Sound Memories

Technical Documentation

Table of Contents

Technical Summary

Sound memories is a browser application that was deployed as a desktop application using the JavaScript library “Electron”.

Stack:

- Front End: Vite + React
- Back End: Node.js
- Database: SQLite

Introduction

Sound Memories, as an application, is designed to serve as an organizational system that lives on top of an existing filesystem; it is not its own filesystem. This software was originally written as a browser application and later ported to an executable format using JavaScript libraries. As of this writing, the client of this project has indicated a desire to develop a prototype version of what is meant to be a large-scale, distributed application, with which she can gather feedback and judge market interest.

Overview

Sound Memories interfaces with a database using HTTP requests to return information about audio files. The only compatible filename extensions for user uploads (at the time of writing) are .mp3, .wav and .ogg.

Any data that’s read, updated, uploaded or deleted is displayed to the end-user on the “front end” of the application using the React.js framework, and the actual interfacing with the SQLite database is performed on the “back end” using Node.js. Additionally, an “invitation link” functionality exists, but is in an early stage as of this writing. User logins don’t currently exist, but can be implemented in the future; a table dedicated to storing login info exists in the database already.

Being a web-based application at its core, Sound Memories runs on two locally-hosted ports: “5173” for the front end, and “3000” for the back end. Code runs every time the program is launched to free these ports up; it is not wise to run other applications that are dependent on these ports while Sound Memories is currently running.

Setting Up

To install all of the dependencies for Sound Memories, install a package manager to start. A common one is “Node Package Manager” (or “npm”). From there, navigate to the project’s root directory, and run the install command for your package manager of choice (for “npm”, it would be “npm install” or “npm i”). The install command will likely take some time to complete. Once the install has finished, navigate to the “frontend” directory (“/frontend”) and run the install command there as well. After this, all dependencies should be installed and you are ready to begin working on the project.

The “Front End”

The client-facing (or “Front End”) portion of Sound Memories consists of the application’s graphical user interface (also referred to as “GUI”), and was written in Vite and React.js. The GUI has several tabs that allow the user to perform different actions. Due to the nature of Sound Memories’ origin as a browser application, no page reloading is needed to access the different tabs. Debug statements on the front end portion of the application are displayed in the browser console, which can be enabled using “Enable Developer Tools” under the “View” tab of the window (or the shortcut, which is CTRL + SHIFT + I on Windows). All data actions (such as creation of, retrieval of, updating of, or deletion of database records) are performed using JavaScript’s built-in “fetch” API. Last but not least, ESLint is used to ensure quality code is being written for the front end side of the application.

The “Back End”

The server-side (or “Back End”) portion of Sound Memories is where all of the application logic sits; handling of HTTP requests that are passed in from the front end, initialization of the database, interfacing with said database, and handling file uploads.

Routes

Explanation

Routes will follow this format:

- “route name” – HTTP VERB: Description of what action occurs
    - 000 (Status Code) – What behavior caused this code to return

Routing List

- “/audio” – GET: Will attempt to fetch all audio files stored in the database.
    - 200 (OK) – Audio files were successfully returned.
    - 404 (Not Found) – The database’s “audio_files” table is empty.
    - 500 (Internal Server Error) – The connection between the front end and back end was interrupted before retrieval could be completed.
- “/audio/:id”- GET: Will attempt to fetch an audio file stored in the database with a given ID number.
    - 200 (OK) – Audio file was successfully returned.
    - 404 (Not Found) – No audio file was found in the database with the provided ID number.
    - 500 (Internal Server Error) – The connection between the front end and back end was interrupted before retrieval could be completed.
- “audio/:id” – DELETE: Will attempt to delete an audio file stored in the database with a given ID number.
    - 200 (OK) – Audio file was successfully deleted.
    - 404 (Not Found) – No audio file was found in the database with the provided ID number.
    - 500 (Internal Server Error) – The connection between the front end and back end was interrupted before deletion could be completed.
- “audio/update” – PUT: Will attempt to update the metadata of an audio file stored in the database.
    - 200 (OK) – Audio file’s metadata was successfully updated.
    - 404 (Not Found) – Either the selected file does not exist in the database or no changes were made to its metadata.
    - 500 (Internal Server Error) – The connection between the front end and back end was interrupted before the update operation could be completed.
- “audio/filter/:filter” – GET: Will attempt to return a list of audio files in the database that match a selected tag.
    - 200 (OK) – List was successfully returned.
    - 404 (Not Found) – No audio files found with the selected tag.
    - 500 (Internal Server Error) – The connection between the front end and back end was interrupted before retrieval could be completed.

