

As a graph I can lay down “building-block” nodes for users to use as a basis to get started with coding their musical compositions with the base genesis node (similar to a genesis block found in blockchain) acting as the base template for songs of the entire system. Users could define building-block nodes with auxiliary and utility functions as well if they so wish and gain some credit later on for users using their functions and algorithms in their musical compositions.

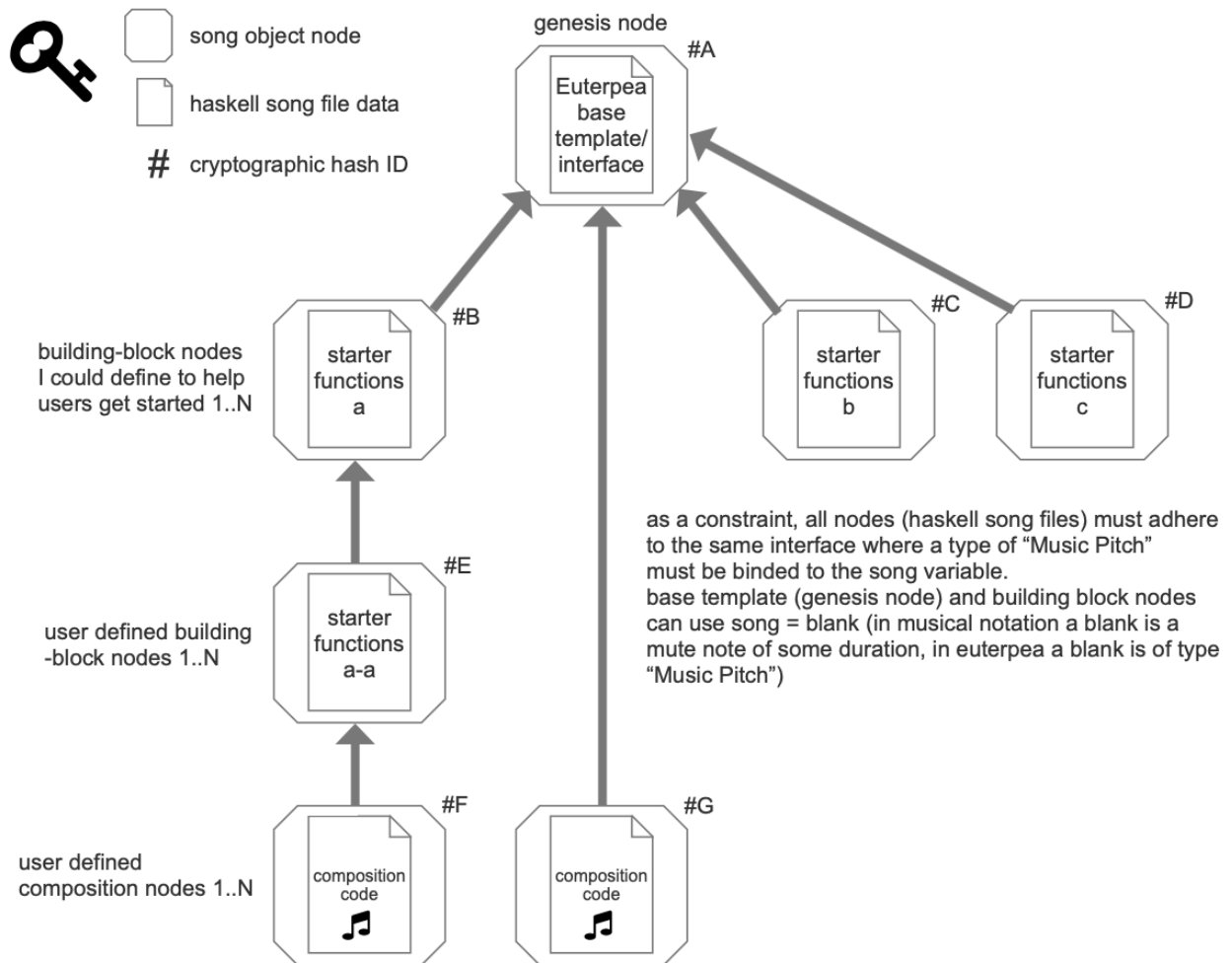


FIGURE 12 A VISUALISATION OF HOW THE DATA STRUCTURE (DAG) WILL SERVE ITS PURPOSE FROM A SYSTEM/APPLICATION PERSPECTIVE

This type of data structure can be easily implemented in Gun.js as it is at it's core a graph database. Links and references can easily be made between objects and identified/accessed via the hash the content of the object produces from the local RAD database (radata folder and files) and Gun core API. *Note: a blank note is the same as a rest note of no duration, in euterpea syntax it would look like "rest 0" for example which produced a type of "Music Pitch".*

```
*Main> :t rest
rest :: Dur -> Music a
```

FIGURE 13 FUNCTION SIGNATURE OF REST CHECKED FROM GHCI

Problem, Approach and Development

Problem Definition

By now the problem has been investigated and researched thoroughly enough to have a *clear definition of and concrete idea what the system is I will exactly be building* and with what tools and technology. I can therefore start to formally define the clear requirements of the system.

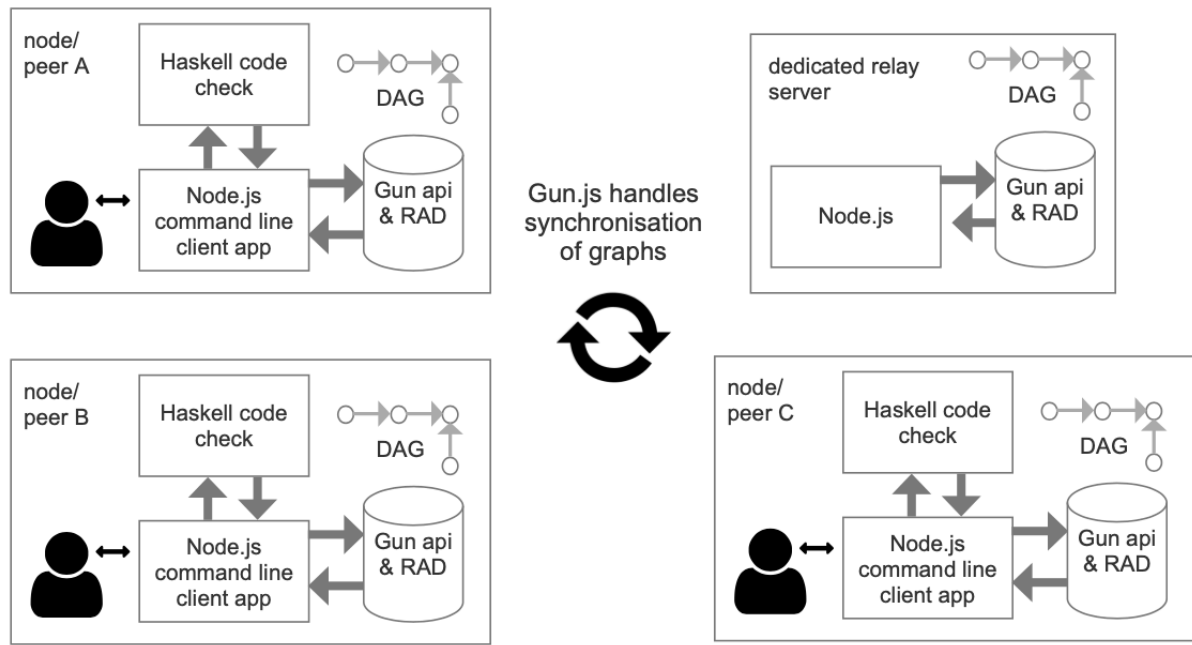


FIGURE 14 A CLEAR OVERVIEW OF THE SYSTEM I NEED TO DEVELOP (HASKELL CODE CHECK IS OUT OF SCOPE FOR THIS STUDY)

Types of Users

First, I need to define what the core users of my system will be. *Primary users of the system will be programmers with prior experience writing code who are hobbyists or researchers within the field of computer music who will interact with the system via a CLI.* Secondary users may be musicians looking to learn about writing computer music code also via interaction with the CLI, tertiary users may be those with no programming experience but wish to quickly explore and listen to the produced music of the system using a user friendly Graphical User Interface (GUI) via a website. Now that I've established the primary user of my system I can start framing user stories around them.

