record-log
Programmer's Documentation

Spring Term 2024 - CS 422 Anthony Hornof

Ethan Hyde (EH)

John O'Donnell (JO)

Table of Contents

1. Introduction

This document was created to provide a comprehensive guide to understanding the Record-log application. It is a robust, full stack tool with the intention of being used by both music professionals and hobbyists to categorize different songs and visualize the thematic links between them. By using the MERN stack (MongoDB, Express, React, and Node), we were able to create a seamless user experience with an intuitive design. It is structured to support the necessary CRUD operations and visualization through a graphical interface. This document explains exactly how we did that.

2. record-log Source Files

2.1. backend/models/record.model.js

This file is responsible for creating a Mongoose schema definition for a MongoDB collection, to be used in a Node.js application.

2.1.1. - Imports and Setup

The code first imports the ‘mongoose’ library, which is a MongoDB tool designed to work with an asynchronous environment. Then the code creates a shortcut to the ‘Schema’ constructor to be easily referenced.

The schema is created with the intention of using it in our database. Each field in the schema is assigned either ‘true’ or ‘false' which equate to the field being required or optional. Lastly, the code creates a Mongoose model named ‘Entry’ using the defined schema. It exports this model to other parts that are able to interact with the ‘entries’ collection and reference the MongoDB database.

2.2. backend/routes/entries.js

This code created an Express.js router module that is responsible for handling API calls that perform CRUD operations on our MongoDB collection using Mongoose and the Express.js framework. It defines all the necessary routes to interact with our ‘Entry’ model, created above in *Section 2.1.*

2.2.1. - Imports and Setup

First the Express router and the ‘Entry’ model are imported using the two ‘require’ statements at the top of the file.

2.2.2 - GET Route - All Entries

We then define a GET route at the root path ‘/’ to fetch all the entries from the database. This is done by using the ‘Entry.find()’ function to retrieve all the entries, and sends them back in the form of a JSON. If something goes wrong, an error message will be sent instead.

2.2.3 - POST Route

This function defines a POST route at ‘/add’ to create a new entry. It extracts the data from the request body while creating a new ‘Entry’ instance. The new entry is then saved to the database, followed by a success message if all goes correctly. Otherwise, an error message is sent.

2.2.4 - GET Route - Fetch Entry by ID

This route is different from the route defined in *Section 2.2.2.* We defined another get route at ‘/:id’ to fetch a specific entry by its ID. This uses ‘Entry.findByID(req.params.id)’ to retrieve the entry and return it in the form of a JSON.

2.2.5 - DELETE Route - Remove Entry by ID

Defining a delete route at ‘/:id’ allows for the user to remove any entry they created from the database. It uses ‘Entry.findByIDAndDelete(req.params.id) to delete the entry. It then is followed by sending a success or error message if something went wrong.

2.2.6 - PUT Route - Update Entry By ID

This route allows for the user to update their entries. It defines a PUT route at '/update/:id' to update the fields in an entry based on their ID. It uses 'Entry.findByID(req.params.id)' to find the entry. After finding it, it updates the fields in the entry with the fields in the request body. Then it saves the entry to the database and sends either a success or error message.

2.2.7. PUT Route - Delete List Name from Entry

For our list functionality, we created a final PUT route at '/delete-list/:entryID' to delete a specific list name from an entry. It extracts the list name from the request body, finds the entry by its ID, then removes the specified list name from the 'lists' field. It then saves the updated entry to the database, returning a success or error message.

2.3. backend/server.js

This file is responsible for creating a Node.js server using the Express framework. It sets up the server to handle all of the API calls for the CRUD operations on the MongoDB database. It leverages Mongoose for database interaction and CORS for handling cross origin requests, and transferring data between the browser and server.

2.3.1. Imports and Setup

We use express to handle the creation of the server and define routes for the API CRUD operations. CORS is then imported to handle the Cross-Origin Resource Sharing. Mongoose is to interact with MongoDB, and 'dotenv' is to load the environment variables from the '.env' file, which connects to our MongoDB Atlas cluster.

