

# Change History

| Version | Changes |
| --- | --- |
| 0.1 | * First Iteration |
| 0.2 | * Added Glossary, Change History, Cover Page, Context. * Restructured and reworded Sections 1-7. * Expanded section 8. |

# Glossary

| Term | Definition |
| --- | --- |
| Framework | A platform for developing software applications. Contains in-built functionality to assist in the development of software. |
| Project Brief | Virtual Jukebox – Project Brief |
| Frontend | The frontend of an application is defined as what the users can see and directly interact with. |
| Backend | The backend of an application relates to the server-side tasks and database communication. |
| Database | A structured set of data held in a computer or server. |
| SDK  (Software Development Kit) | A collection of software development tools in one package |
| API  (Application Programming Interface) | A software interface which offers a service to other pieces of software. |
| Web Socket | A web communication protocol allowing two-way communication |
| RDBMS  (Relational Database Management System) | Software that enables the creation and management of a relational database. A relational database is a structured database supporting the ability for data to link from one table to another, to form relationships. This enables ease of multi-table queries. |
| SRID  (Spatial Reference Identifier) | Indicates a specific spatial reference system (i.e. a set of parameters to describe geometry). e.g. Longitude and Latitude. |
| ACID  (Atomicity Consistency, Isolation, Durability) | Principles for data reliability.  Atomicity - uninterruptible operations,  Consistency - data integrity after deletion, insertion and updating,  Isolation - transactions will not affect others,  Durability - changes guaranteed to remain. |

Contents

[**Change History**](#_heading=h.vsc8coomwzmu) **[1](#_heading=h.vsc8coomwzmu)**

[**Glossary**](#_heading=h.10yyhgnf9ftw) **[1](#_heading=h.10yyhgnf9ftw)**

[**1 Introduction**](#_heading=h.30j0zll) **[4](#_heading=h.30j0zll)**

[1.1 Purpose](#_heading=h.1fob9te) [4](#_heading=h.1fob9te)

[1.2 Audience](#_heading=h.3znysh7) [4](#_heading=h.3znysh7)

[1.3 Project Background](#_heading=h.2et92p0) [4](#_heading=h.2et92p0)

[**2 Context**](#_heading=h.tyjcwt) **[5](#_heading=h.tyjcwt)**

[2.1 Potential Technology Summary](#_heading=h.1t3h5sf) [5](#_heading=h.1t3h5sf)

[2.1.1 Application Requirements](#_heading=h.9m7sssutdp3f) [5](#_heading=h.9m7sssutdp3f)

[2.1.2 Frontend Framework](#_heading=h.2xplzsmg8mr1) [5](#_heading=h.2xplzsmg8mr1)

[2.1.3 Backend Framework](#_heading=h.2s8eyo1) [5](#_heading=h.2s8eyo1)

[2.1.4 Database](#_heading=h.3rdcrjn) [5](#_heading=h.3rdcrjn)

[2.1.5 Streaming SDK](#_heading=h.26in1rg) [5](#_heading=h.26in1rg)

[2.2 Potential Application Architecture](#_heading=h.3dy6vkm) [6](#_heading=h.3dy6vkm)

[**3 Web Application Frontend Frameworks**](#_heading=h.lnxbz9) **[6](#_heading=h.lnxbz9)**

[3.1 Application Criteria](#_heading=h.35nkun2) [6](#_heading=h.35nkun2)

[3.1.1 Learnability](#_heading=h.1ksv4uv) [6](#_heading=h.1ksv4uv)

[3.1.2 Resources](#_heading=h.44sinio) [6](#_heading=h.44sinio)

[3.1.3 Performance](#_heading=h.2jxsxqh) [7](#_heading=h.2jxsxqh)

[3.1.4 Mobile friendly](#_heading=h.99h7nsyer4jp) [7](#_heading=h.99h7nsyer4jp)

[3.2 Three Main Frameworks Researched](#_heading=h.z337ya) [7](#_heading=h.z337ya)

[3.2.1 React](#_heading=h.1y810tw) [7](#_heading=h.1y810tw)

[3.2.2 Angular](#_heading=h.4i7ojhp) [7](#_heading=h.4i7ojhp)

[3.2.3 Vue](#_heading=h.2xcytpi) [7](#_heading=h.2xcytpi)

[**4 Web Application Backend Frameworks**](#_heading=h.1ci93xb) **[8](#_heading=h.1ci93xb)**

[4.1 Application Criteria](#_heading=h.3whwml4) [8](#_heading=h.3whwml4)

[4.1.1 Learnability](#_heading=h.2rd0pe2j2ge1) [8](#_heading=h.2rd0pe2j2ge1)

[4.1.2 Resources](#_heading=h.wgd29o4438jf) [8](#_heading=h.wgd29o4438jf)

[4.1.3 Database Support](#_heading=h.7jbnjlxor3jz) [8](#_heading=h.7jbnjlxor3jz)

[4.1.4 Security](#_heading=h.8joqw9lq24fn) [8](#_heading=h.8joqw9lq24fn)

[4.2 Three Main Frameworks Researched](#_heading=h.qsh70q) [8](#_heading=h.qsh70q)

[4.2.1 Django](#_heading=h.4iiywmr4ndci) [8](#_heading=h.4iiywmr4ndci)

[4.2.2 Ruby on Rails](#_heading=h.bmrmid1ma9b4) [9](#_heading=h.bmrmid1ma9b4)

[4.2.3 ExpressJS](#_heading=h.u4xw1wttgsov) [9](#_heading=h.u4xw1wttgsov)

[**5 Database Systems**](#_heading=h.3as4poj) **[9](#_heading=h.3as4poj)**

[5.1 Application Criteria](#_heading=h.1pxezwc) [9](#_heading=h.1pxezwc)

[5.1.1 Performance and Scalability](#_heading=h.fa0pxd3444yj) [9](#_heading=h.fa0pxd3444yj)

[5.1.2 Security](#_heading=h.xco192d090qp) [10](#_heading=h.xco192d090qp)

[5.1.2 Difficulty](#_heading=h.o0asg1ky5dam) [10](#_heading=h.o0asg1ky5dam)

[5.2 Database Types](#_heading=h.49x2ik5) [10](#_heading=h.49x2ik5)

[5.3 Three Main Databases](#_heading=h.2p2csry) [10](#_heading=h.2p2csry)

[5.3.1 MySQL vs PostgreSQL](#_heading=h.147n2zr) [13](#_heading=h.147n2zr)

[5.4 Caching Systems](#_heading=h.po7vudysm4yw) [13](#_heading=h.po7vudysm4yw)

[5.4.1 Redis](#_heading=h.nylim1dl6r9k) [13](#_heading=h.nylim1dl6r9k)

[5.5 Object-Relational Mappers](#_heading=h.qxokmxps888l) [14](#_heading=h.qxokmxps888l)

[5.5.1 Node.js ORMs](#_heading=h.bjpuwliz1gt7) [14](#_heading=h.bjpuwliz1gt7)

[5.5.2 Python ORMs](#_heading=h.1at9kwvh7cqu) [15](#_heading=h.1at9kwvh7cqu)

[5.5.3 TypeORM Prototyping](#_heading=h.y67et07c2s9z) [15](#_heading=h.y67et07c2s9z)

[5.5.4 Django ORM Prototyping](#_heading=h.b8t0rwe7oxh7) [15](#_heading=h.b8t0rwe7oxh7)

[**6 Messaging Solutions**](#_heading=h.xdnm1eklz1ns) **[15](#_heading=h.xdnm1eklz1ns)**

[6.1 Application Criteria](#_heading=h.ihv636) [15](#_heading=h.ihv636)

[6.2 Pre-made Chat Services](#_heading=h.32hioqz) [16](#_heading=h.32hioqz)

[6.2.1 TalkJS](#_heading=h.1hmsyys) [16](#_heading=h.1hmsyys)

[6.2.2 SendBird](#_heading=h.41mghml) [16](#_heading=h.41mghml)

[6.2.3 Chat Engine](#_heading=h.2grqrue) [16](#_heading=h.2grqrue)

[**7 Music Streaming Platform**](#_heading=h.vx1227) **[16](#_heading=h.vx1227)**

[7.1 Application Criteria](#_heading=h.3fwokq0) [16](#_heading=h.3fwokq0)

[7.2 Candidate Music Streaming SDKs](#_heading=h.1v1yuxt) [17](#_heading=h.1v1yuxt)

[7.2.1 Spotify](#_heading=h.1s4zd3yggwh3) [17](#_heading=h.1s4zd3yggwh3)

[7.2.1.1. Overview](#_heading=h.uj9646493o06) [17](#_heading=h.uj9646493o06)

[7.2.1.2. Discussion](#_heading=h.75pjhuj68afv) [17](#_heading=h.75pjhuj68afv)

[7.2.2 Apple Music](#_heading=h.19c6y18) [17](#_heading=h.19c6y18)

[7.2.2.1. Overview](#_heading=h.neppn6dxzosa) [17](#_heading=h.neppn6dxzosa)

[7.2.2.2. Discussion](#_heading=h.3jkic6sbu7kd) [17](#_heading=h.3jkic6sbu7kd)

[7.2.3 Deezer](#_heading=h.3tbugp1) [18](#_heading=h.3tbugp1)

[7.2.3.1. Overview](#_heading=h.bdcz9jh66qvv) [18](#_heading=h.bdcz9jh66qvv)

[7.2.3.2. Discussion](#_heading=h.ee7vbi5jtnbd) [18](#_heading=h.ee7vbi5jtnbd)

[7.3 Comparison Matrix](#_heading=h.mmfe9b223ym0) [18](#_heading=h.mmfe9b223ym0)

[**8 Conclusion**](#_heading=h.nmf14n) **[19](#_heading=h.nmf14n)**

[8.1 Frontend Framework](#_heading=h.i34a3tqb31dh) [19](#_heading=h.i34a3tqb31dh)

[8.2 Backend Framework](#_heading=h.2g6ty3jnidwg) [19](#_heading=h.2g6ty3jnidwg)

[8.3 Database System](#_heading=h.j86o90db4s5u) [19](#_heading=h.j86o90db4s5u)