Development using Agile Scrum Framework

For the scope of this dissertation I will concern myself primarily with building a prototype or minimum viable product (MVP) that primary users can interact with first. I can develop the functionality needed for the initial version of the system by defining a set of requirements as user stories. *These user stories will then form the basis of for which tasks need to be implemented as I will be developing the system using the Agile scrum framework,* my project scrumboard can be viewed at <https://trello.com/b/Ea8ivGP8/euterpea-tree>.

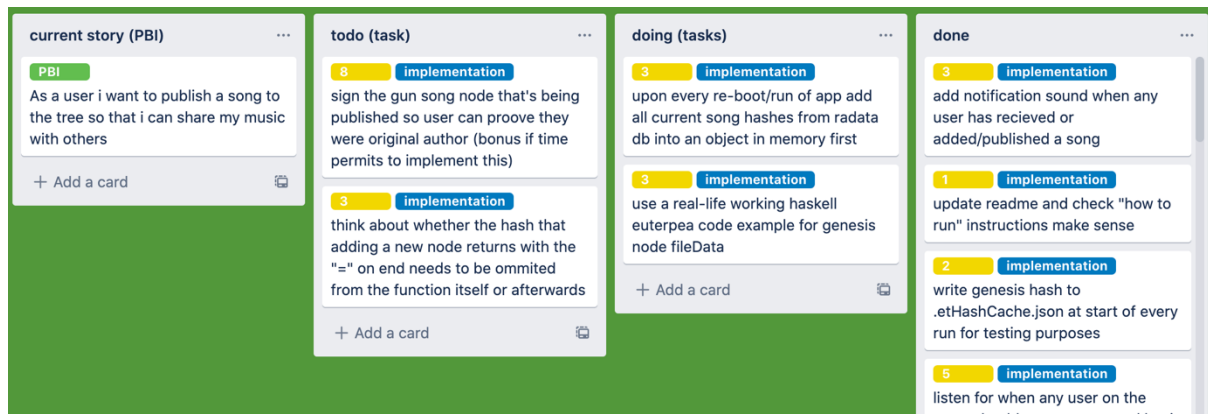


FIGURE 15 SCREENSHOT OF MY SCRUMBOARD STRUCTURE AND TASKS USED TO MANAGE AND ORGANISE APPLICATION DEVELOPMENT <https://trello.com/B/Ea8ivGP8/EUTERPEA-TREE>

User Stories (user requirements)

Here are the following core set of user stories I've defined for development of the MVP of system, tasks were made, estimated and implemented around these.

- As a user i want to *publish* a song to the tree so that i can share my music with others (MVP)
- as a user i want to *checkout* a song on the tree, so that i can play it, view the code and alter it's code before publishing (MVP)
- as a user i want to *list* songs on the tree, so that i can discover songs and the data (like hashes) associated with songs (MVP)
- as a user i want to be assisted in creating the *init* config file so that i can create my artist name and details and publish songs (near-future)
- as a user I want too to use '*status*' command to see last checked out song node, so i know what song i'm working with/on at any given time (near-future)

Use Case Diagram of System

A use case diagram was made using UML to better understand how different types of users will interact with the system, and to capture how interactions will look with the entire system vs the MVP.

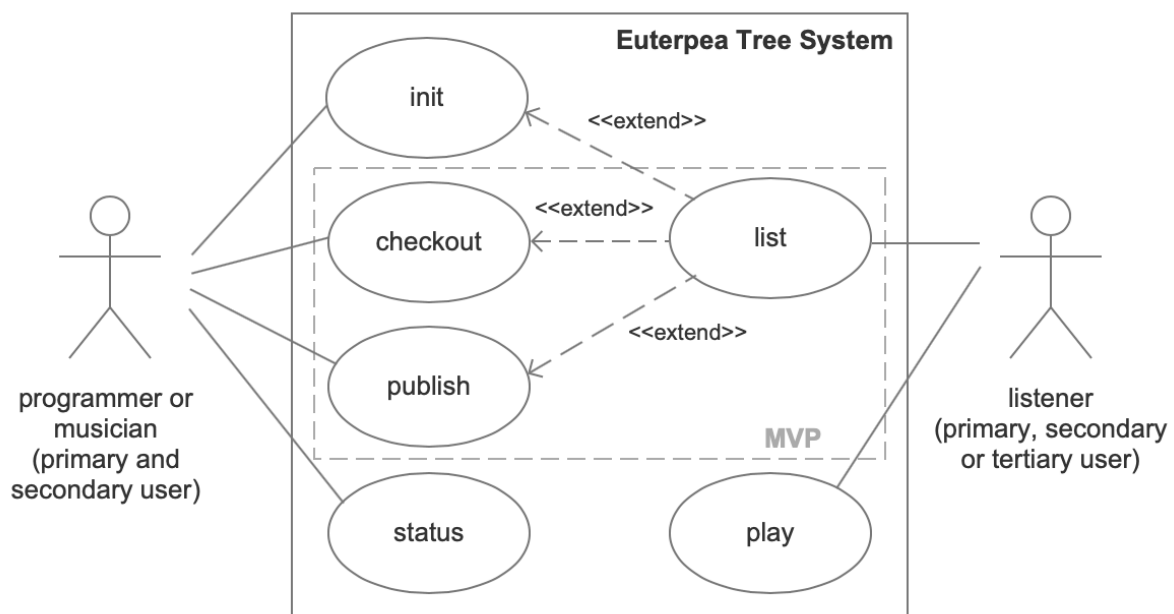


FIGURE 16 UML USE CASE DIAGRAM OF SYSTEM THAT DEMONSTRATES HOW USERS WILL INTERACT WITH THE SYSTEM

Approach (design)

Project Structure

Users who start a euterpea tree based project will need to have the following key 3 files in their directory.

▼ Euterpea Tree project directory
 .etHashCache.json
 etConfig.json
 song.hs

FIGURE 17 EUTERPEA TREE PROJECT FILES

The idea is that a "init" command (similar to purpose in git) which I can implement in the future will setup and bootstrap a users directory with these three files and walk them through configuring them easily via command line prompts. Also similar to initialised git projects, in the future I want users to be able to initialise many different euterpea tree project folders all under different artist names if they so wish.

