

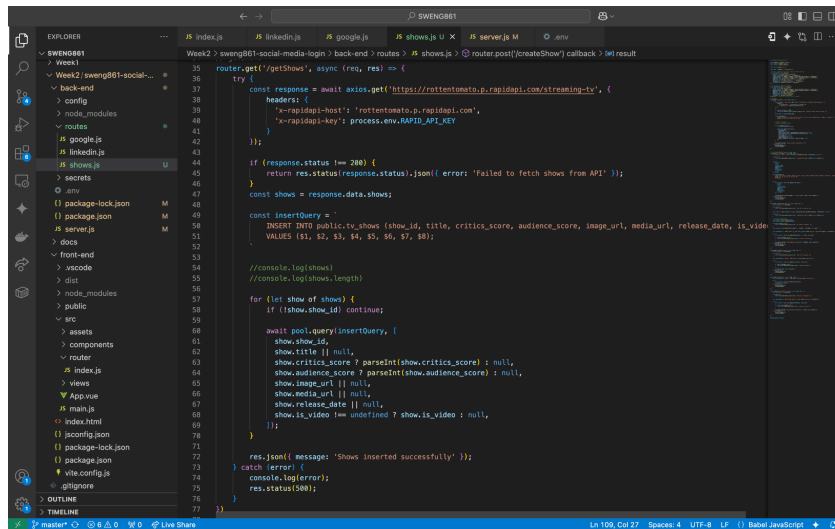
Introduction

This document outlines the applications API endpoints, what they do, what parameters they take, response variables, and the implementation process.

Implementation

The first part of implementing CRUD operations is always retrieving data to work with. For my project, I decided to use the free API, available via <https://rapidapi.com/>, called Rotten Tomato - tv shows, which has data on various tv shows from the popular website called Rotten Tomatoes.

To retrieve the data, I retrieved an API Access Key from the website, and made a GET request to the resource. In the case that something went wrong with this request, I immediately check if the status code is anything other than 200, and return an error message if this is the case. After that validation check, it was time to parse through the data and insert it into a database I created called tv_shows. Using a simple insert query, and confirming that each show_id (my primary key in my table) was not null or undefined, I was able to insert all the data returned from my api call. Below is a snippet of the code for executing this functionality.

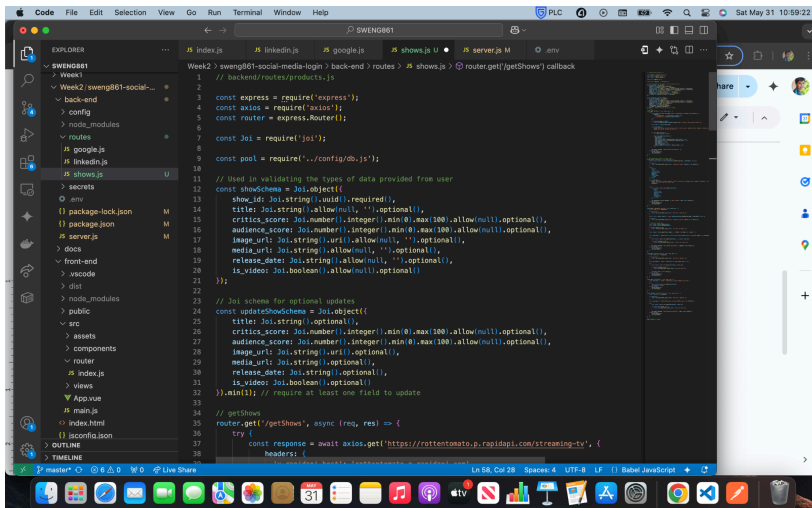


```
router.get('/getShows', async (res, res) => {
  try {
    const response = await axios.get('https://rotentomato.p.rapidapi.com/streaming-tv', {
      headers: {
        'x-rapidapi-host': 'rotentomato.p.rapidapi.com',
        'x-rapidapi-key': process.env.RAPID_API_KEY
      }
    });
    if (response.status !== 200) {
      return res.status(response.status).json({ error: 'failed to fetch shows from API' });
    }
    const shows = response.data.shows;
    const insertQuery = `
      INSERT INTO public.tv_shows (show_id, title, critics_score, audience_score, image_url, media_url, release_date, is_video
      VALUES ($1, $2, $3, $4, $5, $6, $7, $8);
    `;
    //console.log(shows);
    //console.log(shows.length);
    for (let show of shows) {
      if (!show.show_id) continue;
      await pool.query(insertQuery, [
        show.show_id,
        show.title || null,
        show.critics_score ? parseInt(show.critics_score) : null,
        show.audience_score ? parseInt(show.audience_score) : null,
        show.image_url || null,
        show.media_url || null,
        show.release_date || null,
        show.is_video !== undefined ? show.is_video : null,
      ]);
    }
    res.json({ message: 'Shows inserted successfully' });
  } catch (error) {
    console.log(error);
    res.status(500);
  }
});
```