[8.4 Messaging Solution](#_heading=h.an1mxk31mfw9) [19](#_heading=h.an1mxk31mfw9)

[8.5 Music Streaming Platform](#_heading=h.o5xyi8nrcpam) [20](#_heading=h.o5xyi8nrcpam)

[**Appendix**](#_heading=h.37m2jsg) **[20](#_heading=h.37m2jsg)**

[3.x (Web Application Frontend Frameworks)](#_heading=h.1mrcu09) [20](#_heading=h.1mrcu09)

[4.x (Web Application Backend Frameworks)](#_heading=h.46r0co2) [21](#_heading=h.46r0co2)

[5.x (Database Systems)](#_heading=h.2lwamvv) [22](#_heading=h.2lwamvv)

[6.x (Music Solution)](#_heading=h.111kx3o) [24](#_heading=h.111kx3o)

[7.x (Music APIs)](#_heading=h.3l18frh) [24](#_heading=h.3l18frh)

[**References**](#_heading=h.206ipza) **[24](#_heading=h.206ipza)**

[3.x (Web Application Frontend Frameworks)](#_heading=h.4k668n3) [24](#_heading=h.4k668n3)

[4.x (Web Application Backend Frameworks)](#_heading=h.2zbgiuw) [24](#_heading=h.2zbgiuw)

[5.x (Database Systems)](#_heading=h.1egqt2p) [24](#_heading=h.1egqt2p)

[6.x (Music Solution)](#_heading=h.3ygebqi) [25](#_heading=h.3ygebqi)

[7.x (Music APIs)](#_heading=h.2dlolyb) 25

# 1 Introduction

## 1.1 Purpose

This report details the research that was conducted on various technologies for Virtual Jukebox. This document will summarise research findings, and outline technologies that were found to be suitable for the project

## 1.2 Audience

This report is aimed at the stakeholders involved in the development of Virtual Jukebox. The main stakeholder is Amristar, which is the supervisor and client of the project. Stakeholders also include the staff involved with Computing Capstone Project 1 at Curtin University, which act as the co-supervisors.

## 1.3 Project Background

Jukeboxes were one a common sight at bars, discos and any social gathering. They allowed users to choose a song from a playlist to add to a queue that would eventually be played at the gathering.However, due to the explosive rise of music streaming services, jukeboxes have since become nothing more than a novelty. Virtual Jukebox aims to reproduce the functionality of a real jukebox in a digital, web application format. Users will be able to connect to a local jukebox using a map or a QR code, and be able to pick songs from a playlist which will be added to the queue.

# 2 Context

## 2.1 Potential Technology Summary

### 2.1.1 Application Requirements

Virtual Jukebox must provide a mobile compatible website for the users to interact with the application, which will be accomplished using a frontend web development framework. Users must also be able to sign up as either a host or a guest and be able to log into their accounts. User data must also be persistent, and a database management system will be required to achieve this. To read and write data to a database, a backend framework will be required to serve as an interface between the client and the database. The backend will also be responsible for receiving events from connected users and broadcasting these events to all connected users in the same session to create a live-chat room and a voting system. Virtual Jukebox must also provide host user accounts with the ability to create their own rooms, and provide the functionality for playing music which will be achieved using a music SDK.

### 2.1.2 Frontend Framework

The most suitable frontend framework researched was Vue. As the project has a fairly low number of frontend requirements, a simple and effective framework such as Vue was researched to be suitable for the Virtual Jukebox project.

### 2.1.3 Backend Framework

The most suitable backend framework researched was Django. Django provides many of the applications requirements natively such as user authentication, provides protections against common security vulnerabilities, and was found to be the most similarly aligned with the current skills of the team.

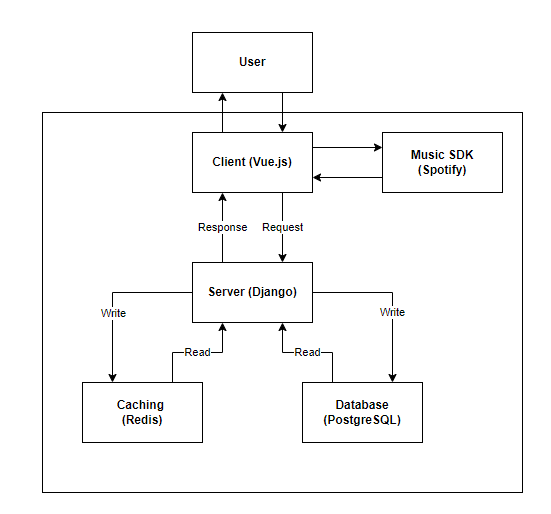
### 2.1.4 Database

The most suitable database system researched was PostgreSQL, due to a variety of reasons including more native encryption algorithms, more support for geo-location data type storage, more scalability support, and (when compared to other DBMSs including MySQL). Django ORM will be used alongside PostgreSQL to enable ease of integration with the Virtual Jukebox backend and its database. The ORM will also integrate without much additional configuration with the backend due to Django supporting Django ORM natively. This will remove the need to learn and implement an entirely different language to the backend when interacting with the database.

### 2.1.5 Streaming SDK

The most suitable music streaming SDK researched was Spotify. While all three candidate SDKs (Apple Music, Deezer, and of course, Spotify) met the Application Criteria to the same, acceptable level, where Spotify truly distinguished itself from the alternatives was the level of community engagement. There is no shortage of information (video tutorials, guides, blog posts etc.) in regards to development using the Spotify API. Therefore, developing the core application will be a much more straight-forward process. As a result, the time gained from working with a more familiar SDK will allow for more dedication to other aspects of development, namely, feature-adding, testing, and bug-fixing–all of which will contribute to providing users with a more enjoyable experience.

## 2.2 Potential Application Architecture



# 3 Web Application Frontend Frameworks

## 3.1 Application Criteria

There are some criteria that are needed from a framework that relates directly to how Virtual Jukebox works.

### 3.1.1 Learnability

The framework is required to be easy to learn and adapt, as it will need to be adopted by the group who may or may not have had experience using the framework. This criteria implies that the framework that will be chosen must have a low learning curve. The criteria is not entirely based on the framework itself, but it also considers the knowledge and experience of the members in the group. A framework that is hard to learn, but if most of the group members know how to use it, will be rated as a viable option instead of a non-viable option.

### 3.1.2 Resources

The framework having a large ecosystem and a high adoption rate is a criteria that ties into the learnability criteria as a framework that has a large ecosystem will also have an exhaustive amount of available resources that will be extremely useful for the development of Virtual Jukebox. A framework may be easy to learn, but have very little usage or resources available which would hinder the production of the application, this is why Resources and Learnability are separate criterias.

### 3.1.3 Performance

Virtual Jukebox will be playing music over a mobile/desktop, and this means that the framework chosen must be relatively fast as out-of-sync music playback is not wanted. Performance of the framework must be generally good overall, because the application has to have a smooth user experience based on load times.

### 3.1.4 Mobile friendly

The frontend framework must be able to make a web page mobile friendly on multiple mobile platforms. Most of the users will be using the application on their mobile devices and thus a framework that integrates well with a mobile view of a web page is extremely important.

## 3.2 Three Main Frameworks Researched

### 3.2.1 React

React is a perfectly suitable frontend framework that satisfies all given requirements of the Virtual Jukebox. Through prototyping, React was found to be relatively simple to read and understand. The team has limited React and JSX experience, but it was concluded through prototyping that React would be achievable to learn. React has an enormous community, and therefore a large number of resources, however React’s official documentation is generally seen as sub-par, being generally out of date[ref]. React is an extremely high-performance framework, and can easily support a large number of users and high interactivity[ref]. React is also mobile friendly, with all of the components being able to scale to a mobile device. React is the most popular frontend JavaScript framework available, and given that it satisfies all of the requirements of the Virtual Jukebox, it is a perfectly acceptable framework to use for the project.

### 3.2.2 Angular

Angular is a perfectly suitable frontend framework that satisfies all given requirements of the Virtual Jukebox. Through prototyping, Angular was found to be difficult to learn due to its complex and structured MVC approach, however excellent official documentation and resources led to the conclusion that Angular has mild learnability. As mentioned, Angular has a large ecosystem, excellent official documentation[ref], and therefore an extremely large set of online resources, satisfying the resources requirement. Angular is an extremely high-performance framework, and can easily support a large number of users and high interactivity[ref]. Angular is also mobile friendly, with all of the components being able to scale to a mobile device, especially the Angular-Material provided by Google[ref]. Angular is a feature-rich and complicated framework, given that it satisfies all of the requirements of the Virtual Jukebox, it is a perfectly acceptable framework to use for the project.

### 3.2.3 Vue

Vue is a perfectly suitable framework for frontend development although it is not as feature rich or as popular as Angular or React, but its main strength is its simplicity. Vue is considered the easiest framework to learn and get started with, boasting that you can have an app started “in minutes” [ref]. Through prototyping, Vue was found to be the easiest framework to learn and adapt. Vue has a relatively small ecosystem[ref] and number of resources, making it harder to find help for Vue specific issues, and harder to find useful extensions. Vue’s simplicity is seen generally as a downside due to it being more work than necessary to implement complex reactivity, and the flexibility of the framework usually leads to more errors than its React and Angular counterparts. Vue, like React and Angular, is seen as completely mobile-friendly, and its components can easily scale onto a mobile screen. Vue is generally seen as a perfect framework for simple, small scale apps, and for this reason, and the reasons above, Vue is considered a suitable frontend framework for the Virtual Jukebox.

# 4 Web Application Backend Frameworks

## 4.1 Application Criteria

Virtual Jukebox will require the retrieval of data from an external server and it will not be possible to have all processing occur on the client side of the application. Virtual Jukebox will require communication between a client and a server to allow for the client side of the application to provide updates in real-time. In order to implement features including the live chat room and users having the ability to vote for songs, the WebSocket API can be used. The WebSocket API is used to establish bidirectional communication between a browser and a server, which allows users to send an event to a server and receive responses to events rather than having to poll for a response. The WebSocket protocol is fully supported by all modern web browsers and is not constricted to any specific backend technology, but different backend technologies have different implementations for handling events and have access to specific libraries.