- “/tags” – GET: Will attempt to return a list of all tags currently stored in the database.
    - 200 (OK) – List of tags was successfully returned.
    - 500 (Internal Server Error) – The connection between the front end and back end was interrupted before retrieval could be completed.

- “audio/:id/tags” – GET: Will attempt to return a list of tags that are associated with a given audio file in the database.
    - 200 (OK) – List was successfully returned.
    - 500 (Internal Server Error) – The connection between the front end and back end was interrupted before retrieval could be completed.
- “/audio/assign-tag” – POST: Will attempt to add a tag in the database to the currently-selected audio file.
    - 200 (OK) – Tag was successfully added.
    - 500 (Internal Server Error) – The connection between the front end and back end was interrupted before assignment operation could be completed.
- “audio/tag/:tagName” – GET: Does not appear to be used anywhere in the Sound Memories application. Seems to serve a similar purpose to “audio/filter/:filter” above, but with less robust error handling and no success condition. Seems redundant, may be safe to remove in a future update.
    - 500 (Internal Server Error) – This is the only status code returned by this route.
- “audio/update/tags” – PUT: Will attempt to update the tags associated with an audio file in the database.
    - 200 (OK) – Tags were successfully updated.
    - 400 (Bad Request) – The audio file’s ID number or tag’s ID number was not included in the request.
    - 404 (Not Found) – The audio file was not found in the database.
    - 500 (Internal Server Error) – The connection between the front end and back end was interrupted before update operation could be completed.
- “audio/remove-tag” – POST: Will attempt to remove a tag with a given ID number from an audio file with a given ID number.
    - 200 (OK) – Tag was successfully removed.
    - 500 (Internal Server Error) – The connection between the front end and back end was interrupted before retrieval could be completed.

- “/upload” – POST: Will attempt to upload a user-provided audio file into the database. This route makes a call to the “uploadAudio” function from the “uploadController.js” file instead of the function from “controller.js”; more information can be found in the controllers section.
    - 201 (Created) – File was uploaded successfully and a new record in the database has been created.
    - 500 (Internal Server Error) – The connection between the front end and back end was interrupted before file upload could be completed.
- “/audio/upload” – POST: Will attempt to read in metadata about the uploaded file and store the metadata in the database. This route makes a call to the

"uploadAudio" function from the "controller.js" file instead of the function from "uploadController.js"; more information can be found in the controllers section.

- 201 (Created) – Metadata was successfully entered into the database.
- 500 (Internal Server Error) – The connection between the front end and back end was interrupted before insertion operation could be completed.