After gathering my data, I could then start implementing my CRUD operations.

The first API call to implement was the create operation, which I named as createShow for the endpoint. When implementing this endpoint, the main concern here is validating that the data provided from the user is valid according to my database structure. To validate the data, I ended up installing a node package called Joi, which is a

popular data validation package for JavaScript. Below is a snippet of the Joi validation object I created.

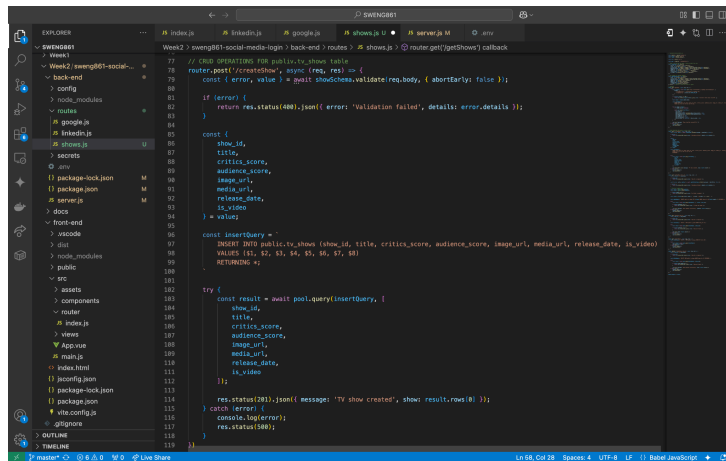


```
const showSchema = Joi.object({
  show_id: Joi.string().required(),
  title: Joi.string().allow(null).optional(),
  critics_score: Joi.number().integer().min(0).max(100).allow(null).optional(),
  audience_score: Joi.number().integer().min(0).max(100).allow(null).optional(),
  image_url: Joi.string().url().allow(null).optional(),
  media_url: Joi.string().allow(null).optional(),
  release_date: Joi.string().allow(null).optional(),
  is_video: Joi.boolean().allow(null).optional()
});

// Joi schema for optional updates
const updateShowSchema = Joi.object({
  title: Joi.string().optional(),
  critics_score: Joi.number().integer().min(0).max(100).allow(null).optional(),
  audience_score: Joi.number().integer().min(0).max(100).allow(null).optional(),
  image_url: Joi.string().url().optional(),
  media_url: Joi.string().optional(),
  release_date: Joi.string().optional(),
  is_video: Joi.boolean().optional()
}).min(1); // require at least one field to update

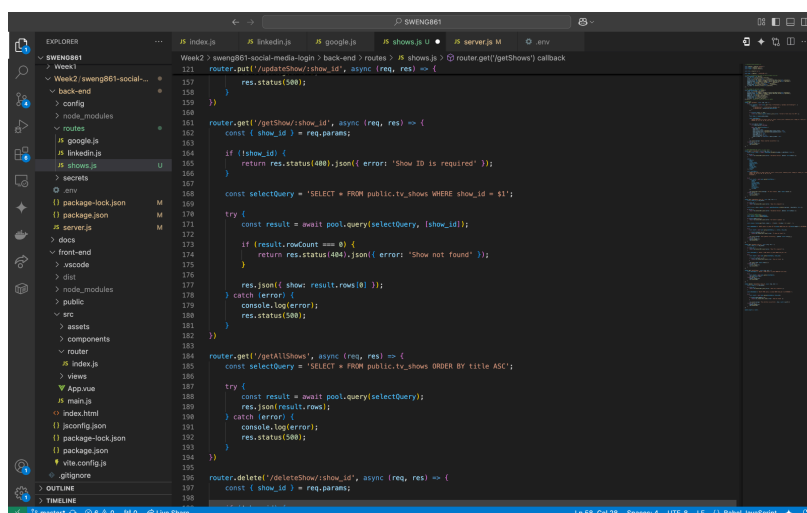
// getShow
router.get('/getShows', async (res, res) => {
  try {
    const response = await axios.get('https://rotentomato.p.rapidapi.com/streaming-tv', {
      headers: {
```