### 4.1.1 Learnability

The framework is required to be easy to learn and adapt, as it will need to be adopted by the group who may or may not have had experience using the framework. This criteria implies that the framework that will be chosen must have a low learning curve. The criteria is not entirely based on the framework itself, but it also considers the knowledge and experience of the members in the group. A framework that is hard to learn, but if most of the group members know how to use it, will be rated as a viable option instead of a non-viable option.

### 4.1.2 Resources

The framework having a large ecosystem and a high adoption rate is a criteria that ties into the learnability criteria as a framework that has a large ecosystem will also have an exhaustive amount of available resources that will be extremely useful for the development of Virtual Jukebox. A framework may be easy to learn, but have very little usage or resources available which would hinder the production of the application, this is why Resources and Learnability are separate criterias.

### 4.1.3 Database Support

The backend technology selected must also be able to connect to a database management system in order to store and retrieve user data. Due to the frequent access and modification of data required by the project, an ORM solution is essential.

### 4.1.4 Security

Virtual Jukebox will allow users to create and log into their own accounts. Users will be able to create accounts as a host or a guest, and each role will have access to different functionality. Authentication will be required to verify the identity of the individual attempting to login and restrict access to certain features and pages. Due to this, security will also be considered to prevent any possible data breaches.

## 4.2 Three Main Frameworks Researched

### 4.2.1 Django

Django was found to be a perfectly suitable backend framework for the Virtual Jukebox project given the requirements. Through prototyping, Django was found to be extremely simple to learn and adapt, especially when compared to Ruby on Rails and ExpressJS. Django has an extremely large ecosystem, including an enormous community with thousands of extensions[ref], and great up-to-date official documentation[ref]. Prototyping showed that Django has no issues connecting to a PostgreSQL database, and Django’s ORM solution was found to be satisfactory. Django was found to have an impressive number of in-built security features, providing protection against SQL injections, cross-site scripting and other common backend attacks[ref]. Overall, Django satisfies all of the requirements of the Virtual Jukebox project, and is therefore perfectly suitable as a backend framework.

### 4.2.2 Ruby on Rails

Ruby on Rails (RoR) was found to be a perfectly suitable backend framework for the Virtual Jukebox project given the requirements.Through prototyping, RoR was found to be moderately simple to learn, although less so than Django. Ruby’s flexible implementation allows the code to be easily modified and migrated, allowing for faster development[ref]. RoR has a large ecosystem, including a large number of community resources, extensions, and excellent official documentation. Prototyping showed RoR had no issues connecting to a PostgreSQL database, and the ORM solution of ActiveRecord was deemed to be satisfactory. Like Django, RoR comes built in with a large number of security features, protecting against a large range of common attacks[ref]. RoR is scalable, high performance and a perfectly suitable backend framework for the Virtual Jukebox project.

### 4.2.3 ExpressJS

ExpressJS is an extremely suitable backend framework that can be used in the development of the Virtual Jukebox given the requirements. Through prototyping, ExpressJS was found to have a moderate learnability. ExpressJS supplies the user with a debugger, which is an excellent tool for development, but is primarily written in NodeJS, which was found to have a decent learning curve. ExpressJS has an extremely large community and ecosystem, with a large number of resources available, however some of the modules that are frequently used have poor documentation. ExpressJS contains the library Socket.io, which if used, could simplify messaging requirements of the Virtual Jukebox[ref]. ExpressJS was found to support any database available[ref], satisfying the database support requirement. Furthermore, NodeJS and ExpressJS both have a large number of ORM solutions[ref]. ExpressJS has security capabilities, and can be a Secure backend framework to use, but it does not come built in with any built-in security features[ref] like RoR or Django. ExpressJS, uses Snyk which checks the vulnerability of the dependencies against the Snyk open sourced database[ref]. ExpressJS satisfies all the criterias needed for the Virtual Jukebox and is a backend framework that would work extremely well.

# 5 Database Systems

## 5.1 Application Criteria

The requirement of a database becomes apparent when it is observed that Virtual Jukebox involves multiple user types (guests and hosts), and (virtual) jukeboxes at multiple locations, each supporting playlists of songs. Specifically, the database will be responsible for storing user accounts including login credentials and user types, saved host playlists including their list of songs, and jukebox data including its host, chat logs per session, songs in the queue, a connection code, guest tokens, and location data (geographically based). The following criteria outlines the technology needs and considerations in the storage of this data:

### 5.1.1 Performance and Scalability

Although the scale of Virtual Jukebox will be small initially, it has the potential to be exposed to a vast number of users, hence requires sufficient scalability in data storage in terms of quantity, and data types. It will also need to be efficient for processing large volumes of data in short timeframes, therefore performance is a prime consideration. Caching will also be considered to provide quicker access to frequently used data. Indexes will additionally be utilised for access to frequently used data that must remain consistent/persistent as opposed to a cache which possesses risk of data loss due to on-memory storage.

### 5.1.2 Security

Security is another concern for Virtual Jukebox due to the fact that user data will be stored. Confidentiality will need to be addressed regarding the storage of user data, especially regards to authentication methods such as passwords. Encryption methods will provide security via data confidentiality, whereas databases with ACID compliance will enable integrity across all types of data when performing queries/operations. Accessibility must also be considered to prevent data leakage via authentication to databases.

### 5.1.2 Difficulty

As the development of Virtual Jukebox will take place within a restricted time-frame of approximately one year (half development, half planning), it is crucial that the difficulty of the database system selected does not possess a steep learning curve for ease of development. The system decided upon will need to possess enough features to be suitable for Virtual Jukebox requirements, while still being quickly learnable in comparison to database systems considered difficult.

## 5.2 Database Types

Two types of database types have been considered for use with Virtual Jukebox including relational databases (structured data driven), and NoSQL (unstructured data driven) databases. After consideration between the two, an RDBMS is preferred as most if not all data to be stored will follow a specific format, hence will be structured. Structured data is suitable for queries, which would be useful for application functionality such as returning all active jukebox locations, instead of having to extract unstructured data, then search for specific keywords manually. Although unstructured data support would support increased scalability (by not needing to comply with a defined schema), it is not required for the current specification of the software. It is always possible to introduce a NoSQL solution later on for any unstructured data storage requirement, whereas starting with a NoSQL system such as MongoDB would restrict or minimise the ability to query data initially. Additionally, many SQL systems contain support for NoSQL functionality such as JSON data storage in both MySQL and PostgreSQL, which could be used to assist in future scalability [5.6].

## 5.3 Three Main Databases

A total of eight databases were initially considered with the database requirements kept in mind. These systems included Oracle, MySQL, MS SQL Server, PostgreSQL, MongoDB, IBM DB2, Redis, and SQLite. After several analyses of these eight systems, three have stood out as being suitable for use in production.

The three database systems which seem to suit the requirements of Virtual Jukebox include MySQL, Redis, and PostgreSQL. As core query functionality and security is desirable, and the scope of the application is not substantial or focused around large-scale analytics, simpler databases are preferred over complex, feature-rich, proprietary ones. Oracle, MS SQL Server, and IBM DB2 were ruled out due to their difficult learning curve and their overwhelming feature-set [5.1]-[5.4]. SQLite was ruled out due to its lack of features, hindering any form of expandability, as well as its embedded nature which is not relevant in a web-server setting [5.5]. As relational databases seem more suitable in this scenario, MongoDB was ruled out. Redis is a consideration, however, due to its ability to be used in conjunction with other database systems as a caching tool to improve performance of frequently accessed data, even though it is a NoSQL system [5.3].

Redis could be used alongside a separate traditional RDBMS, simply for caching frequently accessed data such as chat-messages, session user-count, guest-credit count, and the songs in a queue. PostgreSQL is another consideration as although it is primarily an RDBMS, it offers support for JSON objects due to its object-relational nature. This would be helpful in the future, if the program would ever need to store unstructured data, without the need for a NoSQL implementation, while still providing all the RDBMS features including query access. MySQL is another option to consider due to its simplicity in implementation, with enough features to suit Virtual Jukebox’s data storage requirements. Each of these three systems are open-source, allowing for commercial usage without fees if needed [5.4]. They are also all relatively low in-terms of resource consumption, except for Redis which utilises random access memory more than the other two to obtain caching capabilities [5.4]. See below for a more in-depth comparison table between the three systems.

Points derived and summarised from [5.4]-[5.10]:

| **Criteria** | **MySQL** | **PostgreSQL** | **Redis** |
| --- | --- | --- | --- |
| **Utility** | - SQL RDBMS, which  aims to be fast and reliable with a limited feature-set  - Easy to learn  - Base edition is free. Other editions are paid | - SQL RDBMS with object support such as JSON  - Aims to provide large-scale performance and scalability with a larger feature-set  - Integration of functions made in programming languages (Java, C, etc.)  - Different user roles/privileges  - Completely Open-source | - NoSQL in-memory database/caching system  - Volatile (data stored in RAM) with options to enable non-volatility  - Key-value storage architecture  - Native data structures including strings, hashes, lists, sets, bitmaps, geospatial indexes, streams, etc.  - Open-source (BSD License) |
| **Performance** | - Faster in read-only operations (compared to other SQL DBMS)  - Memory Storage Engine provides frequently used table support  - Query cache for frequently used queries | -Faster with complicated queries on large data sets.  - Faster with read-write operations on large data sets with concurrent support (compared to other SQL DBMS)  - Table Partitioning can enable increased performance to frequently accessed parts of a table | - High memory (RAM) usage as it is an in-memory cache.  - Low disk usage if used solely as a volatile cache.  - In-memory cache provides very quick  read/write operations (no disk I/O needed except for non-volatility)  - Very fast reads with keys due to its key-value storage structure - O(1) via hashing. |
| **Scalability** | - Scales vertically (Requires more powerful hardware to upscale)  - Less features to support large-scale operations | - Scales vertically (Requires more powerful hardware to upscale)  - No restriction on database size  - Multi-Version Concurrency Control to support many concurrent users (via parallel queries).  - More indexing methods compared to MySQL | - Larger databases will require more RAM (Virtual memory not supported).  - Unstructured data support enables ease of addition of new data types/structures.  - Horizontally scalable (more machines and/or more RAM to upscale). |
| **Security** | - TLS encryption support | - Data encryption algorithms supported: AES, 3DES  - Native SSL encryption support for connections | -Potential data loss from storage of volatile in-memory data.  - Designed for use in trusted environments  - TLS encryption support |
| **Difficulty** | -Not case sensitive  - Considered easy to set-up and use  - Well-documented | -Case Sensitive  - Generally considered harder to set-up and use, largely due to its  inconsistent documentation. | - Considered easy/medium to set-up and use  - Mild learning curve compared to other NoSQL and SQL systems  - Commands are case-insensitive, but keys are case sensitive |
| **NoSQL Support** | - Later releases provide some NoSQL capabilities including JSON data management/storage. | - Some NoSQL support allows for the storage and management of unstructured data (JSON/key-value). | - Fully NoSQL without native SQL support.  - Unstructured data support |