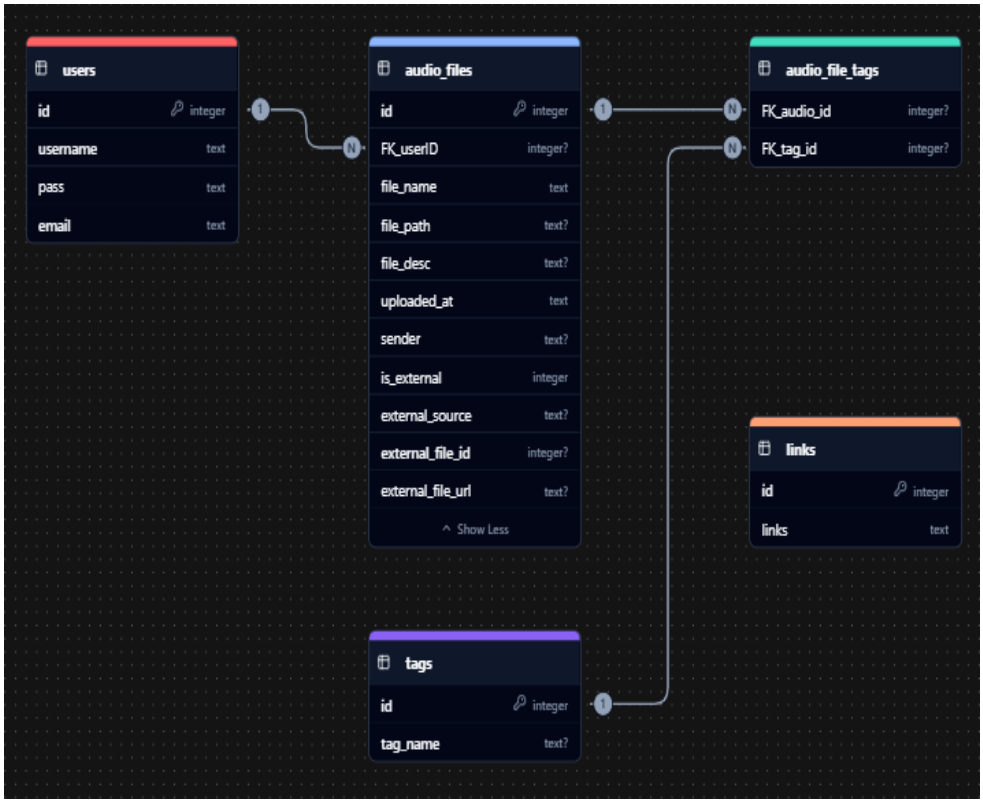
The Controllers

There's two different controller files: "controller.js" and "uploadController.js". All of the functions that interface with the database are contained in "controller.js". The file "uploadController.js" is only used to upload files themselves into Sound Memories as an application, and "uploadController.js" relies on the "Multer" package to handle file uploads. Currently, the only supported filetypes are .mp3, .wav and .ogg. Having two separate functions to handle the uploading of a file may be something worth taking a second look at in the future.

Due to the usage of the "better-sqlite3" package in Sound Memories, all functions in "controller.js" are synchronous; Promises and the "async" and "await" keywords are not supported unless "better-sqlite3" is replaced with a different SQLite package.

The Database

Sound Memories reads and writes files with a SQLite database as its backbone. The database that Sound Memories reads from is stored on the user's machine as a file in the root directory called "audio.sqlite". To the right is an image of the database schema (generated using chartdb):



Any column with question marks at the end of its datatype definition (such as "file_path" in the "audio_files" table) supports null values.

Currently, a function exists in the file “seed.js” to populate the database with dummy data if no audio files are present. This is to prevent unintended behavior, but can be removed in the future if necessary.

Complete List of Files and Functions (By Directory)

Root Directory

- “.gitignore” – Contains a list of files, folders and filename extensions for the Git Version Control System to ignore when listing files that have been added, changed, or deleted.
- “audio.sqlite” – The database that powers Sound Memories as an application. Will be automatically created the first time the program is run if it doesn’t already exist.
- “forge.config.js” – A configuration file for “Electron Forge”, the library used to deploy Sound Memories as an executable application.
- “log.txt” – SQL statements that are run on the database file and debug information from “main.js” are output here.
- “main.js” – Handles all of the desktop-specific logic, creating the application window.
- “package-lock.json” and “package.json” – A list of JavaScript packages that are needed for the back end of Sound Memories to run.
- “README.md” – A markup file that contains a brief introduction to the program as well as instructions on how to run it. The information may be slightly outdated due to the majority of it being written for an early version of Sound Memories.
- “server.js” – The main entry point to Sound Memories as an application. It enables the JavaScript package “Express”, establishes a test API endpoint, creates a router for the front end, and starts up the back end server.

Any other files in this directory can be safely deleted or ignored. All functions within files in this directory should be fairly self-explanatory or else are used for configuring builds rather than the functionality of the application itself.

“controllers” Directory

“controller.js”

- Description: The main controller file. Handles all interactions with the database, and the functions it exports are called by the router file.