.etHashCache.json

Whenever you check out a song, the hash of that data object will be written to this file so when you publish a song the system reads from the same file to know what the previously last checked out song

was, this is fundamentally how links between song nodes in the graph are created. The “.” precedes the file name similar to .git or .gitignore because it should be hidden from the user to avoid tampering with it. If the previous song hash was stored in memory only it would be forgotten between each run of the application, so it’s essential the hash is stored and persisted between sessions of running your machine as a node/peer.

```
{ } .etHashCache.json > ...  
1 | { "checkedOutSongHash": "HUe0CvtXF1CnSxccNfkXYqxI+InXaEUxSDZQAe57wB0" }
```

FIGURE 18 .ETHASHCACHE.JSON FILE'S OBJECT STRUCTURE

etConfig.json

the Euterpea Tree config file is where the programmer put's in there artist name for their songs to be published under, when publishing a song the name is read from this file and put into the song data object to added to the gun database graph. In future releases this config file may also include additional details such as contact information (should fans of their music seek to get in touch) and about.

```
{ } etConfig.json > ...  
1 | {  
2 |   "artist": "christo"  
3 | }
```

FIGURE 19 CURRENT ETCONFIG.JSON FILE OBJECT STRUCTURE

song.hs

all songs produced in the system must adhere to the same basic haskell interface so songs can be checked by a child process opened from node.js for validity and correctness before being published into the graph. This allows songs to be playable and shuffled through by default in the future from a simple command line “play” feature which will be essential to quickly listening to and discovering others music.

```
> song.hs > ...  
import Euterpea ( play, rest, Music, Pitch )  
1 import Euterpea  
2 -- https://www.euterpea.com/  
3  
4 -- make sure a midi synthesizer is running before executing -  
5 -- compiled executable  
6  
7 -- form compositional code here such as melodies etc.  
8 -- then bind to song variable  
9  
10 song :: Music Pitch  
11 song = rest 0  
12  
13 main :: IO ()  
14 main = play song
```

FIGURE 20 SONG.JS FILE CONTENTS: THIS WILL FORM THE BASE TEMPLATE OF ENTIRE SYSTEM VIA THE "GENESIS" ROOT NODE OF GRAPH

Development and Implementation

Architectural Code Structure and Dependencies

I want a clear separation of concerns between code that handles command line related functions and database (abstraction over gun api) read and write functions. This will allow me test database related queries in isolation quickly without having to boot-up the command line, I've also isolated read and write file utilities into their own files so they can be tested easily. *commandLineUserInterface.ts* operations depend on running database queries, writes and reads to files in a synchronous fashion where one action must be completed before moving onto the next one, due to the asynchronous nature of callbacks in node.js and the Gun.js API I had to find techniques to refactor and wrap all code inside promises https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise which allow functions to be waited on using "await" syntax before progressing with program execution, the timing and ordering of these operations formed some of the core challenges with the implementation of the MVP which I discuss in *Advanced JavaScript Patterns used In Development of System: async/await* section later.

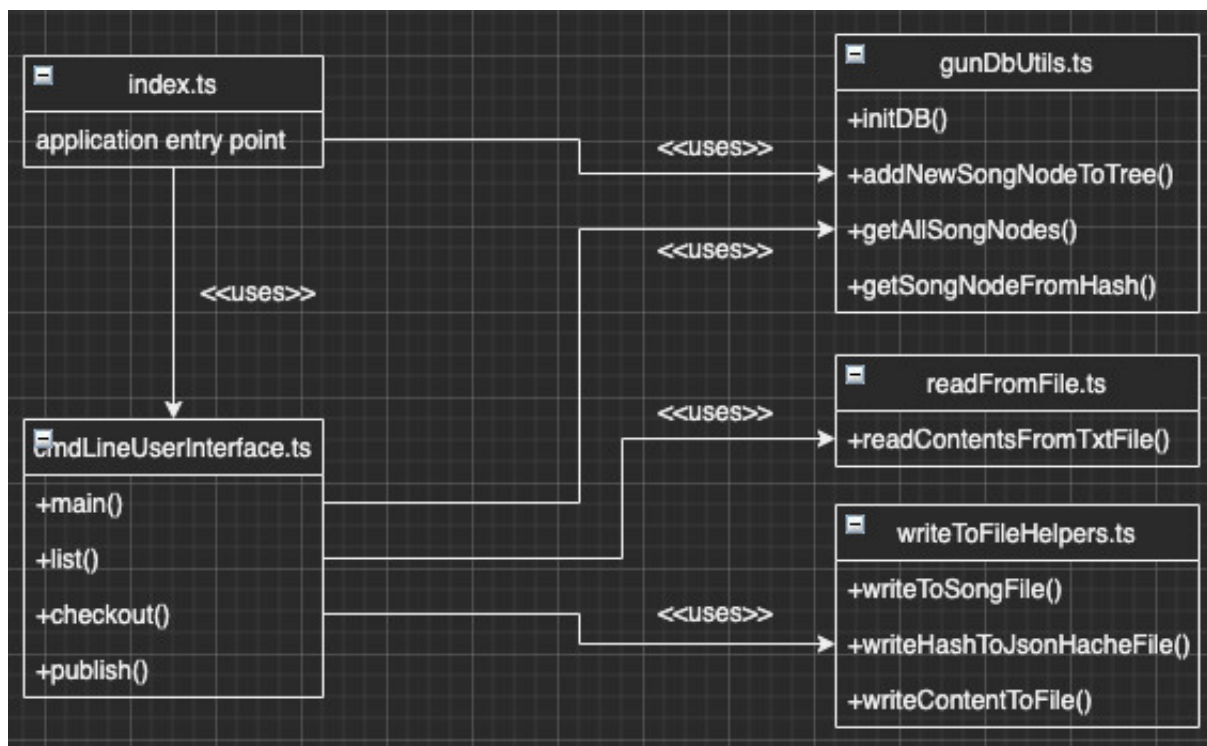


FIGURE 21 UML PACKAGE DIAGRAM OF MAIN EXPOSED PUBLIC FUNCTIONS IN SYSTEM AND DEPENDENCIES BETWEEN FILES

Design and Structure of Command Line System

Handling and routing of commands and flags

I wanted the system that handles command line input (from users) to have an extensible structure from the start so new commands and their associated flags could be continuously added in the quickest way possible following the same established pattern of how the other commands were implemented. To achieve this I first created an object where an command name is a function property (method) on the object, the `main()` method on this object will first format user input to lowercase and then check if the

inputted user command exists as a key on the object, if there's a match the function that matches the key associated with the function is called with the user provided arguments and flags as strings, if there is no match I've specified a pattern for a fallback method to be called instead, I've specified the "main()" function be the fallback method in this case which re-attempts the whole process of asking the user for input and routing it again. note: the external interface of the command functions is all the same but the internal implementation details between them is analogous depending on what operations need to be carried out.