### 5.3.1 MySQL vs PostgreSQL

For the RDBMS selection between MySQL and PostgreSQL, although MySQL would be sufficient and is generally considered easier to use according to [5.5], PostgreSQL is preferred due to a variety of reasons. Firstly, PostgreSQL is completely open-source, therefore all its features are available for any form of usage for free, whereas MySQL has fees for increased functionality. PostgreSQL and MySQL are both highly scalable, with both supporting multi-version concurrency control enabling efficient concurrent operations which would reduce the change of locking for when multiple users access the same data [5.6]. Concurrent operation support would be useful when multiple users of Virtual Jukebox are requesting songs and/or changing song priorities. Both PostgreSQL and MySQL provide several indexing methods, but PostgreSQL additionally supports expression indexing (for faster function calls in queries), and partial indexing (for specific queries based on a condition) [5.22]. PostgreSQL natively complies with ACID transaction properties, enhancing data integrity, although MySQL does too with the help of the InnoDB storage engine (which comes with MySQL by default) [5.23]. PostgreSQL also supports a range of data encryption algorithms such as AES and 3DES, as well as providing varied NoSQL support in key-value storage such as JSON indexing, and XML support, enabling storage using both dynamic and static schemas, whereas MySQL simply has JSON storage support [5.11]. This would be useful in the case of upscaling Virtual Jukebox to include additional features in the future which would require unstructured data storage.

5.5.2 Database Prototyping  
Pure SQL implementations for a sample database have been prototyped for both MySQL and PostgreSQL to get a feel of differences in syntax and database connection procedures. Judging from the syntax alone, commands in the PostgreSQL CLI are much more symbolic than the plain English-like commands that MySQL interprets. For example, showing all database tables in MySQL requires a “SHOW TABLES” command, whereas PostgreSQL requires a “\dt” command. Although this was observed, PostgreSQL showed more support for complex data types than MySQL, such as the storage of Virtual Jukebox’s jukebox locations. PostgreSQL utilises the PostGIS extension, which although requires a separate extension installation, provides a simple geographical coordinate representation using the “geography” data type. By default this datatype stores latitude and longitudes using a SRID of 4326, meaning it interprets coordinates with respect to the Earth’s surface. Although this was also possible for MySQL, it required a conversion from the “Point” datatype to a separate geometry data type, which then needed a manual declaration of its SRID. Other than this however, both database systems were very similar syntactically. See appendix 5.A for sample database schemas in each language.

## 5.4 Caching Systems

Caches are often used when users or software require frequent access to some data store that utilises disk I/O and/or web-based access methods. This is to enable faster access to such data, which provides an overall faster application by temporary storage on-memory instead of on-disk. Virtual Jukebox would benefit from such a feature for scenarios such as when guests are accessing data from the server multiple times in short time frames. For example, accessing the current song queue several times between requesting songs.

### 5.4.1 Redis

As an RDBMS is preferred for this software, and persistence is required, Redis is ruled out as a standalone/primary database system, however, it will still be used as a caching system alongside a traditional non-volatile SQL RDBMS to enhance performance, which is noted by [5.10] to be a common design pattern. As Redis will be used, disk I/O will be reduced when reading frequently used database contents due to its on-memory nature [5.10]. Contents that do not need to be frequently accessed or do not need real-time access with Virtual Jukebox, will not be stored in-memory using Redis, but will instead solely be kept in the primary RDBMS. Other contents that would require frequent access however, such as playlist contents, and queue contents/song priorities will be stored in the Redis on-memory cache, with scheduled writes to the disk for backups/persistence depending on the type of data. Redis also natively supports data structures such as sorted sets/lists which would be helpful in tracking queue song orders, and geospatial indexes which would be useful in storing active jukebox locations [5.9].

## 5.5 Object-Relational Mappers

Object-relational mappers (ORM) are software libraries that enable object-relational mapping when interacting with data from a traditional database system. Object-relational mapping refers to the process of writing database queries and accessing data (from a database) via the use of object-oriented techniques using a selected programming language, instead of relying on writing SQL queries [5.12]. By utilising an ORM, there is less need for writing complex SQL queries in database access and management by instead using calls to a library. ORMs typically form a connection between a software system’s backend, and its database system. By doing so, ORMs provide ease of use for developers not well versed in specific database systems, especially for the intricacies of query languages between database systems. An ORM has been selected for use within the backend Virtual Jukebox for a variety of reasons. Firstly, as PostgreSQL possesses a higher learning curve compared to other relational database systems [5.4], and is the prime candidate for Virtual Jukebox’s database implementation, an ORM will assist in reduced development time, especially in writing and accessing queries. Also, although ORMs typically reduce reading and writing performance (as a library call must be made to produce the corresponding SQL query), it realistically wouldn’t be a concern in the context of Virtual Jukebox as queries do not require complex logic (the most complex would be jukebox location queries). If the complexity of the software increases significantly, effort would be required in shifting from an ORM implementation to either a more efficient one, or to a pure SQL querying system.

### 5.5.1 Node.js ORMs

Three Node.js-based ORMs were selected for consideration including Sequelize, TypeORM, and Prisma. As each of these ORMs are Node.js-based, they will pair well with a node.js backend framework such as Express.js. Each ORM also supports PostgreSQL, and MySQL as well as many other database systems [5.13]-[5.15]. Prisma was ruled out due to its lack of functionality in supporting PostgreSQL’s PostGIS data types such as geometry types which would be useful for storing Virtual Jukebox’s jukebox locations [5.16]. Sequelize was also ruled out due to its lack of support, and poor documentation expressed in the community, which would be a detriment to the restricted time-frame of development [5.17]. TypeORM was selected to be the most suitable out of these three considered ORMs, due to its feature set, including the ability to be used alongside a Redis cache [5.18]. Due to the fact that ORMs abstract database interaction using a library as an interface, it will restrict complex SQL from being executed via the ORM which could affect ease of scalability if extended features are added in the future. TypeORM combats this by incorporating a raw query option, where raw SQL may be executed as per the base database system’s standard [5.19].

### 5.5.2 Python ORMs

Two Python-based ORMs taken into consideration include SQLAlchemy, and the Django ORM. For the case of a Python-based backend framework (such as Django), these ORMs will integrate well with it due to their shared language and support. Comparing the two, they are both suitable for Virtual Jukebox due to the ability to support PostgreSQL and MySQL, and both possess large communities with extensive documentation with SQLAlchemy having the most support [5.20]. This would be useful for speeding up development when issues arise regarding the ORM. Django ORM’s learning curve however, is considered much lower in comparison to SQLAlchemy due to its lower complexity [5.20]. Django ORM is a prime candidate for the ORM for use with Virtual Jukebox’s database due to its low difficulty, and its ability to model spatial data from PostGIS using GeoDjango [5.21].

### 5.5.3 TypeORM Prototyping

TypeORM has been prototyped with a simple implementation which maps entities (objects) in TypeScript to a local PostgreSQL database, using a similar schema to that of the final product (Users, Jukeboxes, Playlists, Playlist\_Songs). It has been tested via modelling data including spatial data types for the Virtual Jukebox’s jukebox locations via coordinates using PostGIS, as well as simple queries to retrieve such data. The first-hand experience of a simple implementation was functional, but fairly difficult as its documentation seemed lacking, with many aspects requiring research outside the documentation, such as utilising its API for inserting data with a foreign key. TypeORM foreign key declarations are also quite tedious as relationships must be explicitly defined on each table involved instead of simply defining the foreign key in a single table (for a two table 1:N relationship). Due to its complexity, TypeORM has been ruled out as an ORM for use in production. See appendix 5.B for the syntax required for defining TypeORM database functionality.

### 5.5.4 Django ORM Prototyping

Django ORM is in-built with the Django framework, making it easy to set up with a Django backend. After the development of an initial prototype in connection to a PostgreSQL sample database, it was observed that defining models (entities) was much simpler compared to the TypeORM ORM. This was due to less need of defining relationships at both ends, and the simplicity of defining geography data types, which involved a single key-word with the help of the GeoDjango library. Like TypeORM, it also supports raw SQL execution in the case that complex queries need to be executed. Compared to TypeORM, the Django ORM’s syntax is more readable, despite including all the necessary features needed for data modelling and database operations for Virtual Jukebox, making it very suitable for use in production. See appendix 5.C for the syntax required for implementing database functionality in the Django ORM.

# 6 Messaging Solutions

## 6.1 Application Criteria

Virtual Jukebox must provide a live chat-room for each session where users can communicate through instant messages in a group setting, with chat messages being programmatically moderated for offensive language.

## 6.2 Pre-made Chat Services

### 6.2.1 TalkJS

TalkJS is a pre-built chat API offering all the features that will be required in the application as well as many more. Features that are included in all subscription tiers but are unneeded include email/SMS notifications, replying via email, file transfers, and live location sharing.

Pricing begins at a minimum of $279 USD per month, with this increasing to $569 per month if a chatroom requires more than 30 users at a time.

### 6.2.2 SendBird

SendBird offers the creation of real-time communities with a live chat functionality. Features include sending live GIFs, audio, images, typing indicators, profanity filters, spam protection, and moderation tools for specific users.

SendBird offers a free trial version under a developer subscription, with up to 100 monthly active users, and up to 10 concurrent connections at a time. The cheapest subscription tier begins at $399 USD per month and allows for up to 2,000 concurrent users, and 100,000 maximum users per month.