2.3.2 Middleware Setup

The command 'app.use(cors());' enables CORS to be used on all the routes, which allows the server to accept requests from any origin. The command 'app.use(espress.json())' parses the incoming requests with JSON, which makes the JSON data available in 'req.body.'

2.3.3. Router Setup

The next portion of code imports the router module that we defined in './routes/entries' by using a required statement. Then this router is mounted on the '/entries' path, so that any requests to '/entries' are handled through this router

2.3.4. MongoDB Connection

The 'mongoose.connect(process.env.MONGO_URI)' command connects to the MongoDB database using the Atlas cluster URI variable defined in the '.env' file. Then a success message or an error message is outputted to the console. Lastly, the server is started on the specified port with the ‘app.listen’ command. A message is then outputted to the console indicating what port the server is running on.

2.4. record-log/listView.js

2.4.1 ListView

This file displays a text box that allows the user to enter the name for a newly created list. The file creates a hashmap that stores the multiple user lists. The hashmap contains a string list_id as the key and an array entries[] as the value. The entries are fetched from the database and are used to populate the

2.4.2 fetchEntries()

This function helps to populate the lists hashmap using the entries from the database. The function loops through the entries retrieved from the database. For each entry, it loops through the lists array for that entry. If the list is already within the hashmap and has entries, the entry is pushed to the entries array for that list within the hashmap. If the list is not within the hashmap, a new key, value pair is added with the key being that list_id and the value being an array containing only that entry. When this is done the lists hashmap is set to the temporary hashmap.

2.4.3 handleListSave()

This function handles the case where a new list is created using the textbox and save button. It adds a new key, value pair to the lists hashmap. The key is the name retrieved from the textbox and the value is an empty array where the entry_id will be stored. The function then resets the name retrieved from the textbox so it can be filled with the next created list.

2.4.4 handleListClick(list_id)

This function handles the event that a user clicks on a list to display its details. The selectedList is set to the key,value pair of list_id, entries[]. This allows the rendering of the new selectedList. The selectedEntry is also reset because we don’t want to display those entry details when trying to open a new list.

2.4.5 handleCloseStylizedBox()

This function is used to close the boxes that are rendered to contain list and entry information. It simply resets the selectedList and selectedEntry to null.

2.5. backend/listMenu.js

2.5.1 DisplayEntry()

The DisplayEntry() function takes inputs for the specified entry and also takes a function that handles closing the display. The function retrieves the required entry information and then displays it in a new display box which is stylized by css. There is also a close button that calls the given close functionality.

2.5.2 ListWindow

When the user clicks on a desired list the file saves that list as the selectedList. The selectedList is displayed as a stylized css box. Within the selectedList display the file displays the entries associated with that list. When the user clicks an entry, the file saves the selectedEntry and calls DisplayEntry() which displays all the corresponding entry information in another stylized css box.

2.5.3 togglePopup()

This function changes the showPopup value to toggle the rendering of a dropdown list of entries to be added to a list within the lists interface.

2.5.4 handleEntryClick(entry)

This function changes the selectedEntry within listView.js to the given entry

2.5.5 closeEntryDetails()

This function resets the selectedEntry within listView.js to null. The result is the entry details are no longer displayed.

2.5.6 handleSelectedEntry(entry)

This function handles adding entries to lists from within the list interface. If the chosen entry is not a member of any lists, the lists array is set to contain the selectedList list_id. If the entry is already a member of some list, the selectedList list_id is pushed to the lists array. The function then updates the database with the new lists array for that entry.

2.5.7 handleDeleteEntry(id)

This function handles the case where a user deletes an entry from a list. The function removes the entry_id from the entryList and resets the selectedEntry if that entry is being deleted.

2.5.8 handleDeleteList()

This function allows users to delete lists from within the list interface. It loops through the entries within that list and calls the handleDeleteEntry on them which removes them from the list.

2.5. record-log/src/components/pallette.js

This file only has one purpose, which is to create a theme by using the Material-UI function ‘createTheme.’ It allows for us to create a consistent theme that can be accessed from across our application, and color elements with the same hex code colors.