- Functions:
  - getAll() – Calls the corresponding function from “db.js” (in the “db” directory), and returns an appropriate status code based on the behavior of the database.
  - getFiltered() – Receives the filter from the request body, then calls the corresponding function from “db.js” (in the “db” directory) using that filter as a parameter. Returns an appropriate status code based on the behavior of the database.
  - getById() – Receives the ID from the request body, then calls the corresponding function from “db.js” (in the “db” directory) using that ID as a parameter. Returns an appropriate status code based on the behavior of the database.
  - uploadAudio() – Receives an HTTP request from the front end and splits it into its individual properties, then passes all of those properties in as parameters to the corresponding function from “db.js” (in the “db” directory). Returns an appropriate status code based on the behavior of the database.
  - deleteAudio() – Receives the ID from the request body, then checks to see if the ID exists in the database by calling the database’s “getById” function. If the file itself exists on the system, the file will be deleted. Calls the corresponding “deleteAudio()” function from “db.js” (in the “db” directory) if the ID exists in the database, using that ID as a parameter. Returns an appropriate status code based on the behavior of the database.
  - getTags() – Calls the corresponding function from “db.js” (in the “db” directory), and returns an appropriate status code based on the behavior of the database.
  - getTagsByAudioId() – Receives the ID from the request body, then calls the corresponding function from “db.js” (in the “db” directory) using that ID as a parameter. Returns an appropriate status code based on the behavior of the database.
  - assignTag() – Receives the ID of both the tag and the audio file that the tag should be assigned to from the request body. Calls the corresponding function from “db.js” (in the “db” directory), and returns an appropriate status code based on the behavior of the database.
  - removeTag() – Receives the ID of both the tag and the audio file that the tag should be removed from as parameters from the request body. Calls the corresponding function from “db.js” (in the “db” directory), and returns an appropriate status code based on the behavior of the database.
  - getAudioFilesByTagName() – Receives the name of the tag as a parameter from the request body. Calls the corresponding function from “db.js” (in the “db” directory), but only returns status code 500. This function appears to

be redundant, and may be worth investigating to see if it can be replaced with “getById()”.
    - updateAudio() – Receives an HTTP request from the front end and splits it into its individual properties, then passes all of those properties in as parameters to the corresponding function from “db.js” (in the “db” directory). Returns an appropriate status code based on the behavior of the database.
    - updateTags() – Receives the ID of both the tag and the audio file that the tag should be assigned to from the request body. Calls the corresponding function from “db.js” (in the “db” directory), and returns an appropriate status code based on the behavior of the database.

“uploadController.js”

- Description: This is the other controller file. It makes use of functions from the “multer” package to upload the user’s requested file into Sound Memories’ filesystem, while the main controller file handles interactions with the database. The only purpose this file appears to serve is to handle the uploading of a user-submitted file.
- Functions:
    - storage() – Makes a call to the “diskStorage” function from the “multer” package and passes in the request body, a file, and a callback function (which normalizes the upload path for the file) as the destination property. From there, a name for the file is generated by appending a call to “Date.now()” to the original filename, and this “new” filename serves as the filename property that gets passed to multer.
    - fileFilter() – Receives a request, file, and callback function as its parameters. Checks that the MIME type of the file is of an approved list of types (“audio/mpeg”, “audio/wav”, or “audio/ogg”), and if it is, then the function returns true. If the MIME type isn’t on the whitelist, then an error is returned instead.
    - upload() – Makes a call to the “multer” function, passing in a call to “storage()” and a call to “fileFilter()” as parameters.
    - uploadAudio() – Receives a request and response as its parameters. If there isn’t a file present in the HTTP request, an error is returned. Otherwise, a JSON response will be returned that includes a success message, the filename, and the URL of the newly-uploaded file.

It may be worth consolidating these functions into the file “controller.js” in a future update, as the existence of two controller files seems a bit redundant.