I will use the "checkout" command function as an example to delve into some of the implementation details of how commands are processed.

```
checkout: async function(args: string[]) {  
    // implement checkout  
    // checkout stores the hash of current checked-out song in .etHashCache.json  
    // then writes to the song.hs file the content data from gunDB node
```

FIGURE 22 ALL COMMAND "ROUTING" FUNCTIONS HAVE THE SAME SIGNATURE/INTERFACE. NOTE: THE USEFULNESS OF TYPESCRIPT IN BEING ABLE TO SPECIFY DATA TYPES FOR BETTER READABILITY

Using *regular expressions*, the provided user arguments and flag strings with "—" preceding them are carefully separated/filtered apart.

```
// separate hash arg from flags, result of filter should be size 1 (1 hash arg, rest flags)  
const checkoutArgs = args.filter((str) => {  
    const regex = /^--\w+$/;  
    const found = str.match(regex);  
    // console.log(found);  
    return !found;  
});  
  
const flagArgs = args.filter(str => !checkoutArgs.includes(str)); // separate flag args from hash
```

FIGURE 23 ARGUMENTS AND FLAGS ARE EXTRACTED AND SEPARATED AT THIS STAGE FOR FURTHER PROCESSING

The user provided arguments are then checked (differently depending on the command function that is being run) and the flags sent off for routing and re-routing to start marshalling of a params data object (depending on command being run) to be sent off to an appropriate gunDbUtils.ts query function.

```
if(checkoutArgs.length > 1){
  console.log("too many args");
  checkoutOptionsFlagRouter({"--help": null}, []);
}else if(checkoutArgs.length == 0 && flagArgs.length == 0){
  console.log("no args");
  checkoutOptionsFlagRouter({"--help": null}, []);
}else if(flagArgs.length > 0){
  const flagKeyValPairArr: object[] = flagArgs.map(str => turnFlagIntoKeyValPair(str));
  const initialKeyValPairFlagObject = flagKeyValPairArr.shift(); // this mutates original arr
  // @ts-ignore
  checkoutOptionsFlagRouter(initialKeyValPairFlagObject, flagKeyValPairArr);
}else if(checkoutArgs.length == 1){
  const hashFromUser: string = checkoutArgs[0];
  const songNode = await gunDBHelper.getSongNodeFromHash(hashFromUser);
  if(songNode){
    console.log("VALID HASH");
    await writeToSongAndUpdateHashCacheOrFallback(songNode, hashFromUser);
  }else{
    console.log("INVALID HASH");
  }
}

};

return await boundGoToOrFallback("", []);
```

FIGURE 24 THIS IF/ELSE STATEMENT HERE IS HANDLING VARIOUS EDGE CASES OF USER INPUT (ARGUMENTS AND FLAGS) AND CALLING THE APPROPRIATE CODE IN RESPONSE. EDGE CASES DIFFER DEPENDING ON THE COMMAND FUNCTION BEING CALLED

```
const checkoutFlagOptions = {
  "--help": async function(flagVal: string, otherFlagKeyValPairs: object[]){
    console.log("usage: checkout <hash?>");
    console.log("description: load song data into song.hs file");
    console.log("checkout flags are:");
    console.log("--new");
    console.log("--help");
    checkoutOptionsReRouteIfMoreFlagObjectsRemain(otherFlagKeyValPairs);
  },

  "not found": async function(){
    console.log("flag not recognised: try 'checkout --help'");
  },

  "--new": async function(flagVal: string, otherFlagKeyValPairs: object[]){
    const genesisSongNode = await gunDBHelper.getSongNodeFromHash(GENESIS_HASH);
    await writeToSongAndUpdateHashCacheOrFallback(genesisSongNode, GENESIS_HASH);

    checkoutOptionsReRouteIfMoreFlagObjectsRemain(otherFlagKeyValPairs);
  }
}

const checkoutOptionsFlagRouter = optionsFlagRouter.bind(checkoutFlagOptions, "not found");
const checkoutOptionsReRouteIfMoreFlagObjectsRemain = createReRouterIfMoreFlagObjectsRemain(checkoutOptionsFlagRouter);
```

FIGURE 25 THESE ARE THE FLAG HANDLER METHODS FOR THE CHECKOUT COMMAND, NOTE HOW THE SAME PATTERN OF ROUTING AND GOING TO A FALLBACK FUNCTION IF A FLAG IS NOT RECOGNISED HAS BEEN ADOPTED HERE AS THE OUTER COMMAND OBJECT

I think the activity diagram below perfectly illustrates how an command is routed then an array of flags is reduced down, processed and re-routed recursively until the array is empty and no items (flags) remain for

processing. This approach to handling, processing and reducing flags was inspired by Haskell’s functional approach to recursively computing arrays.

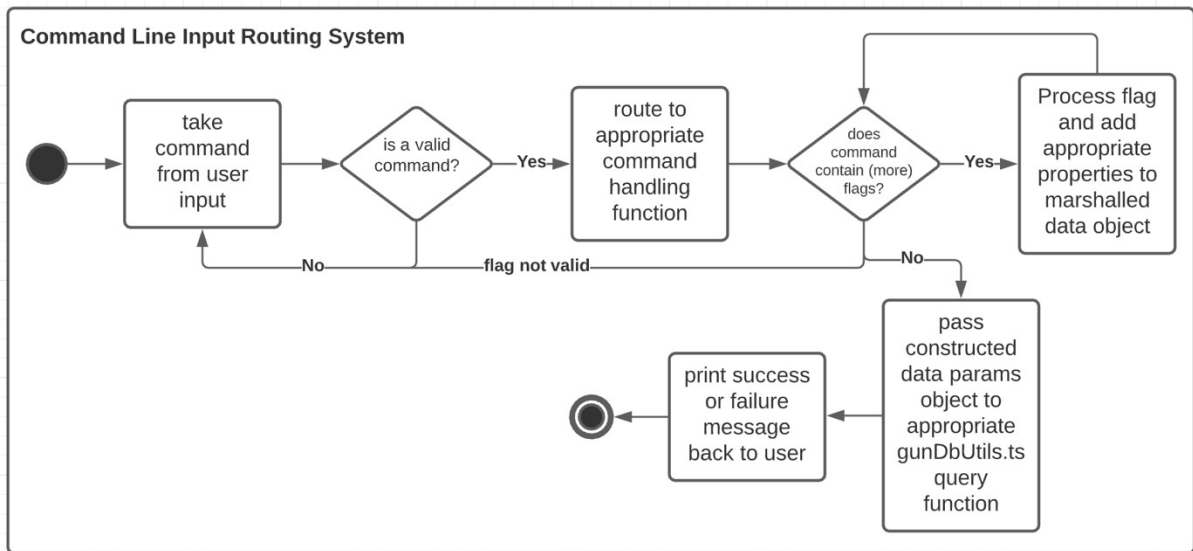


FIGURE 26 ACTIVITY DIAGRAM OF COMMANDS AND FLAGS ROUTING SYSTEM

Note: you may have noticed nearly all functions in code examples so far are preceded by “async”, this is key in ensuring all operations (where needed) behave in a synchronous ordered manner, code will not continue to execute for example until user input has been entered, and users will not be prompted to enter user input again until code has finished its orderly execution (which is not the default behaviour). this has to be carefully coded-in to avoid the asynchronous “callback hell” nature of JavaScript which would create a bad user experience.

Gun API and Graph Design

Graph data model design

Before writing the code I carefully designed and modelled the nodes and relationships (links/references between nodes) of the graph (the database design structure) with ease of traversal and future queries in mind.

GunDB by default creates relationships in one direction and doesn't force you to define properties for edges. It gives you the freedom to decide which edges need properties and which entities need bi-directional relationships. You have complete control when it comes to designing the graph model of your system (Meyghani, 2018)

A lot of parts in the graph have been designed to be purposely bi-directional which will speed up traversal significantly and reduce the complex code needed to write long compound style queries.