Joi is simple and easy to use. You can declare a validation object, giving it the exact tags of your column names, and tell it to check for variable type, specific values, null or not null, optional, etc. For my create operation, I created the showSchema variable as shown in the picture, which outlines the exact constraints of my table. Then, using my validation object, I can call Joi's built-in validation() function which checks if the user input matches the exact criteria in the validation object. If any part of it fails, my endpoint returns status code 400 with the message "Validation failed" and printing out the exact error message. If the validation check passes, I can proceed to insert the provided data into a new entry in my table. If all goes well with this query, the endpoint returns status code 201 with message "Tv show created" and it prints out the json object of the entry. Below is a snippet of the code described above.



```
77 // CRUD OPERATIONS FOR public_tv_shows table
78 router.post('/createShow', async (req, res) => {
79   const { error, value } = joi.showSchema.validate(req.body, { abortEarly: false });
80   if (error) {
81     return res.status(400).json({ error: "Validation failed", details: error.details });
82   }
83   const {
84     show_id,
85     title,
86     critic_score,
87     audience_score,
88     image_url,
89     media_url,
90     release_date,
91     is_video
92   } = value;
93   const insertQuery =
94     `INSERT INTO public_tv_shows (show_id, title, critic_score, audience_score, image_url, media_url, release_date, is_video)
95     VALUES (${show_id}, ${title}, ${critic_score}, ${audience_score}, ${image_url}, ${media_url}, ${release_date}, ${is_video})`;
96   try {
97     const result = await pool.query(insertQuery, [
98       show_id,
99       title,
100       critic_score,
101       audience_score,
102       image_url,
103       media_url,
104       release_date,
105       is_video
106     ]);
107     res.status(201).json({ message: "Tv show created", show: result.rows[0] });
108   } catch (error) {
109     console.log(error);
110     res.status(500);
111   }
112 });
```

Next endpoint implemented was the retrieve endpoints. I created two retrieve endpoints, one which passes in a show_id and retrieves a specific entry, and one which retrieves all entries in the table. Below is a snippet of the code for each endpoint.