### 6.2.3 Chat Engine

ChatEngine is a chat API used in the creation of instant chat messaging systems which can be used with a React.js front-end. ChatEngine aims to provide a simple service with a developer-friendly API and much lower pricing than other readily available services. ChatEngine provides the implementation of React components and hooks for use in assembling the user interface and managing state, as well as handling the implementation of a server that can be used to store messages and user accounts.

The cheapest pricing plan begins at $21 USD per month per 3000 users and 50,000 messages. Features include Secure Sockets Layer (SSL) encryption and storing chat logs for 30 days.

# 7 Music Streaming Platform

## 7.1 Application Criteria

Virtual Jukebox will require the Playback Controller of any candidate SDK to provide the functionality to:

* Play and pause track playback
* Seek to a given timestamp
* Skip to the next track
* Skip to the previous track
* Adjust volume
* View playback history
* View the songs in the queue
* Append a song to the queue

In addition, Virtual Jukebox will, and will likely require the API of any candidate SDK to provide the functionality to:

* Create a playlist in the User’s library
* Retrieve a playlist from the User’s library
* View the songs in a playlist from the User’s library
* Search for a song in a playlist from the User’s library
* Search for a song to add to a playlist from the User’s library
* Add a song to a playlist from the User’s library
* Remove a song from a playlist from the User’s library
* Sort the songs in a playlist from the User’s library by artist
* Sort the songs in a playlist from the User’s library by genre
* Sort the songs in a playlist from the User’s library by year

## 7.2 Candidate Music Streaming SDKs

The first candidate music streaming SDK is Spotify SDK (which includes the Web Playback SDK and Web API offerings). Next, Apple Music SDK (which includes the MusicKit JS SDK and Apple Music API products). Lastly, Deezer (which includes the JavaScript SDK and API).

### 7.2.1 Spotify

The Spotify Web Playback SDK is an SDK designed and implemented using JavaScript which allows Developers to integrate the streaming playback of Spotify content into their web applications [7.2].

Decisively, the level of engagement with the Spotify Platform services, both from Spotify itself and Developers, is the single largest contributing factor to its potential selection. The documentation for both the Web Playback SDK and Web API are extensive to say the least, complete with code examples, troubleshooting information and guides, along with much more information. As for Community engagement, there is a plethora of video tutorials discussing all aspects of development with the SDK–from user authentication, making requests, and integrating with a JavaScript based frontend framework such as Vue or React.

An important consideration, however, is the rate limits of the API itself. Unfortunately, however, there does not seem to be any clear indication as to what the limit is nor how it is calculated. As per Rate Limits, ‘Spotify’s rate limit’, “Spotify’s API rate limit is calculated based on the number of calls that your app makes to Spotify in a rolling 30 second window. If your app exceeds the rate limit for your app then you’ll begin to see 429 error responses from Spotify’s Web API, and you may hear from users about unexpected behaviour that they have noticed while using your app. The limit varies depending on whether your app is in development mode or extended quota mode.” Given the ostensibly low usage requirements for Virtual Jukebox, it appears as though rate limits will not be a concern. However, if any issues were to arise, Spotify discusses strategies for developing the app with rate limits in mind; they will be reviewed if deemed necessary during development.

It was discovered that any and all usages of the Spotify platform must be limited to a strictly non-commercial capacity. As per Spotify Developer Terms, ‘Section III Licenses and Permissions’, “1. License to Developer. Subject to and conditional upon your compliance at all times with the Developer Agreement, Spotify grants to you a limited, non-exclusive, non-transferable, non-sublicensable, revocable right to do the following: (a) Streaming and Non-Streaming SDAs. Use the Spotify Platform to develop and distribute Streaming SDAs and Non-Streaming SDAs that comply with the Branding Guidelines: (i) for private personal use; and (ii) on Approved Devices; and (iii) in accordance with the Developer Policy.”

### 7.2.2 Apple Music

With the Apple Music SDK, which includes the MusicKit JS framework and the Apple Music API, Apple Music users can access content from the Apple Music catalogue as well as their Library through JavaScript applications. Once a user permits access to their Apple Music account, the Developer application allows users to create playlists, add songs to their library and play the entire contents of the Apple Music catalogue directly through the browser the application is running on, all without the usage of any additional plugins. [7.3].

Unlike both the Spotify SDKs, the Apple Music SDK features the ability to directly access the Apple Music API through the playback controller, rather than forcing Developers to manipulate both elements separately. From a development standpoint, it will allow for much cleaner, concise code which may bear great significance as we further develop the application in regards to feature adding, testing and bug-fixing as it could potentially decrease core application development time, allowing greater development time for enhancing user experience through the addition of supplementary features, testing and bug-fixing–much like what was discussed in regards to the Spotify SDK. However, unlike the Spotify SDK, the level of engagement with the Apple Music SDK is significantly lower. The Documentation for the SDK services is comparable to that of Spotify, but there is far less information (video tutorials, blog posts, guides etc) available from third-party sources.

Once again, it is important to consider request rate limits. Much like the Spotify API, the Apple Music API documentation doesn’t offer any concrete information in regards to rate limits. As per Generating Developer Tokens, ‘Request Rate Limiting’, “Apple Music API limits the number of requests your app can make using a developer token within a specific period of time. If this limit is exceeded, you’ll temporarily receive 429 Too Many Requests error responses for requests that use the token. This error resolved itself shortly after the request rate has reduced.” However, as previously mentioned, the usage requirements for Virtual Jukebox do not indicate that rate limits will be of any concern at all.

It was discovered that any and all usage of the Apple Music platforms must be limited to that of a strictly non-commercial capacity. As per the Apple Developer Program License Agreement, ‘MusicKit’, “...If you access the MusicKit APIs or MusicKit JS, then you must follow the Apple Music Identity Guidelines. You agree not to require payment for or indirectly monetize access to the Apple Music service (e.g. in-app purchase, requesting user info) through your use of the MusicKit APIs, MusicKit JS, or otherwise in any way.”

### 7.2.3 Deezer

The Deezer SDK is a JavaScript SDK that allows developers to integrate functionality with the Deezer services through their web-based JavaScript applications. With the SDK, developers can authenticate users, stream music and access the Deezer API endpoints.

Exactly like the Apple Music SDK, the Deezer SDK also allows Developers to query the Deezer API through a playback controller running on their website. Again, this could potentially allow for more maintainable code, providing the same future benefits in regards to ongoing development that were discussed in regards to the Apple Music API. However, the Deezer SDK suffers from the same relatively low engagement levels that the Apple Music SDK does. The SDKs documentation is more user-friendly than that of the Apple Music SDK but not as detailed as the Spotify SDK documentation. Furthermore, there is more information available in regards to usage guidance with the Deezer SDK. Namely, through the facilitation of guides on the Deezer website but there isn’t the same level of third party support that is enjoyed by the Spotify SDK.

In addition, it is important that we discuss rate limits. As per Deezer FAQs for developers, ‘III. Search endpoint’, “there is no limitation on data in the API, but there is a query quota...”, however, upon further research into the query quota Deezer offers no further information. This is far less than ideal as it would be beneficial to at the very least have an idea of the behaviour of the API at higher loads to keep in mind during development, however, no such information has been provided.

Much like both the Spotify and Apple Music Platforms, usages of the Deezer Platform must, too, be limited to that of a personal capacity. As dictated by the Deezer Terms of Use, Terms of Use of Deezer for Developers, subsection ‘IV. Non-Commercial Use’, “The Developer agrees that the use of the Services is strictly limited for a non-commercial purpose and in a non-commercial environment. It means that the Developer shall not perceive, receive, generate, benefit or create directly or indirectly, any moneys, incomes, revenues, data or any other consideration in connection with the use of neither the Services themselves, nor any and all Content accessed through the Services…The Developer undertakes to inform by any means any person accessing the Content through the Webpages and/or Personal Applications of its conditions of use and notably that the streaming of the recordings is limited to a strictly private use within a family scope...”

# 8 Conclusion

## 8.1 Frontend Framework

Angular, React and Vue were found to be perfectly suitable for the Virtual Jukebox project. Through extensive research, it was found that the strengths of Vue were most suited to the requirements of the project. The Virtual Jukebox project has a fairly low number of frontend requirements, therefore a simple and effective framework such as Vue is to be the most appropriate. Vue will be able to satisfy the requirements of the Virtual Jukebox project

## 8.2 Backend Framework

Django is the most suitable backend framework for use in the development of Virtual Jukebox. Although all the researched backend frameworks are compatible with the requirements of Virtual Jukebox, Django was found to be preferable in certain areas. Django was viewed positively due to its protection against common web application security vulnerabilities such as cross-site scripting and SQL injection, allowing for more focus on the development of the required features of Virtual Jukebox. Django also has extensive and detailed documentation, and provides support for use with a PostgreSQL database.

## 8.3 Database System

After prototyping and a comparative analysis between MySQL and PostgreSQL, Virtual Jukebox will utilize the PostgreSQL database system to enable persistent data storage of aspects including user data, jukebox data including their locations, and playlist data. This is primarily due to PostgreSQL’s extended support in geographical location support (with the PostGIS library), as well as its pure open-source licensing, and compatibility with a wider range of data types compared to MySQL. Although PostgreSQL is often considered more difficult to learn, that will be combated with the usage of the Django ORM which will pair well with the Django backend due to its native integration. Django ORM enables an easier process of development in database interaction due to the lack of needing to write SQL queries, and the linkage of objects to database tables and entities. Additionally, Virtual Jukebox will incorporate Redis caching to be used alongside PostgreSQL in order to provide quicker access to frequently used data.

## 8.4 Messaging Solution

Chat Engine is the most suitable option out of the pre-made solutions as it offers all of the features that are required for the application without any extra unnecessary features, however the provided interface components are only available when using a React.js frontend. Due to SendBird and TalkJS charging much higher prices which increase as the scale of the application increases, these are deemed unsuitable as they also offer many features which are unneeded in the current specifications. A chatting system will be manually implemented utilising the websocket API on the client side, and message events will be handled using Django Channels in the backend of Virtual Jukebox.