“db” Directory

“db.js”

- Description: This file contains functions that will create the database file and populate it with dummy data (if the file doesn’t already exist), and also handles all querying of the database file. Functions from this file are called by the file “controller.js” in the “controllers” directory.
- Functions:
    - createContributionRequest() – Accepts a link as its parameter, then prepares an INSERT query and SELECT query. The INSERT query runs with the provided “link” parameter, and if successful, then returns the link as a result of the SELECT query. If the insertion wasn’t successful, nothing is returned. This appears to be the implemented version of the commented-out function from the top of “controller.js”.
    - getAll() – Prepares a SELECT query to retrieve all audio files from the database. Returns everything that is retrieved from the database as a result of the SELECT query as an array of objects.
    - getFiltered() – Receives a “tagName” parameter to use as an argument in a WHERE clause for a SELECT statement to the database. Returns all database records that match the WHERE clause.
    - getById() – Receives the ID of an audio file to use as an argument in a WHERE clause for a SELECT statement to the database. Returns all database records that match the WHERE clause. Returns null if no matching records were found.
    - uploadAudio() – Receives properties of a file from the request body as parameters, then prepares an INSERT statement using those properties. Returns the ID of the last row inserted into.
    - deleteAudio() – Receives the ID of an audio file to use as an argument in a WHERE clause for a DELETE statement to the database. Returns the number of rows in the database that were affected by the DELETE statement.
    - getTags() – Prepares a SELECT query to retrieve all audio file tags from the database. Returns everything that is retrieved from the database as a result of the SELECT query as an array of objects.
    - getTagsByAudioId() – Receives the ID of an audio file to use as an argument in a WHERE clause for a SELECT query to the database. Returns the tags associated with an audio file with the provided ID.
    - addTag() – Receives the name of a tag to insert into the “tags” table of the database via INSERT query. Returns the ID of the last row inserted in the database.

    - assignTag() – Receives the ID of a tag and the ID of an audio file, and attempts to run an INSERT query in the database to associate the two. No values appear to be returned from this function.
    - removeTag() – Receives the ID of a tag and the ID of an audio file, and attempts to run a DELETE query in the database to remove any association between the two. No values appear to be returned from this function.
    - getAudioFilesByTagName() – Receives the name of a tag and attempts to run a SELECT query in the database to retrieve any information about audio files in the database that have the tag associated with them. Returns all of the rows that were provided as a result of the SELECT query.
    - updateAudio() – Receives numerous properties of an audio file as parameters, and dynamically builds an UPDATE query based on the presence of a given property, then attempts to run the UPDATE query. If no rows were affected, this function will return null. If the UPDATE query fails to run, this function will return an error. If the UPDATE query successfully runs and affects rows, the function will then run a SELECT query to return the updated row.
    - updateTags() – Receives the ID of a tag and the ID of an audio file, and attempts to run an INSERT query in the database to associate the two. If successful, the number of rows that were altered by the INSERT query will be returned. If not successful, an error will be returned instead.

The function “getAudioFilesByTagName()” may be worth investigating in a future update as it seems a bit redundant. Its corresponding function in “controller.js” isn’t fully functional; it’s only able to return a 500 status code. The route associated with the “controller.js” function, “audio/tag/:tagName” in “controller.js” also does not appear to be implemented anywhere in Sound Memories.

“schema.js”

- Description: This file contains the database schema. It has one function that gets called near the top of “db.js”.
- Function:
    - initializeSchema() – This function receives a database file as a parameter, and runs a series of CREATE TABLE IF NOT EXISTS queries on the database file. It starts by running a pragma to turn on foreign keys, then creates the tables one by one. See the “The Database” section of this document for more information about the schema.

This file is complimentary to “db.js” as the database cannot function without a schema. However, besides initializing the database, “schema.js” doesn’t do much; it could be consolidated into “db.js” in a future update if need be.

“seed.js”

- Description: This file contains a function that will populate the database with dummy data if there aren’t any audio files currently in the database. This is a development-oriented file and should not be included in deployable iterations of Sound Memories.
- Function:
    - “plantSeeds()” – This function receives a database file as a parameter, and runs a series of INSERT OR IGNORE queries on the database file to insert dummy data into the database’s tables. This insertion operation will only occur if there are no audio files currently present in the database, however.

Again, this file is only meant for development purposes. Public releases of Sound Memories as an executable program likely should not include this file to save space and prevent end-user confusion.

“files” Directory

The files in this directory aren’t of importance, since this is where the user’s audio files will be uploaded to. No code exists here.