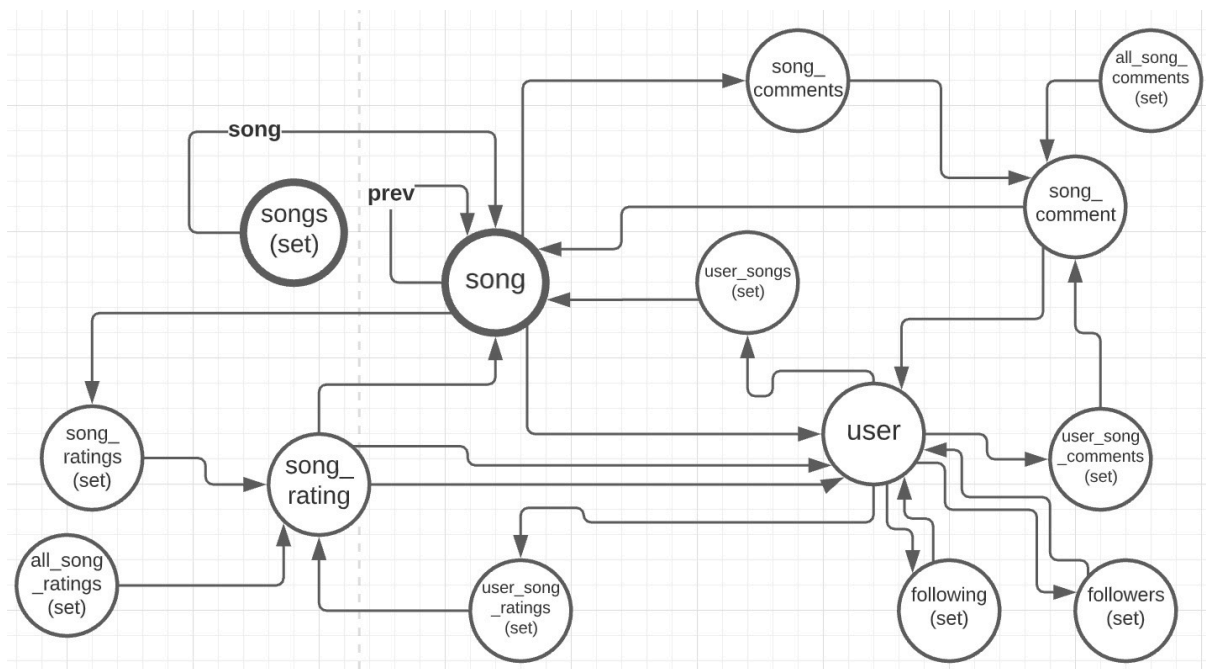


FIGURE 27 AN ENTIRE GRAPH MODEL OF WHOLE DATABASE, THE SONG NODE IS AT THE CENTRE OF THE DESIGN

I've made the border of nodes in the MVP bold (song and songs set), the system can run just with these two nodes in existence, the rest of the system can be developed and integrated in later. The current database design isn't really agile or flexible though in the sense that because song nodes are immutable, I can't suddenly add references to song_comments, song_ratings and user nodes later, so a slight re-design of this model could be done to make it more compliant with more of an agile style of development.

API

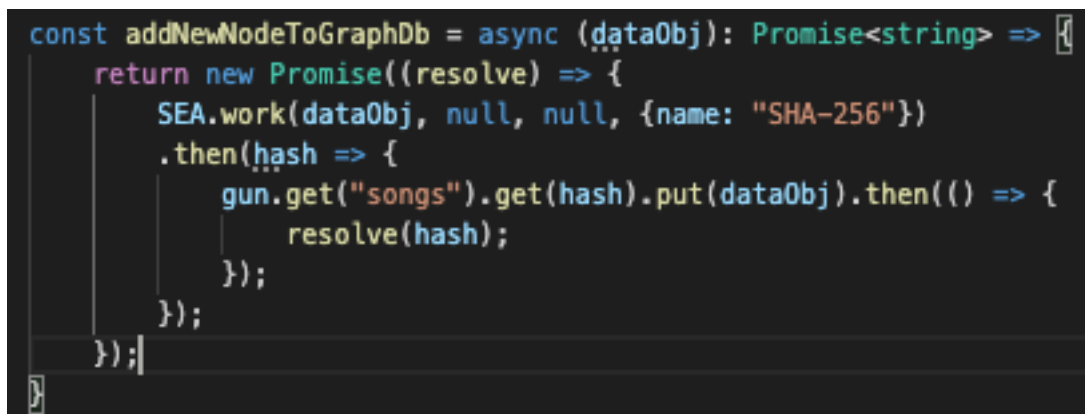
Currently in the application I'm using the following fairly simple aspects of the gun API:

- **Gun.get()** retrieves object nodes from the "songs" directory in the graph by it's hash string
- **Gun.put()** adds new song nodes into the graph under the "songs" directory/set
- **Gun.on()** subscribes to any changes that occur to the "songs" set/folder using a subscribe/publish model. In this case, users are notified when a new song has been added/published to the database, if nodes in the songs set from one peer don't exist in another peers graph, they can be checked for and added if they don't exist yet on another users machine, this is how graph data can essentially be synced between peers

Hashing and Adding Immutable Objects (Song Nodes) to Graph

A vital part of my system is that added song data is immutable once published, this is achieved through Gun.js's Security Encryption Authentication (SEA) API. I first get a hash of the object data (which also contains the Haskell file data as a string) using the SHA-256 algorithm, I then store the original object data with the hash as the node identifier under the songs set, the resulting hash is always returned as part of the function so it can be displayed to user upon successfully publishing a new song. Note: if another user publishes the same song with the exact same file data then the timestamp can be used to determine the first of it's kind, the previous hash reference will also refer to the original copy in which it came from.

The gun API ensures that if I try to modify or mutate the object data of the node located at the hash of that same data and the modification doesn't match the original hash the change will be rejected; this is how I achieve immutability of song data in the system.



```
const addNewNodeToGraphDb = async (dataObj): Promise<string> => {
  return new Promise((resolve) => {
    SEA.work(dataObj, null, null, {name: "SHA-256"})
      .then(hash => {
        gun.get("songs").get(hash).put(dataObj).then(() => {
          resolve(hash);
        });
      });
  });
};
```

FIGURE 28 THE FUNCTION THAT DEALS WITH HASHING SONG OBJECTS AND ADDING THEM TO THE DATABASE, THE HASH OF THE OBJECT IS RETURNED AS A PROMISE SO OTHER CODE CAN WAIT ON IT TO COMPLETE BEFORE PROGRESSING

Syncing of Data when Users are Offline

The hash of Root node of tree needs to be the same across all users from when they first download and start using the application even in offline mode, this way when users come online a union of different users tree states can be performed using the genesis node as the overlapping base. For this reason the genesis object node of the tree contains no time-stamp (null) to ensure the hash doesn't ever alter between runs.

```
const initDB = async () => {
  // create base song node ("like genesis block in a blockchain")
  const genesisHash = await addNewNodeToGraphDb(
    {
      artist: `euterpeaTree`,
      createTime: null, // -- technically could fix a time here
      songName: `genesis`,
      fileData: `https://www.euterpea.com/`,
      prev: null // this is the only song node in graph that can have prev = null ref
    }
  );
};
```

FIGURE 29 UPON INITIALISATION OF THE GUN DATABASE THE GENESIS ROOT OBJECT IS ADDED, THE HASH OF THIS NODE FROM THE MOMENT WHEN THE SYSTEM IS DEPLOYED SHOULD ALWAYS STAY THE SAME OR BE A CONSTANT. NOTE: FILEDATA IS SUBJECT TO CHANGE TO THE HASKELL BASE TEMPLATE FILE DATA

Advanced JavaScript Patterns used In Development of System

Here are some of the pattern's I deployed to aid in development and future maintenance of the system.