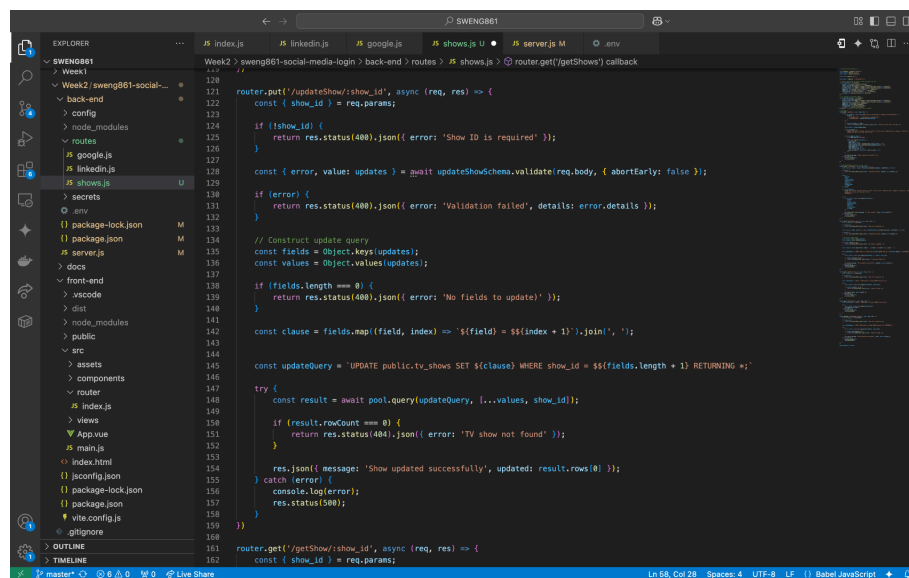


```
121 router.post('/updateShow/show_id', async (req, res) => {
122   res.status(500);
123 });
124 router.get('/getShow/show_id', async (req, res) => {
125   const { show_id } = req.params;
126   if (!show_id) {
127     return res.status(400).json({ error: "Show ID is required" });
128   }
129   const selectQuery = `SELECT * FROM public_tv_shows WHERE show_id = ${show_id}`;
130   try {
131     const result = await pool.query(selectQuery, [show_id]);
132     if (result.rowCount === 0) {
133       return res.status(404).json({ error: "Show not found" });
134     }
135     res.json({ show: result.rows[0] });
136   } catch (error) {
137     console.log(error);
138     res.status(500);
139   }
140 });
141 router.get('/getAllShows', async (req, res) => {
142   const selectQuery = `SELECT * FROM public_tv_shows ORDER BY title ASC`;
143   try {
144     const result = await pool.query(selectQuery);
145     res.json(result.rows);
146   } catch (error) {
147     console.log(error);
148     res.status(500);
149   }
150 });
151 router.delete('/deleteShow/show_id', async (req, res) => {
152   const { show_id } = req.params;
153 });
```

The first endpoint called getShow/:show_id, passes in a show_id to retrieve a specific entry in the table. For validation, I simply check that a show_id was passed into the endpoint. If not, I return status code 400 and print out "Show ID is required". If it gets past that, it queries the table for that entry. If it cannot find an entry with that ID I return status code 404 with message "Show not found". If it's found, it prints out the show in the response object.

The getAllShows/ endpoint is similar and easier. There is no show_id passed in, so it does no validation check. It simply does SELECT * on the table and prints out the result in the object. If something goes wrong it prints out status code 500 and console logs the error.