“frontend” Directory

Root directory

- “.gitignore” – Contains a list of files, folders and filename extensions for the Git Version Control System to ignore when listing files that have been added, changed, or deleted.
- “eslint.config.js” – A configuration file for ESLint, a linting package that was used in the development of Sound Memories. Can be removed if future developers choose make use of a different linting package.
- “index.html” – The base webpage upon which the entire front end of Sound Memories is built. The only thing of note here is that “index.html” calls “main.jsx” in a script tag, and “main.jsx” can be swapped for any other React file (with a “.jsx” filename extension) or JavaScript file (with a “.js” filename extension).
- “package-lock.json” and “package.json” – A list of JavaScript packages that are needed for the front end of Sound Memories to run.
- “README.md” – A markup file that contains some boilerplate information about how to get React.js to work with Vite.

- “vite.config.js” – A configuration file that instructs the package “Vite.js” to use “React.js” as a plugin and to host the application explicitly on port 5173 of the user’s local machine.

“node_modules” Directory

If this directory doesn’t exist, it’ll be created the first time you use the Node Package Manager to install the packages Sound Memories needs as detailed in the Setting Up section. This directory doesn’t really need any attention, it just contains the packages that Sound Memories requires in order to function properly. It’s excluded from the Git Version Control System by design.

“public” Directory

The only file in here is “vite.svg”, which is a logo file. This directory can be ignored without any consequence.

“src” Directory

- “App.css” – The stylesheet for “App.jsx”. Also used in “Recordings.jsx”.
- “App.jsx” – This React file provides the overall structure of the webpage, but is not the entry point for the webpage itself. Calls “Recordings.jsx”, “ManageAudio.jsx”, “RecordAudioRequest.jsx”, “RequestAudioForm.jsx” and “Sidebar.jsx”.
- “CopyLink.css” – The stylesheet for “CopyLink.css”. Not used in any other file.
- “CopyLink.jsx” – The modal pop-up that presents the user with a request link to copy to their clipboard.
- “main.jsx” – The React file that renders the webpage itself. Calls “App.jsx”.
- “ManageAudio.jsx” – This file provides structure for the “Manage Audio” tab of the Sound Memories application. Calls “UploadForm.jsx” and “UpdateRecordings.jsx”.
- “RecordAudioRequest.css” – The stylesheet for “RecordAudioRequest.jsx”. Not used in any other file.
- “RecordAudioRequest.jsx” – The page that allows senders to record audio messages to be sent to the user who requested them. Calls “RecordView.jsx”.
- “Recordings.jsx” – The “view recordings” subsection of the tabs in Sound Memories. Users can view recordings and select a tag to filter by, and audio files that have that tag will be displayed.

- “RecordView.css” – The stylesheet for “RecordView.jsx”. Not used in any other file.
- “RecordView.jsx” – The section of the “record audio request” page that actually allows the sender to record their audio message. Makes POST requests to upload the recordings.
- “RequestAudioForm.css” – The stylesheet for “RequestAudioForm.jsx”. Not used in any other file.
- “RequestAudioForm.jsx” – The modal window that pops up when the user selects the “Request Audio Form” tab.
- “Sidebar.css” – The stylesheet for “Sidebar.jsx”. Not used in any other file.
- “Sidebar.jsx” – The sidebar that allows the user to navigate between the different tabs in Sound Memories.
- “UpdateRecordings.css” – The stylesheet for “UpdateRecordings.jsx”. Not used in any other file.
- “UpdateRecordings.jsx” – The “Manage Audio” tab of Sound Memories. Allows the user to view their stored audio messages (as well as metadata associated with the messages). Also allows the user to update information about the recordings or delete them entirely.
- “UploadForm.css” – The stylesheet for “UploadForm.css”. Not used in any other file.
- “UploadForm.jsx” – The modal window that pops up when the user clicks on the “Upload Audio” button in the “Manage Audio” tab of Sound Memories. Prompts the user to select a file, enter the name of the person who sent it (and an optional description), and select a tag to associate the message with (which is also optional).

“node_modules” Directory

Yes, there can be two instances of this directory: one in the root directory if “npm install” is run from the root directory, and one in the “frontend” directory if “npm install” is run from the “frontend” directory. If this directory doesn’t exist, it’ll be created the first time you use the Node Package Manager to install the packages Sound Memories needs as detailed in the Setting Up section. This directory doesn’t really need any attention, it just contains the packages that Sound Memories requires in order to function properly. It’s excluded from the Git Version Control System by design.