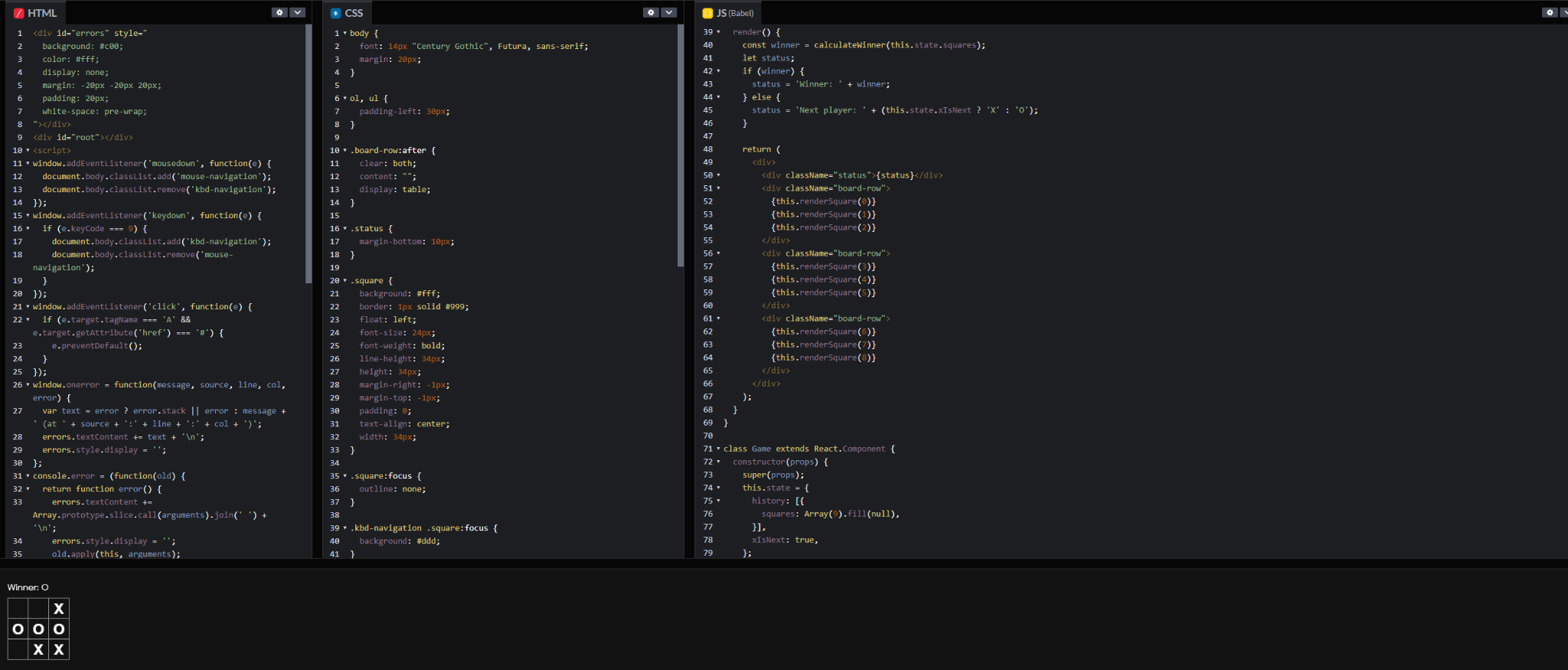
## 8.5 Music Streaming Platform

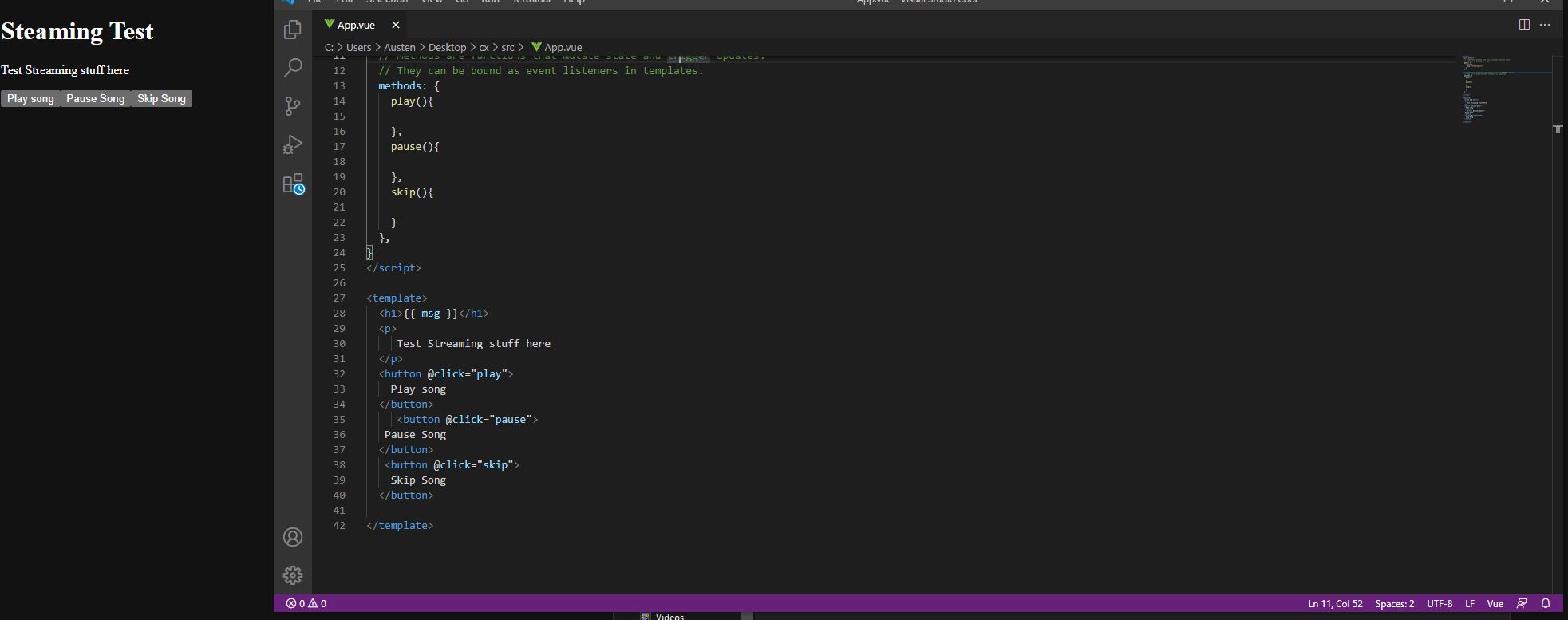
With all aspects of the candidate streaming SDKs considered, it was found that the most appropriate SDK is Spotify. The SDK distinguishes itself from the alternatives through its vastly superior level of engagement. Developers utilising the SDK benefit from a plethora of information (video tutorials, guides, blog posts etc.) in regards to development using the Spotify API. Therefore, developing the core application will be a much more straight-forward process. As a result, the time gained from working with a more familiar SDK will allow for more dedication to other aspects of development, namely, feature-adding, testing, and bug-fixing–all of which will contribute to providing users with a more enjoyable experience.

# Appendix

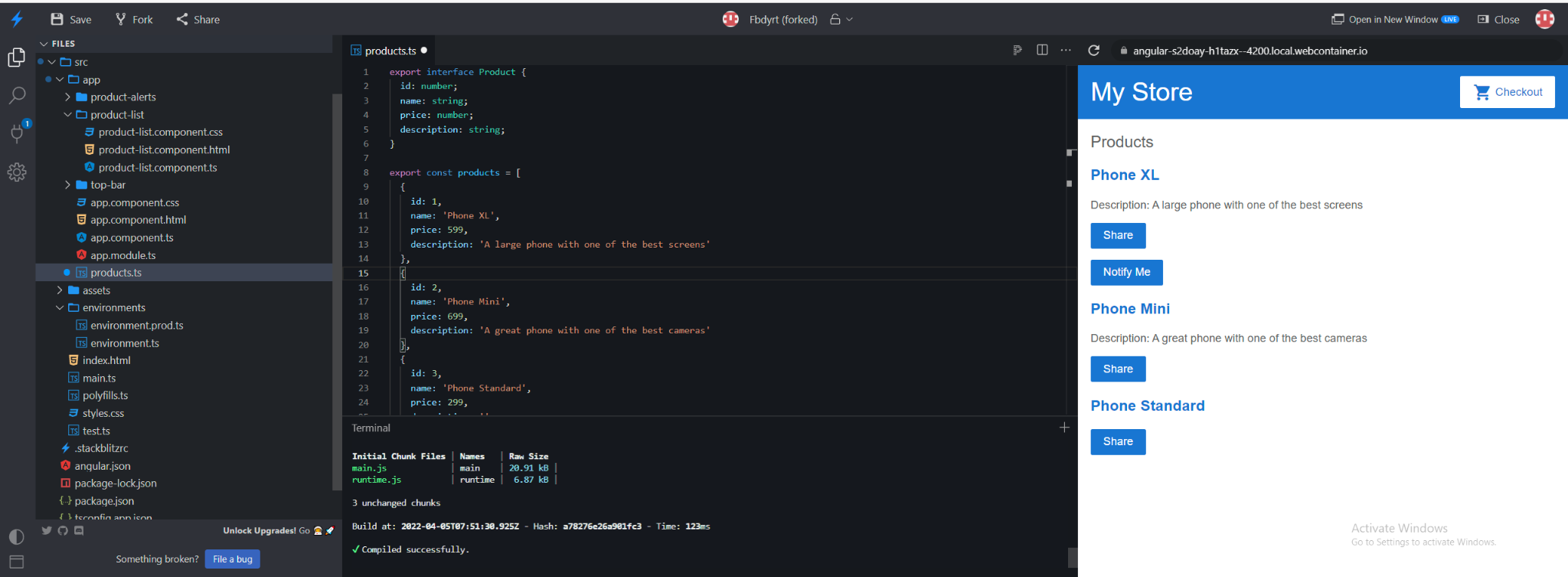
## 3.x (Web Application Frontend Frameworks)