JavaScript "Module pattern"

JavaScript doesn't have the notion of data privacy that you often find in object-oriented languages such as Java, but you can simulate this behaviour using "closures" exposing only the data you wish to and hiding the rest, this is useful to avoid cluttering the global namespace and to make it explicit certain functions aren't meant to be used elsewhere. I used this pattern in the creation of the "cmdLinePrompt" (which is responsible for routing user input commands to functions) to prevent data access to state and functions that are only required within the object.

Hoisting

Unlike most languages JavaScript allows you to access variables before they've actually been executed (code below in the file), this is because it has two phases during it's "execution context": the creation phase where variables are loaded or "hoisted" into memory and the execution phase. There are rare edge cases where this language feature can actually be useful, one of which presented itself in the coding of the application. I needed to pass the "cmdLinePrompt" object into a function which returns another function for routing to commands or falling back to a chosen method but then within the methods of the object itself I need to reference and call that routing function, even though technically at the time the cmdLineObject and it's methods were created the routing function "didn't exist" yet. This JavaScript feature allowed me to design the command line routing system in the way that I previously explained in the section: Design and Structure Of Command Line System.


```

    play: async function() {
        // play a quick sample of any song by hash id
        // or just play song.hs if no argument specified
        // add --auto-play flag, --shuffle flag, --duration=10 flag (mea
        console.log("to be implemented: play will allow you to play cur
        return await boundGoToOrFallBack("", []);
    }

const boundGoToOrFallBack = goToOrFallBack.bind(cmdLineObjApi, "main");
return cmdLineObjApi;

```

FIGURE 30 THIS CODE DEMONSTRATES WHERE HOISTING HAS BEEN USED TO USEFUL EFFECT: NOTICE HOW THE "BOUNDGOTOORFALLBACK()" FUNCTION HAS BEEN CALLED BEFORE IT'S ACTUALLY BEEN DECLARED/RAN

Function currying to suppress initial function calls

I don't want the user to be bombarded with pings and messages upon start-up when the gun database is being initialised and adding the genesis node or other test nodes into the database/graph, but I do want them to be notified when they or others add/publish songs successfully afterwards, so I found a way to suppress these notification for an initial time period by using a function currying pattern and closure which returns a function of which will only run it's call-back function code after a specified amount of time (in milliseconds) has passed.

```

// curry this
function getFunctionThatOnlyRunsAfterNMilliSecs(milliSecs: number){
    let t = new Date().getTime();
    return function(callback){
        if(new Date().getTime() > t + milliSecs){
            callback();
        }
    }
}

```

FIGURE 31 THIS FUNCTION IS MENT TO CALLED AND CURRIED AND THE RESULTING FUNCTION THEN IS THE ONE THAT ONLY EXECUTES IT'S CALLBACK AFTER A SPECIFIED DURATION OF TIME

```

const surpressPassedFunctionCallsUntilInitialsecsPassed = getFunctionThatOnlyRunsAfterNMilliSecs(10000);

```

FIGURE 32 THE FUNCTION IS CURRIED HERE WITH THE SPECIFIED TIME OF 10 SECONDS (10000 MILLISECONDS)

```
gun.get("songs").on((songs) => {
  surpressPassedFunctionCallsUntilInitialsecsPassed(function() {
```

FIGURE 33 HERE I'VE SUBSCRIBED TO NEW SONGS BEING ADDED TO THE SONGS SET, BUT LOGS ABOUT THEM BEING ADDED ARE SUPPRESSED FOR FIRST 10 SECONDS

.bind()

bind allows you to reuse functions but where the "this" keyword within the function refers to different object contexts, this is used so the function that abstracts away routing with flag strings and flag key methods can be used with different flag option objects.

```
function optionsFlagRouter(fallbackFlagKey, flagKeyValPair: object, otherFlagKeyValPairs: object[]){
  const flagKey = Object.keys(flagKeyValPair)[0];
  // @ts-ignore
  if(doesKeyExist(this, flagKey)){
    // @ts-ignore
    return this[toLower(flagKey)](flagKeyValPair[flagKey], otherFlagKeyValPairs);
  }

  // @ts-ignore
  return this[fallbackFlagKey]();
}
```

FIGURE 34 REUSABLE FUNCTION THAT CAN BE BINDED TO DIFFERENT FLAG OBJECT OPTION CONTEXTS

Immediately invoked function expression (IIFE)

sometimes you want a function to run only once and immediately be invoked, this pattern is used at start/entry point of application to initialize the database and start running the command line module code asynchronously.

```
(async ()=>{
  console.log("welcome to ET (Euterpea Tree) CLI: dec
  // set .etHashCache hash to genesis hash at the sta
  await writeHashToJsonHashCacheFile(GENESIS_HASH);
  await gunDbHelper.initDB();
  await cmdLinePrompt.main();
  console.log("running");
})();
```

FIGURE 35 IIFE WRAPPER TO START RUNNING CODE WITH ASYNC AND AWAIT

Async/await

just as code in the command line module is made to await for asynchronous operations to finish before progressing with program execution the same had to be done with query related code in the `gunDbUtils.ts` file. By default the gun database api uses asynchronous callbacks but luckily it provides a `.then()` api method which you can chain onto the end of queries to wait for them to finish before moving on with the next computation. so I had to refactor all query code to use `.then()` and wrap all code inside of promises so they could be waited on from the command line module before continuing program execution and getting user input.

```
const getAllSongNodes = async (listOptionsObject: ListOptionsObject): Promise<any[]> => {
  return new Promise(function(resolve){
    const songData = gun.get("songs") // B
    .then(o => removeMetaData(o)) // C
    .then(refs => Promise.all(Object.keys(refs)
      .map(k => gun.get("songs").get(k).then(o => formatSongGunNodeForUserDisplay(o))
    .then(arr => arr.filter(songNode => !listOptionsObject.artist || songNode.artist =
    .then(arr => arr.slice(0, listOptionsObject.listSize))
    .then(result => {resolve(result)})); // E
  });
}
```

FIGURE 36 LIST ALL SONG NODES QUERY DEMONSTRATING THE USE OF `.then()` TO APPLY CERTAIN COMPUTATIONS SUCH AS FILTERING BY THE LIST OPTIONS OBJECT PASSED IN FROM THE COMMAND LINE MODULE

Analysis

Here I will test the command line interface to interact with the system MVP and test the flexibility and robustness of it against various edge and test cases to demonstrate how well the functionality fulfils the core requirements defined *User Stories (user requirements)* section. *Two peers/nodes are run as the command line application and tested in conjunction with each-other via multicast locally.*

```
welcome to Euterpea Tree CLI: decentralized music creation and distribution
with Euterpea (Haskell library)
Multicast on 233.255.255.255:8765
commands: 'help', 'init', 'list', 'checkout', 'publish', 'status' and 'play'
```

FIGURE 37 UPON STARTING THE APPLICATION YOU WILL BE PRESENTED WITH A LIST OF COMMANDS

```
[randomText
commands: 'help', 'init', 'list', 'checkout', 'publish', 'status' and 'play'
```

FIGURE 38 IF YOU TYPE A COMMAND THAT DOESN'T EXIST YOU WILL BE RE-ROUTED BACK TO MAIN AVAILABLE COMMANDS AGAIN AS DESCRIBE IN SECTION *DESIGN AND STRUCTURE OF COMMAND LINE SYSTEM*

```
play
to be implemented: play will allow you to play current contents of song.hs
or shuffle through all songs in gun database and play them for specified
durations of time each
commands: 'help', 'init', 'list', 'checkout', 'publish', 'status' and 'ccc
```

FIGURE 39 COMMANDS THAT HAVEN'T BEEN IMPLEMENTED YET WILL PROVIDE THE USER WITH A MESSAGE OF THE FUTURE FUNCTIONALITY THEY WILL PROVIDE

In the following sub-sections I provide example test cases for each of the implemented features and commands: *list*, *checkout* and *publish*.