“routers” Directory

The functions in this directory’s only file, “router.js”, are all designed to handle HTTP requests. More information about the router file can be found in the “Routes” section of this document.

“scripts” Directory

- “docker-compose.yml” – This is a leftover file from a previous version of Sound Memories. It serves as a set of instructions to the software “Docker” to install certain software so that Sound Memories could function as intended; it’s left over in case Docker-based development plans arise again in the future.
- “pass_hash.js” – This file contains an early attempt to compute hashes for user passwords. It makes use of the “bcrypt” package to perform the hashing, but no other functionality has been implemented. This file will need to be fleshed out more once user logins get implemented.
- “schema.sql” – An old version of the database schema from an earlier version of Sound Memories. Kept around in case another dialect of SQL is needed later in development.
- “seed.sql” – An old version of the seed data from an earlier version of Sound Memories. Kept around in case another dialect of SQL is needed later in development.

This directory is kept around for legacy purposes and can be safely ignored or deleted without repercussion.

Project Resources

- GitHub Repository: This is an online repository that has the complete development history of Sound Memories up to its basic prototype stage.
- Trello Board: This is an organizational tool that the original development team used to keep track of feature ideas and progression towards specific 2-week goals (also called “Sprint Goals”).
- User Manual: This document is meant for consumer use and should be included alongside this technical document. It serves as a basic instruction manual to demonstrate to an end-user how to properly utilize Sound Memories as an application.
- Technical Document: You’re currently reading it.

Development Roadmap

The client's vision for Sound Memories was to develop a prototype (which should be available at the GitHub repository, found under the "Project Resources" section of this document) and then use the prototype to gauge market interest in Sound Memories as a distributed application.

The client would seek input for the prototype from mental health professionals for further refinement before then gauging market interest and, if the interest is present, continue development to further flesh out Sound Memories as an application, adding in online functionality and more quality-of-life features, then attempting to go public with Sound Memories once the application is in a satisfactory state.

The client's goal is to have Sound Memories be available as a mobile application first and foremost, with a desktop application being available as a companion to the mobile app.

Future Changes to Implement

- Functionality with online storage services (such as Google Drive) was requested by client but not completed by the prototype development team.
- Implementation of a greeting message for the Sound Memories application was requested by client; something like "What do you need?" with the idea being that the user would then select from a series of categories such as "Support", "Love", "Encouragement", etc.
- Support for licensed mental health professionals to create accounts for their clients was requested by the client but not completed by the prototype development team.
- Support for custom visual themes was requested by the client but not completed by the prototype development team.
- The client has a list of template phrases that could be available on the "Request Audio Form" which are not currently implemented.
- A red "recording" dot could be added to the "Record Audio Request" page to indicate to the sender when they are being recorded, and then disappear when the recording is finished.
- The ability to add custom tags was spoken of shortly before project handoff to the client, but was not completed.

- Adding on to the “custom tag” functionality, the “Home” tab of Sound Memories currently has a hard-coded tag list that it displays; updating this to play nice with custom tags was a feature that was discussed by the prototype development team.
- Allowing end-users to sort their messages by sender instead of by audio tag was a feature that was discussed by the prototype development team.
- Adding a reset button to completely wipe Sound Memories’ database seems like a wise idea.
- Optimization of routing; replacing redundant endpoints with more robust ones may be worth looking into.
- There are a few redundant functions that could be removed or consolidated into other functions; “getAudioFileByTagName()” in both “controller.js” and “db.js” is an example.
- Adding functionality for users and user logins is necessary before going public.
- There exists a password-hashing function in the file “passHash.js” in the “src” directory; it’s functional but underdeveloped, and should likely be further fleshed out (along with hash-comparison functions) before user login functionality is considered complete.
- Dummy data that’s automatically inserted into the database should probably be either removed or replaced; currently there’s some copyrighted material in there that would open us up to infringement lawsuits.

Development Credits

Sound Memories initial prototype was developed by:

- William Castillo
- Jared Eller
- David Jarvis
- Huy Nguyen