React Prototyping

  
Vue Prototyping

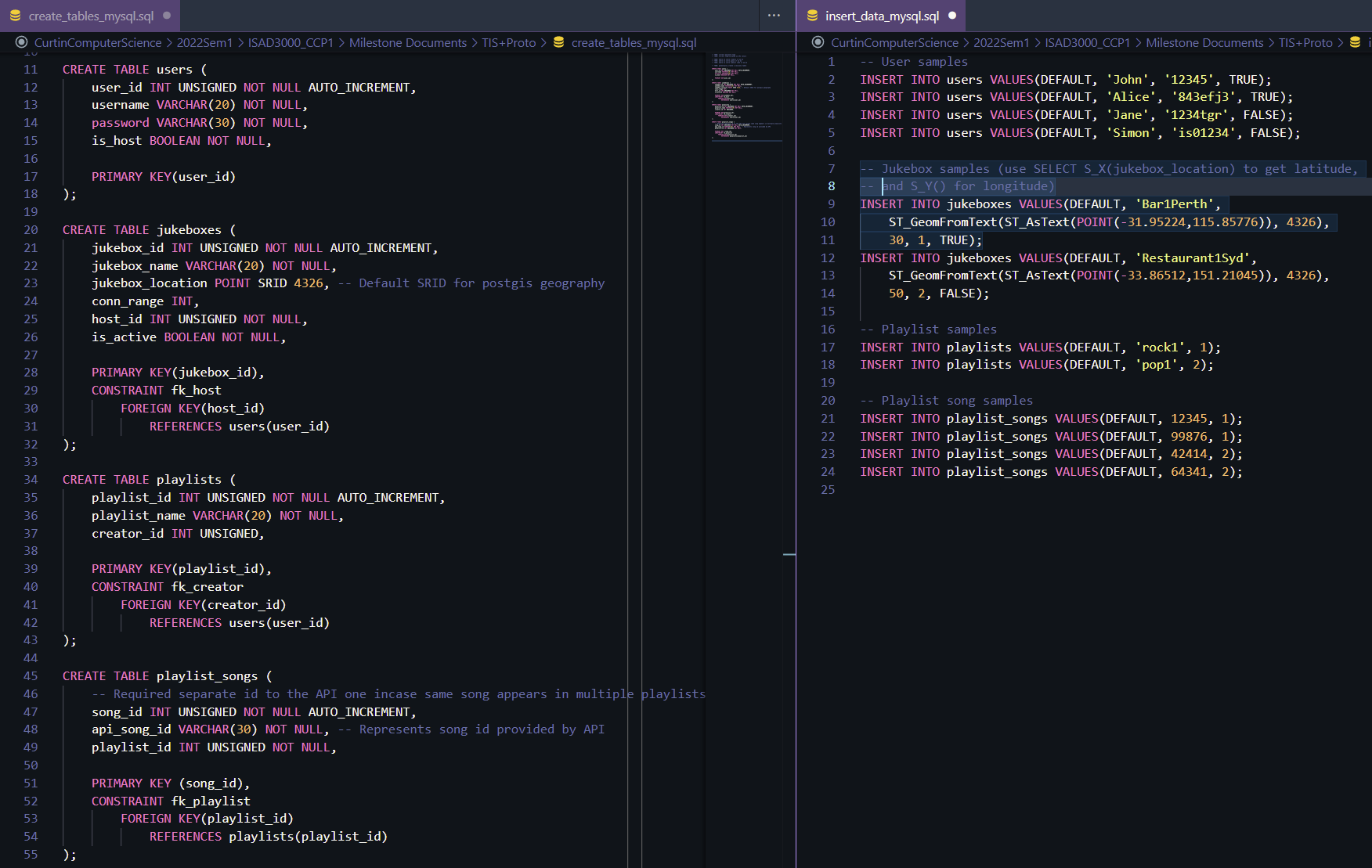


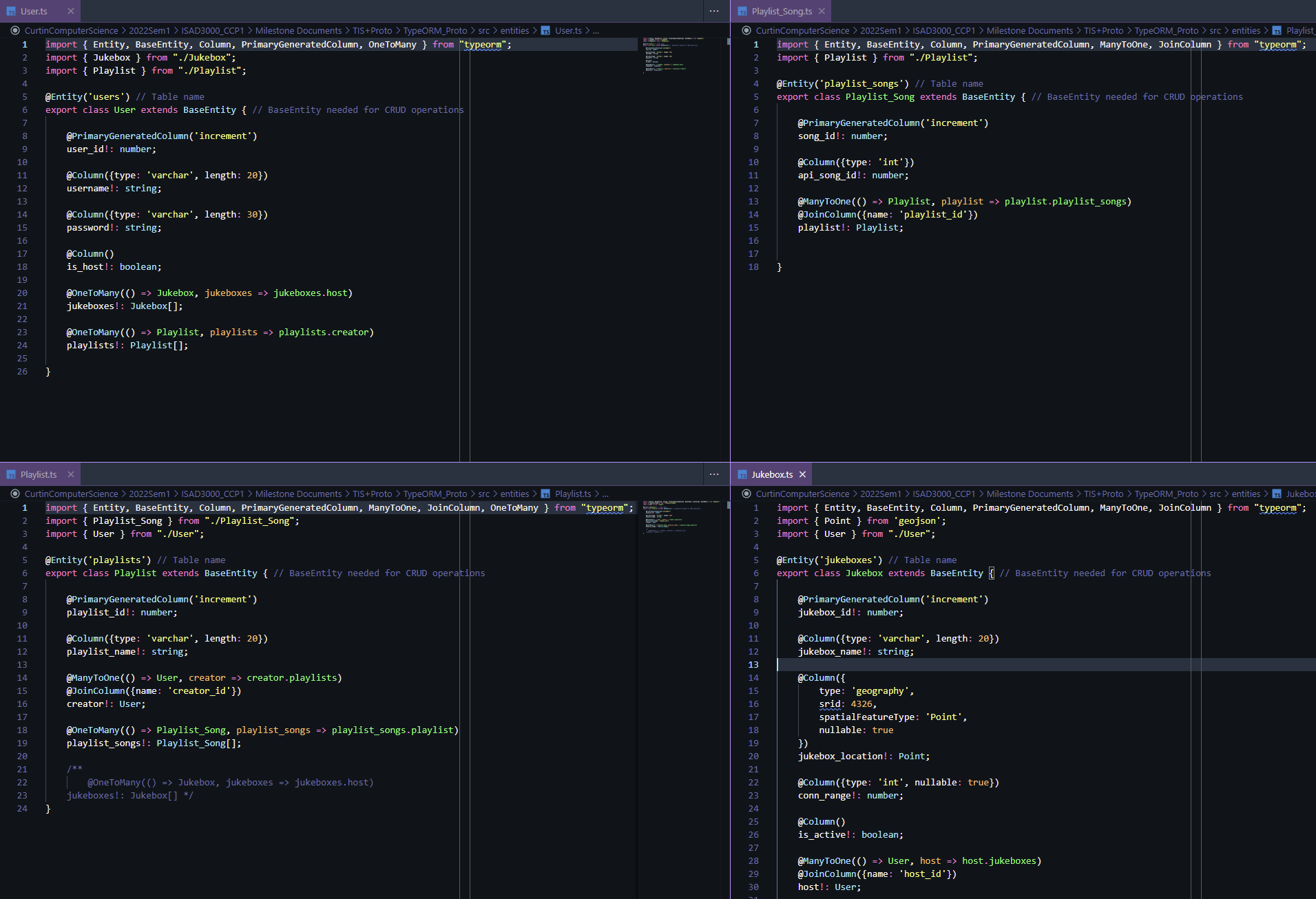
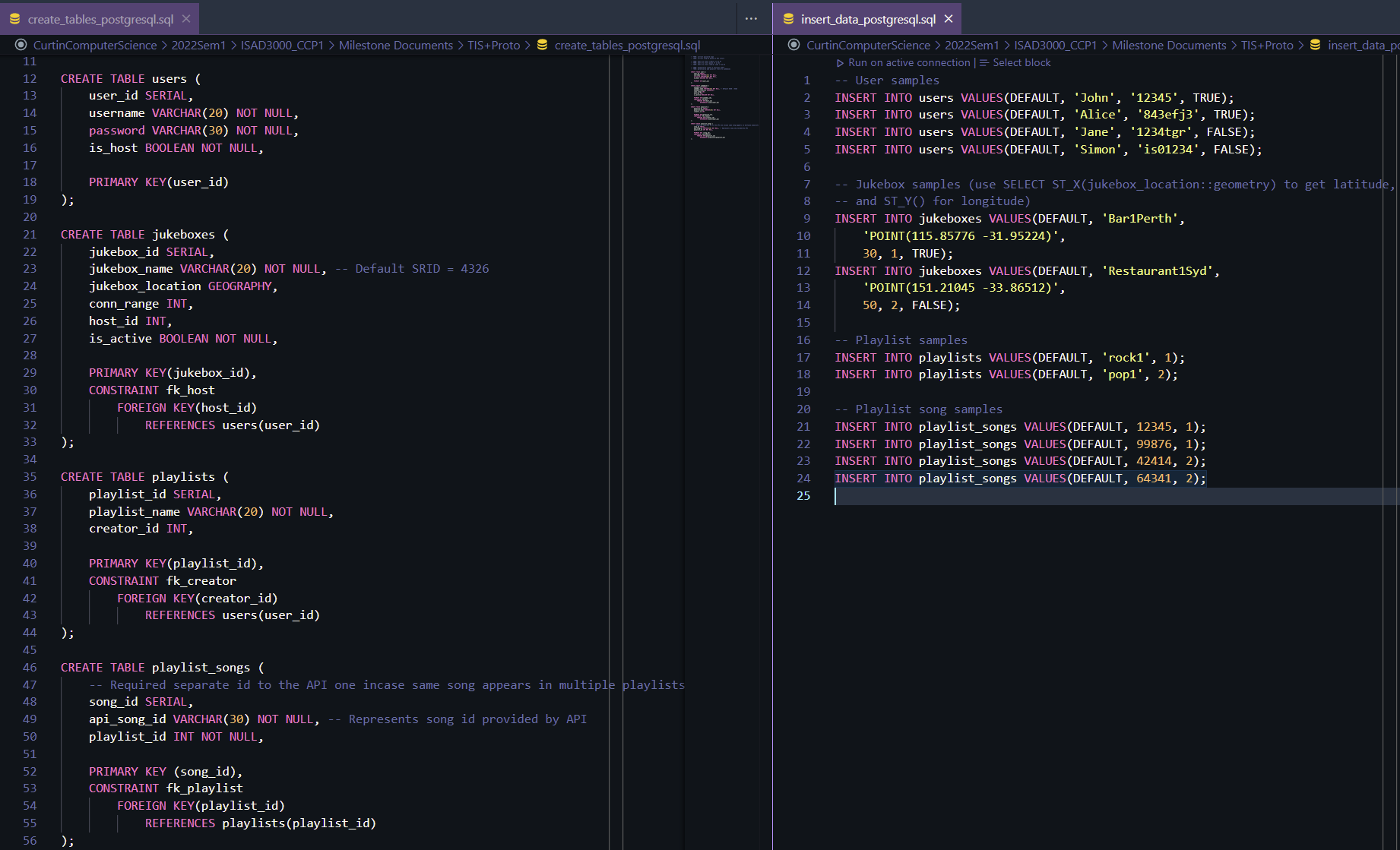
Angular Prototyping