The List Command: as a user I want to list songs on the tree, so that I can discover songs and the data (like hashes) associated with songs

```
[List
[
  {
    artist: 'euterpeaTree',
    creationTime: null,
    prev: null,
    songName: 'genesis',
    hash: 'dn7A3aNITH2eniCa20W7d3/bUI3Ducrtmd6Uq3/DHR4'
  },
  {
    artist: 'chrisMorris',
    creationTime: '05/11/2021, 02:28:57',
    prev: 'dn7A3aNITH2eniCa20W7d3/bUI3Ducrtmd6Uq3/DHR4',
    songName: 'coolAsACucumber',
    hash: 'WhXRCOJVdFxAP+Cwa+SjBRwkrHqYIGGNtJgCSxbGw10'
  }
]
commands: 'help', 'init', 'list', 'checkout', 'publish',
```

FIGURE 40 DEMONSTRATION OF DEFAULT BEHAVIOUR OF LIST COMMAND TO LIST ALL CURRENT SONGS IN THE SYSTEM, THIS ALSO SHOWS STRING HANDLING AND FORMATTING OF USER INPUT THAT MAY HAVE UPPERCASES AND LOWERCASES IN IT

```
[list --artist=chrisMorris
[
  {
    artist: 'chrisMorris',
    creationTime: '05/11/2021, 02:28:57',
    prev: 'dn7A3aNITH2eniCa20W7d3/bUI3Ducrtd6Uq3/DHR4',
    songName: 'coolAsACucumber',
    hash: 'WhXRCOJVdFxAP+Cwa+SjBRwkrHqYIGGNtJgCSXbGw10'
  }
]
commands: 'help', 'init', 'list', 'checkout', 'publish',
```

FIGURE 41 --ARTIST FLAG DEMONSTRATION TO FILTER SONGS BY ARTIST NAME

```
list --artist=bob
[]
commands: 'help', 'init', 'list', 'chec
```

FIGURE 42 HANDLING OF EDGE CASE WHERE THE ARTIST YOUR FILTERING BY DOESN'T EXIST

```
[list --size=1
[
  {
    artist: 'euterpeaTree',
    creationTime: null,
    prev: null,
    songName: 'genesis',
    hash: 'dn7A3aNITH2eniCa20W7d3/bUI3Ducrtd6Uq3/DHR4'
  }
]
commands: 'help', 'init', 'list', 'checkout', 'publish',
```

FIGURE 43 BASIC --SIZE FLAG TEST

```
[list --size=1 --artist=chrisMorris
[
  {
    artist: 'chrisMorris',
    creationTime: '05/11/2021, 02:28:57',
    prev: 'dn7A3aNITH2eniCa20W7d3/bUI3Ducrtd6Uq3/DHR4',
    songName: 'coolAsACucumber',
    hash: 'WhXRCOJVdFxAP+Cwa+SjBRwkrHqYIGGNtJgCSXbGw10'
  }
]
commands: 'help', 'init', 'list', 'checkout', 'publish',
```

FIGURE 44 CHAINING OF MULTIPLE FLAGS CAN BE DONE AS DISCUSSED IN *THE DESIGN AND STRUCTURE OF COMMAND LINE SYSTEM*

```
[list --size=1 --artist=bob
[]
commands: 'help', 'init', 'l-
```

FIGURE 45 HANDLING OF EDGE CASE WHERE --SIZE=1 BUT THE ARTIST BOB DOESN'T ACTUALLY EXIST

The Checkout Command: as a user I want to checkout a song on the tree, so that I can play it, view the code and alter its code before publishing

```
[checkout
no args
usage: checkout <hash?>
description: load song data into song.hs file
checkout flags are:
--new
--help
commands: 'help', 'init', 'list', 'checkout', 'p
```

FIGURE 46 IF NO ARGUMENT IS PROVIDED TO THE CHECKOUT COMMAND IT WILL REVERT TO THE --HELP FLAG ASSOCIATED WITH IT

```
[checkout --help
usage: checkout <hash?>
description: load song data into song.hs file
checkout flags are:
--new
--help
commands: 'help', 'init', 'list', 'checkout',
```

FIGURE 47 EXPLICITLY USING THE --HELP FLAG TO GET HELP INFORMATION FOR COMMAND

```
[checkout WhXRC0JVdFxAP+Cwa+SjBRwkrHQYIGGNtJgCSXbGw10
VALID HASH
STARTING SONG FROM USER TEMPLATE: "coolAsACucumber"
written to: /Users/chris/Documents/dissertation_msc_
updated hash cache
commands: 'help', 'init', 'list', 'checkout', 'publi-
```

FIGURE 48 CHECKING OUT A USER CREATED SONG BY HASH

```
{ } .etHashCache.json > ...
1 | [{"checkedOutSongHash": "WhXRC0JVdFxAP+Cwa+SjBRwkrHQYIGGNtJgCSXbGw10"}]
```

FIGURE 49 THE HASH IN ETHASHCACHE.JSON IS CHANGED (RE-WRITTEN) TO THAT OF THE SONG "COOLASACUCUMBER" AND THE CONTENTS OF SONG.HS IS CHANGED

```
[checkout --new
STARTING SONG FROM BASE TEMPLATE: "genesis"
written to: /Users/chris/Documents/dissertation_msc_project/euterpea-tree/song.hs
updated hash cache
commands: 'help', 'init', 'list', 'checkout', 'publish', 'status' and 'play'
```

FIGURE 50 THE --NEW FLAG IS A SHORTCUT FOR CHECKING OUT THE GENESIS SONG NODE BASE SONG.HS TEMPLATE

```
{ } .etHashCache.json > ...
1  { "checkedOutSongHash": "dn7A3aNITH2eniCa20W7d3/bUI3Ducrtmd6Uq3/DHR4" }
```

