SoundServe Design Document

Ryan Rouleau - July, 2017

Contents

Introduction

This document covers the features and requirements of my capstone project to end my training at the Laboratory for Atmospheric and Space Physics. It also includes the high-level code design, stack chosen, and respective justifications.

Overview

This application will consist of a server with a RESTful API that serves local music files and related information (albums, artists, playlists, etc.) to a client, a Python GUI to manage the music that is being served, and a web based front end that interacts with the API to deliver a high quality, fast, and modern music player.

How Requirements are Fulfilled

GUI

There will be a GUI written in PyQT that interacts with the server to manage the songs (adding/deleting/modifying directories) that are being served to the client. It's important to note that this component will only make simple API requests and will do no actual work. The aim of this approach is to reduce the code complexity while increasing the modularity of the project.

Non - GUI

All of the 'work' (serving music files, storing info about albums, artists, playlists, managing what music is being served, interacting with databases etc.) of this project will be performed by a server written in Node.js using the Express web app framework.

Database

A database will be created which will holds three tables:

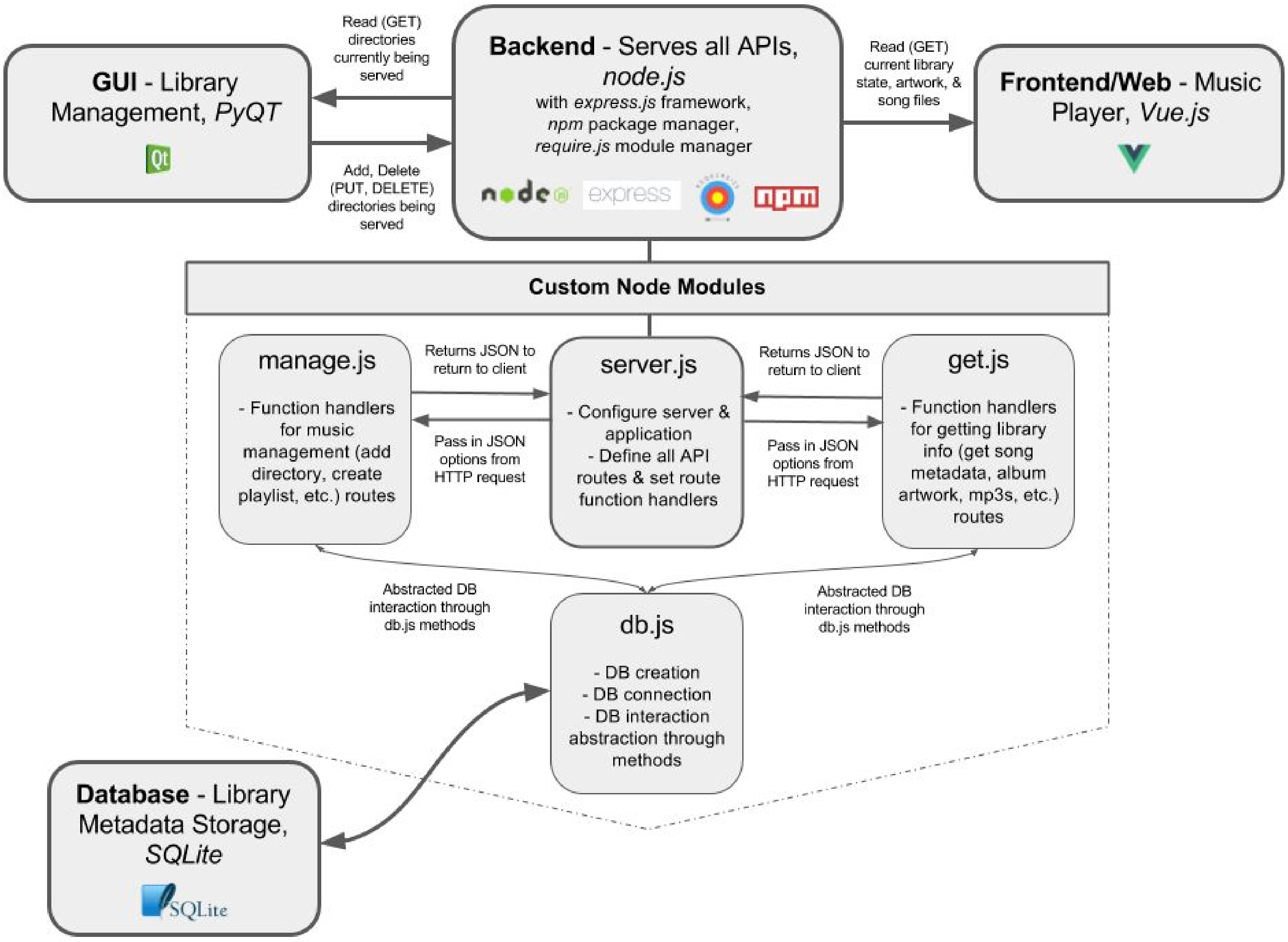
1. The first table will have an entry for every song that is being served and its metadata (unique id, artist, album, duration, year released, song file location, album artwork location).
2. The second table will hold the current directories being served to the client.
3. The third table will contain playlist information.

A more detailed database design will be developed at a later date...

Web

The actual music player interface will be a sleek, fast, and modern single page web app written using Vue.js.