2.6. record-log/src/components/stylizedBox.js

The purpose of this file is to create a stylized box component containing a button that is used in our navigation bar. It uses the Material-UI (MUI) library to create a theme and assist in creating the box itself.

2.6.1. Component Definition

The component ‘const StylizedBox’ defines a functional component that accepts props for ‘text,’ ‘href,’ ‘width,’ ‘height,’ and ‘fontSize.’ The default values are what get used in the navigation header, but these are able to be changed so that we can use this component across our application with different dimensions and text characteristics.

2.6.2. JSX Structure

This file returns a ‘Box’ component that has several different styling properties that define its appearance and behavior. Some of which are the color, background color, and padding.

2.7. record-log/src/Entry.jsx

This file is responsible for defining the user entry component that is accessible on the home page of our application. It handles the functionality for creating new track entries, updating existing entries, and populating all of the entry fields with the user input.

2.7.1. Component Definition and State Variables

The variable ‘const Entry’ accepts 3 props. ‘onEntryCreated’ is a callback function that refreshes the entries list after an entry is created or updated. ‘entryID’ is the ID of the entry to be edited. ‘setEntryID’ is a function to reset the ‘entryID’ after an operation. Next, the long list of state variables are defined to manage the entry fields and related data during operations.

2.7.2. Fetching Entry by ID

The first ‘useEffect’ function calls an ‘axios.get()’ request to fetch the entry data when entryID changes, and populates the state variables with the entry data that was fetched. The next ‘useEffect’ function fetches all of the entries with an ‘axios.get()’ command. It does this when the component mounts and stores them in ‘allEntries.’

2.7.3. Fetching All Entries

This ‘useEffect’ function fetches all of the entries when the component mounts, then stores them in the ‘allEntries’ variable.

2.7.4. Fetching Related Entries

This function will use an ‘axios.get()’ call to fetch all of the related entries by ID, and set them by calling ‘setRelatedEntries.’ This function helps us to create thematic links between each entry and display these links in the graph view.

2.7.5. Creating or Updating an Entry

The function ‘handleCreateEntry’ will either create a new entry or update an existing entry. It will construct a ‘newEntry’ object from the state variables. If ‘entryID’ is provided, it would update the existing entry, otherwise it creates a new entry. After the operation, it clears the input fields and calls the ‘onEntryCreated’ function to refresh the entries list

2.7.6, Clear Fields

The ‘clearFields’ function will clear all the input fields and reset the state variables.

2.7.7. Render Stars

The ‘renderStars’ function will render the star rating elements. It creates an array of 5 stars and maps each star to a span element. The ‘selected’ class is applied to all stars up to a current rating. Clicking the star updates the rating.

2.7.8. Toggle Popup

The togglePopup function toggles the visibility of the related entries popup. If the popup is not shown, it fetches all the entries then toggles the popup state.

2.7.9. Entry Selection

handleEntrySelection will handle the selection of the related entry. It adds the elected entry’s ID to the ‘related’ state and the entry object to the ‘relatedEntries’ state. Then it’ll toggle the popup.

2.7.10. Removing related entries

The function handleRemoveRelatedEntry will remove a related entry from the selection. It filters out the specified entry ID from the ‘related’ state and removes the corresponding entry object from the ‘relatedEntries’ state.

2.8. record-log/src/Home.js

This file is responsible for retrieving the entries created by the user and displaying them to the Homepage.

2.8.1. Component Definition

First, the file sets the ‘const Home’ variable. The ‘entries’ variabele stores the list of track entries, ‘expandedEntries’ keeps track of which entries are expanded, and ‘selectedEntryID’ stores the ID of the entry currently being edited.

2.8.2. Fetch Entries

The function ‘const fetchEntries’ fetches all the track entries from the backend throught a GET request, and updates the ‘entries’ state.

2.8.3. Toggle Entries

‘toggleEntry’ will toggle the expanded state of the entry. It updates the ‘expandedEntries’ state to show or hide the full details of the specified entry.