FIGURE 51 THE HASH IN .ETHASHCACHE HAS BEEN RE-WRITTEN BACK TO THAT OF THE GENESIS SONG NODE

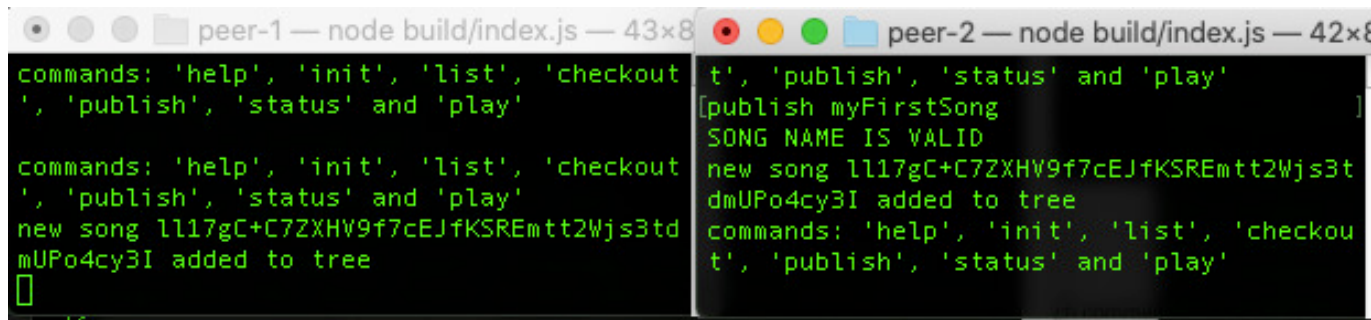
```
[checkout --new
commands: 'help', 'init', 'list', 'checkout', 'pu
STARTING SONG FROM BASE TEMPLATE: "genesis"
written to: /Users/chris/Documents/dissertation_m
updated hash cache
```

FIGURE 52 I ACTUALLY FOUND A BUG WHEN TESTING THE --NEW FLAG WHERE AVAILABLE COMMANDS ARE LISTED BEFORE THE ACTION OF CHECKING OUT A SONG IS COMPLETE; I FIXED THIS BUT HAVE INCLUDED IT HERE TO ILLUSTRATE THE CHALLENGES WITH WRITING ASYNCHRONOUS CODE AND WHY ASYNC/AWAIT IS NEEDED AS MENTIONED IN SECTION *ADVANCED JAVASCRIPT PATTERNS: ASYNC/AWAIT*

The Publish Command: As a user I want to publish a song to the tree so that I can share my music with others

```
[publish
no args
usage: publish <songNameInCammelCase>
description: adds current song (code of song.hs) to the tree (gun database)
you and other uses can discover it.
valid song names can only contain uppercase, lowercase, numbers and underscore.
no spaces are allowed.
example valid song names: mySong, my_song2, mySong2
publish flags are:
--help
commands: 'help', 'init', 'list', 'checkout', 'publish', 'status' and 'play'
```

FIGURE 53 PUBLISH REQUIRES AND ARGUMENT AND WILL RESORT TO ITS --HELP FLAG IF NO ARGUMENT IS PROVIDED



```
peer-1 — node build/index.js — 43x8
commands: 'help', 'init', 'list', 'checkout', 'publish', 'status' and 'play'
new song 1l17gC+C7ZXHV9f7cEJfKSREmtt2Wjs3tdmUPo4cy3I added to tree
[]

peer-2 — node build/index.js — 42x8
t', 'publish', 'status' and 'play'
[publish myFirstSong]
SONG NAME IS VALID
new song 1l17gC+C7ZXHV9f7cEJfKSREmtt2Wjs3tdmUPo4cy3I added to tree
commands: 'help', 'init', 'list', 'checkout', 'publish', 'status' and 'play'
```

FIGURE 54 HERE PEER-2 PUBLISHED THE SONG AND BOTH PEERS ARE NOTIFIED OF ITS ADDITION TO THE TREE IN REAL-TIME. WHEN A NEW SONG IS PUBLISHED ALL ONLINE USERS RUNNING THE APPLICATION WILL RECEIVE A PING (A LITERAL SOUND) AND NOTIFICATION FROM THE COMMAND OF THE HASH OF THE SONG THAT'S BEEN ADDED

Things to Consider with Performance

The system currently takes no more than a second to run each command however I've yet to test the speed of the system and its queries when thousands of song nodes are populated in the database and how it will hold up on a network when thousands of users are simultaneously running the application, the project is however approaching a state where performance testing can begin shortly but this is out of scope for this dissertation.

Conclusion

I've managed to develop the 3 key areas of functionality that are needed for interacting with and testing the system: list, checkout and publish. The system and user experience is robust enough now that user testing can begin and a relay server could be deployed to test the system online oppose to just locally.

I hope I've been able to demonstrate and illustrate my vision with this dissertation for a decentralised music creation system and tool that will benefit programmers and others who hope to learn about computer music coding. I feel I've made great strides and progress towards designing and creating an *educational system for programmers to learn about music and composition creation* and I hope they feel comfortable in sharing and publishing their music because of the systems I've put in place to prevent copying/stealing of credit of musical data. The prototype of the system will need to be deployed and tested next by real programmers in its current MVP state so I can gain valuable feedback about the direction I should prioritize next in its development; I've mentioned these potential future features and areas of the system to develop in section *Aims And Objectives: Long Term Goals out of Scope for this Dissertation*.

I Imagine a future where this system could have its own forks and graphs on different local networks for different schools and departments for people studying music and computer science and perhaps one public "global" network version.

The final submitted system MVP comes with comments in the code and a README explaining how to compile, run and test.

Learnings

During the research and development of this project here are some of the concepts and skills I've learnt:

- Advanced JavaScript patterns and techniques
- How to effectively write asynchronous JavaScript code
- Web 3.0 concepts and the basics of writing decentralised applications (dApps)
- Blockchain and decentralisation concepts
- About multiple libraries (gun.js, IPFS/PLD) and systems
- Merkle tree algorithms
- How Git works "under the hood"
- Music theory
- Improved Haskell programming skills and knowledge
- How write music as code using the Euterpea library

References

- Meyghani, A. (2018). *Data Modeling with GunDB*. Retrieved from Medium:
<https://medium.com/@ajmeyghani/data-modeling-with-gundb-15220cbfb8da>
- Nadal, M. (2021). *Gun Docs*. Retrieved from <https://gun.js.org/>: <https://gun.eco/docs/API>
- Quick, P. H. (2012). *The Haskell School Of Music: From Signals to Symphonies*. Yale University.
- Shao, S. H. (2020). *HASKELL ART IN YOUR BROWSER WITH ASTERIUS*. Retrieved from Tweag: Software Innovation Lab: <https://www.tweag.io/blog/2019-12-19-asterius-diagrams/>
- Vermaak, W. (2022). *What Is Web 3.0?* Retrieved from Alexandria:
<https://coinmarketcap.com/alexandria/article/what-is-web-3-0>
- Wikipedia. (2021). *Computer Music*. Retrieved from Wikipedia:
https://en.wikipedia.org/wiki/Computer_music

appendices

- Project Scrum Board: <https://trello.com/b/Ea8ivGP8/euterpea-tree>
- IPLD graph traversal code Gist: <https://gist.github.com/zoolu-got-rhythm/06dff4394ac0277fc854f808b26bdac8>
- IPFS/IPLD graph of song data code experiment Gist: <https://gist.github.com/zoolu-got-rhythm/3478c9241767d0c3f10301d75d53ea1c>
- Initial Gun.js experiment application for messaging between peers Github repository:
<https://github.com/zoolu-got-rhythm/gun-js-node-cmd-line-experiment>
- My Merkle tree JavaScript implementation, Gist: <https://gist.github.com/zoolu-got-rhythm/ffa7e902b96d00a1a40974244d845a78>