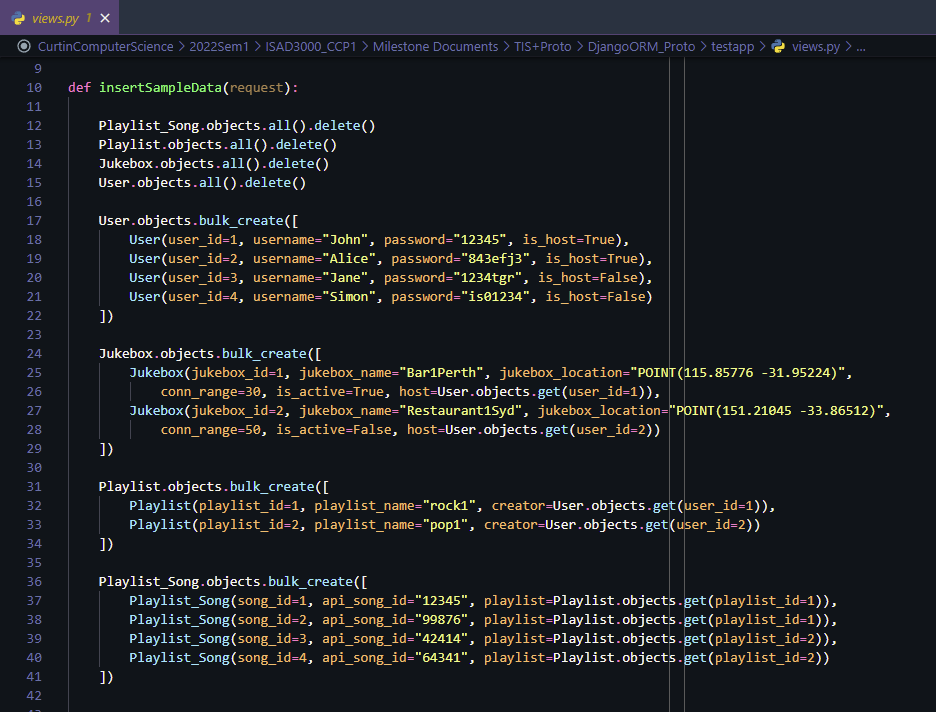
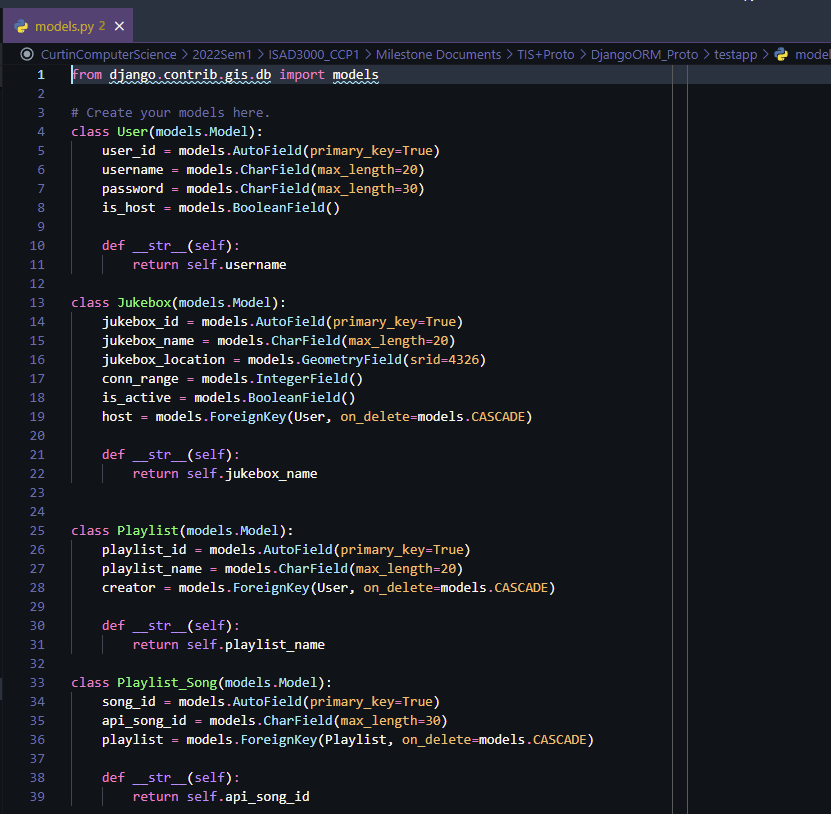
## 4.x (Web Application Backend Frameworks)

## 5.x (Database Systems)

(5.A) Pure SQL Database Schema and inserts, note that the schema is simply a prototype. Production will require hashed passwords and more tables depending on the definition of the Software Requirements Specification.  
**MySQL**

**PostgreSQL**

(5.B) TypeORM schema implementation

(5.C) Django ORM schema implementation and sample data insertion

## 6.x (Music Solution)

## 7.x (Music APIs)

# References

## 3.x (Web Application Frontend Frameworks)

## 4.x (Web Application Backend Frameworks)

## 5.x (Database Systems)

[5.1]C. Arsenault. “The Pros and Cons of 8 Popular Databases.” Keycdn.com. https://www.keycdn.com/blog/popular-databases (accessed Mar. 23, 2022).

[5.2]Idslogic. “The Pros and Cons of Some of the Most Popular Databases.” Idslogic.com. https://www.idslogic.com/blog/2019/09/the-pros-and-cons-of-some-of-the-most-popular-databases.html (accessed Mar. 23, 2022).

[5.3] M. Kamaruzzaman. “Top 10 Databases to Use in 2021.” TowardsDataScience.com. https://towardsdatascience.com/top-10-databases-to-use-in-2021-d7e6a85402ba (accessed Mar. 23, 2022).

[5.4] Altexsoft. “Comparing Database Management Systems: MySQL, PostgreSQL, MSSQL Server, MongoDB. Elasticsearch, and others.” Altexsoft.com. https://www.altexsoft.com/blog/business/comparing-database-management-systems-mysql-postgresql-mssql-server-mongodb-elasticsearch-and-others/ (accessed Mar. 23, 2022).

[5.5] A. Duggal. “SQLite vs MySQL: 5 Critical Differences.” Hevodata.com. https://hevodata.com/learn/sqlite-vs-mysql/ (accessed Mar. 23, 2022).

[5.6] A. Phajdar. “PostgreSQL vs MySQL: 8 Critical Differences.” Hevodata.com. https://hevodata.com/learn/postgresql-vs-mysql/ (accessed Mar. 23, 2022).

[5.7] S. Kakarla. “PostgreSQL vs MySQL.” Sumologic.com. https://www.sumologic.com/blog/postgresql-vs-mysql/ (accessed Mar. 23, 2022).

[5.8] Redis. “Redis Security.” Redis.io. https://redis.io/docs/manual/security/ (accessed Mar. 27, 2022).

[5.9] Redis. “Data Structures.” Redis.com. https://redis.com/redis-enterprise/data-structures/ (accessed Mar. 27, 2022).

[5.10] Redis. “Redis FAQ.” Redis.io. https://redis.io/docs/getting-started/faq/ (accessed Mar. 27, 2022).

[5.11] 2ndQuadrant. “PostgreSQL vs MySQL.” 2ndQuadrant.com. https://www.2ndquadrant.com/en/postgresql/postgresql-vs-mysql/   
(accessed Mar. 29, 2022).

[12] https://blog.bitsrc.io/what-is-an-orm-and-why-you-should-use-it-b2b6f75f5e2a

[13] https://sequelize.org/

[14] https://www.prisma.io/docs/reference/database-reference/supported-databases

[15] https://typeorm.io/

[16] https://github.com/prisma/prisma1/issues/2466

[17] https://openbase.com/js/sequelize/reviews

[18] https://orkhan.gitbook.io/typeorm/docs/caching

[19] https://typeorm.io/entity-manager-api

[20] https://www.eversql.com/django-vs-sqlalchemy-which-python-orm-is-better

[21] <https://pganalyze.com/blog/geodjango-postgis>

[22] https://www.enterprisedb.com/blog/postgresql-vs-mysql-360-degree-comparison-syntax-performance-scalability-and-features

[23] https://insights.dice.com/2015/03/19/why-i-choose-postgresql-over-mysqlmariadb/

## 6.x (Music Solution)

## 7.x (Music APIs)

[7.1] Amristar Solutions Pty Ltd, Virtual Jukebox – Project Brief. Perth, Western Australia. 2022.

[7.2] Spotify. Web Playback SDK Reference, Spotify. Accessed on: Mar. 30, 2022. Available: https://developer.spotify.com/documentation/web-playback-sdk/reference/

[7.3] Apple. MusicKit JS: Add an Apple Music player to your web app, Apple. Accessed on: Mar. 30, 2022. Available: https://developer.apple.com/documentation/musickitjs

[7.4] Deezer. Javascript SDK, Deezer. Accessed on: Mar. 30, 2022. Available: https://developers.deezer.com/sdk/javascript

[7.5] Deezer. API Wrappers, Deezer. Accessed on: Mar. 30, 2022. Available: https://developers.soundcloud.com/docs/api/sdks