2.8.4. Edit Entry

‘handleEditEntry’ sets the ID of the entry to be edited It updates the ‘selectedEntryID’ state with the ID of the entry to be edited.

2.8.4. Delete Entry

The ‘handleDeleteEntry’ function helps to delete an entry from the database through a DELETE request removing the specified entry, and refreshing the list by calling ‘fetchEntries.’

2.8.5. Related Entry Name

The ‘getRelatedEntryName’ function will retrieve the names of related entries based on their IDs. It finds the related entries by their IDs and returns their names in a comma separated string.

2.9. /record-log/src/index.js

This file is responsible for essentially being the entry point of our React app. It creates a root element to mount the React application on, and renders the main ‘App’ component which is wrapped in ‘BrowserRouter’ to enable routing between components.

2.10. /record-log/src/Navbar/js

This file defines the implementation of the navigation bar component that gets rendered at the top of the Record-Log application. It allows for the user to navigate between the different pages of our application using the StylizedBox component created in *Section 2.6.* It also renders the logo image for our application in the top left corner.

2.11. /record-log/src/Popup.js

This file defines a React component called ‘Popup’ which serves as a search instance to find and select database entries. To add to a list. It gets sed in ‘ListMenu.js.’

2.11.1. Popup Definition

The ‘const Popup’ variable accepts three props. ‘onClose’ will handle closing the popup. ‘onSelect’ will handle selecting an entry, and ‘selectedEntries’ contains the entries that have been selected. Next, it defines 3 state variables. ‘allEntries’will store all the entries retrieved from the database, ‘filteredEntries’ wills store all the entries filtered based on the search term, and ‘searchTerm’ will store the user’s input in the search bar.

2.11.2. Data Fetch and Filter

The first ‘useEffect’ runs once when the component mounts and fetches all the entries from the database using a GET request. It updates the ‘allEntries’ and ‘filteredEntries’ variables with the data it fetched.

The second ‘useEffect’ runs each time ‘searchTerm,’ ‘allEntries,’ or ‘selectedEntries’ changes. It filters ‘allEntries’ based on the search term, and ignores the entries that are already selected, which then updates ‘filteredEntries’ with the results.

2.12. record-log/src/Graph.js

This file defines the implementation for the ‘GraphView’ component, which helps users visualize the thematic connection between entries in a graph format. This component gets used in the ‘App.js’ file and is routed to the ‘/graph-view’ path.

2.12.1. ForceGraph2D

ForceGraph2D is a robust and well tested library that helps facilitate the rendering of 2D force-directed graphs. We import it at the top of our file to be used throughout.

2.12.2. Display Node Component

The ‘DisplayNode’ component displays the details of the selected node in a styled ‘Box.’ It uses the MUI ‘useTheme’ hook to access the current theme for styling. This allows for the user to see the details of an entry when they click on its node in the graph.

2.11.3. GraphView Component

This component has 3 state variables. 'graphData' stores the nodes and edges for the entries in the graph, 'loading' is a state that tracks if the data is being fetched, 'selectedNode' tracks the currently selected node.

2.11.4. Fetching Data and Building Graph

First, 'useEffect' will fetch all the entries from our MongoDB Atlas cluster when the component mounts. Then, it will convert the fetched entries into a series of nodes and edges for the graph. Each entry is a node, and the related entries are connected by edges, the thematic links between entries.

2.11.5. Rendering the Component

First, a loading message will be displayed while the data is being fetched. By using 'ForceGraph2D,' we then render the graph with nodes and edges. This function also handles setting the selected node when a node is clicked, displaying its contents using the 'DisplayNode' component.

2.12. Splash Screen

This component is responsible for rendering the splash screen upon startup and when the user clicks the 'Home' tab in the navigation bar. By using a 'useEffect' hook, the splash screen is able to be displayed for only 3 seconds, then disappear. If the user somehow clicks out before the splash screen has finished, cleanup code allows for the animation to terminate before the 3 second timer.