High Level Component Interaction


GUI - Library Management, PyQT
Read (GET) directories currently being served
Add, Delete (PUT, DELETE) directories being served
Backend - Serves all APIs, node.js
with express.js framework, npm package manager, require.js module manager
Read (GET) current library state, artwork, & song files
Frontend/Web - Music Player, Vue.js
Custom Node Modules
manage.js
- Function handlers for music management (add directory, create playlist, etc.) routes
Returns JSON to return to client
Pass in JSON options from HTTP request
server.js
- Configure server & application
- Define all API routes & set route function handlers
Returns JSON to return to client
Pass in JSON options from HTTP request
get.js
- Function handlers for getting library info (get song metadata, album artwork, mp3s, etc.) routes
Abstracted DB interaction through db.js methods
Abstracted DB interaction through db.js methods
db.js
- DB creation
- DB connection
- DB interaction abstraction through methods
Database - Library Metadata Storage, SQLite

This graphic above is currently outdated. The current file structure is as follows:

app.js
package.json
lib/
  db.js
  verbose.js
api/
  apiRouter.js
  manage.js
  albums.js
  album.js
  artists.js
  artist.js
  song.js

Where app.js is equivalent to server.js in the graphic and contains the server configuration and high level route definitions ('/' and pushing '/api/*' to api/apiRouter.js to handle).

package.json is a json file used by the npm module manager which holds information about dependencies, the current version of the project, author, etc.

Each .js file in /api other than apiRouter.js corresponds to an API route (important: not a method -> e.g. manage.js is one route but holds three different methods for PUT/GET/DELETE requests) defined below in the **API Documentation** section.

db.js will create an object with the database connection, but will not have abstracted methods as defined in the graphic. Since each API route is handled in its own file and will use unique SQL queries, it makes more sense to have the SQL queries defined in those route handlers.

verbose.js is a small js file that receives command line arguments when starting the server and if --verbose or -v are present, it exports a boolean to be true, otherwise it exports false. This is then imported in almost all of the other .js files in the project to provide optional debug information to be printed to the console.

Design Justification

I choose to use Node and expose methods exclusively through a RESTful API because:

a) Node is asynchronous and nonblocking and thus the work of scanning directories -> modifying tables in the database -> hitting any public APIs to gather missing metadata and album artwork of songs -> writing album artwork to the disc, will be done concurrently and as a result will be much faster than a backend whose language does not have async programming at its core. Although it would be possible to make this work be done concurrently in for example, Python, it would be difficult and would require a slower, separate library.

b) **Any** arbitrary interface can be built by interacting with the back end exclusively through an API. This means a Python GUI is not necessary and the management of music could be handled by the web front end alone, or the entire front end could be written as a native desktop / mobile app, or the front end could be a world changing, revolutionary, AR experience! Wow! The possibilities are endless! Overall, no matter how the front end is built, the backend code is the same. This prevents the server from having to render any sort of user interface which also creates a distinct separation between the backend and frontend. This has benefits in speed, code organization, and code manageability. However, with all this good, there is some bad. I've identified two challenges that this design brings to the table:

   i) The client has to do more work since the interface will not be rendered ahead of time. To reduce the impact of this, I've chosen to use Vue.js for my frontend javascript framework. It implements a modern virtual DOM for fast manipulation of the page and has built in caching and optimizations. Vue will also make the front end code more manageable due to being reactive to changes in data as well as having the ability to create custom elements.

   ii) Displaying the progress of, for example, scanning through a directory when one is added will be difficult or nearly impossible to display on the front end. To solve this, a websocket connection can be made when scanning a new directory where messages are constantly sent to the client containing info on current progress. Another approach is to create an additional API route that can return, when hit, the current progress of a scan of a directory if one is active. Due to the complexity, the implementation of this will be a stretch goal.

   Overall, it is my opinion that the benefits of exclusively interacting with the backend through a RESTful API heavily outweigh these identified disadvantages.

Feature Set

- As a user I should be able to --- **add or delete a directory of music being served to the frontend player**
  --- *so I can manage my music library*

- As a user I should be able to --- **view what directories of music are currently being served**
  --- *so I can more easily manage my music library*

- As a user I should be able to --- **select a song and then be able to play, pause, go to the beginning, seek through, and see current progress through that song**
  --- *so I can listen to my music like any other music player I'm familiar with*

- As a user I should be able to --- **go the next or previous song in an album, playlist, or artist page with a click of a button**
  --- *so I can easily traverse that album, playlist, or artist page*

- As a user I should be able to --- **view and interact with my library by album, artist, or playlist**
  --- *so I can see and understand my music in a visual way*

- As a user I should be able to --- **view information (e.g. year released) about a song or album**
  --- *so I can understand the context of what I am listening to*

Stretch Goals - (more to be added during development)

- As a user I want to have --- **the option to stream spotify music along with my own library**
  --- *so I can enjoy a larger range of music*

- As a user I want to be able to --- **create / update / delete playlists**
  --- *so I can listen to music in the way I prefer*

- As a developer I want to --- **have a docker container of this project**
  --- *so I can quickly deploy this app to any system*

- As a user I hope the web front end --- **is beautiful**
  --- *so I enjoy using and navigating my music player*

- As a user I hope to be able to --- **see the live progress of directories of music being added**
  --- *so I know the app is working and not broken during the scanning of large directories*

API Documentation

See github readme: https://github.com/ryanrouleau/song-serve

Raw markdown: https://drive.google.com/file/d/0B2JmhxMUj9HfN3FiN0F6SmtKdEU/view?usp=sharing