For my update operation, I had to create another Joi validation object. This one was slightly different because it won't have a show_id to validate. For the update operation, the user must pass in a show_id as a parameter to tell the endpoint which entry to update, then they have to pass in a json object in the body of which values to update. The Joi object only validates the body values. Then in the actual endpoint code, I simply validate that a show_id was passed into the endpoint similarly to the getShow endpoint. If validation passes, I construct my query to only update the values passed in, and I execute the query. If no entry under that show_id was found, it returns status code 404 with message "Tv show not found". If the query executed throws a server error or database error, the endpoint returns with status code 500 and prints the error. If everything goes to plan, the response object prints "Show updated successfully" and prints the entry itself. The code snippet can be found below.



```
router.put('/updateShow/show_id', async (req, res) => {
  //
  const { show_id } = req.params;

  if (!show_id) {
    return res.status(400).json({ error: 'Show ID is required' });
  }

  const { error, value: updates } = await updateShowSchema.validate(req.body, { abortEarly: false });

  if (error) {
    return res.status(400).json({ error: 'Validation failed', details: error.details });
  }

  // Construct update query
  const fields = Object.keys(updates);
  const values = Object.values(updates);

  if (fields.length === 0) {
    return res.status(400).json({ error: 'No fields to update' });
  }

  const clause = fields.map((field, index) => `${field} = ${values[index + 1]}`).join(', ');

  const updateQuery = `UPDATE public.tv_shows SET ${clause} WHERE show_id = ${show_id} RETURNING *`;

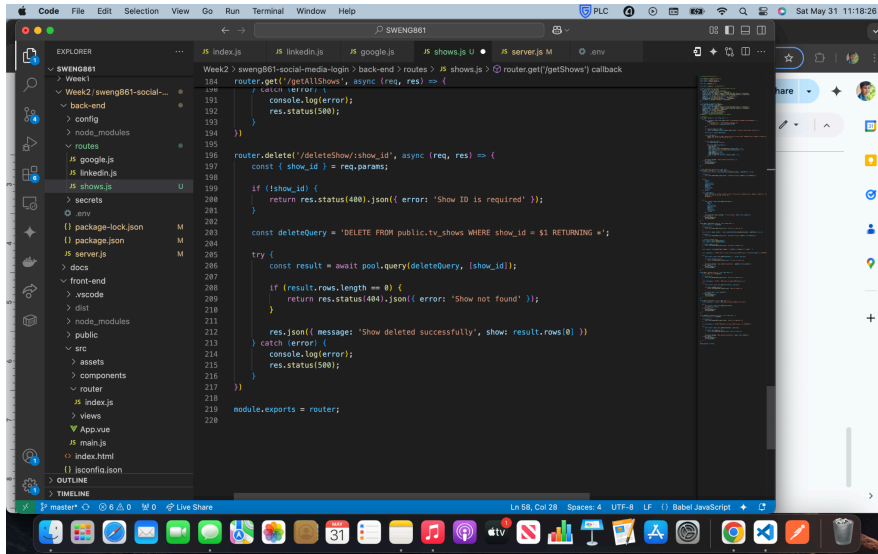
  try {
    const result = await pool.query(updateQuery, [...values, show_id]);

    if (result.rowCount === 0) {
      return res.status(404).json({ error: 'TV show not found' });
    }

    res.json({ message: 'Show updated successfully', updated: result.rows[0] });
  } catch (error) {
    console.log(error);
    res.status(500);
  }
});

router.get('/getShow/show_id', async (req, res) => {
  const { show_id } = req.params;
```

Finally, the last operation is a simple delete operation. For this, the user must pass in a show_id to tell the endpoint which entry to delete. The code, again, validates that a show_id was passed in. If not, it returns status code 400 and prints "Show id is required". If it gets past that, it executes the delete query. If no entry with that show_id was found, it prints "show not found" with status code 404. If a database or server error occurs, it returns status code 500 and prints the error message. If the entry was found and deleted, it prints "show deleted successfully" and prints out the object that was deleted. See the code snippet below.



Endpoint Summary

/createShow - endpoint to create a show entry. Takes in a json object in the request body. Request body must have a uuid show_id value. See example below:

```
{
  "show_id": "00000000-0000-0000-0000-000000000000", // uuid, required
  "title": "...", // string, optional
  "critics_score": 12, // integer, 0 <= x <= 100, optional
  "audience_score": 13, // integer, 0 <= y <= 100, optional
  "image_url": "...", // string, optional
  "media_url": "...", // string, optional
  "release_date": "...", // string, optional
  "is_video": true // boolean, optional
}
```

/getShow/:show id - endpoint to retrieve a specific show entry. Takes in a show id as a parameter.

/getAllShows - endpoint to retrieve all show entries.

/updateShow/:show_id - endpoint to update a specific show entry. Takes in a show_id as a parameter. Takes in a json object in the request body stating which columns and values to update the entry with. See example below:

<http://localhost:3000/updateShow/00000000-0000-0000-0000-000000000000>

```
{
  "title": "...", // string, optional
  "critics_score": 12, // integer, 0 <= x <= 100, optional
}
```

Note: This would update the entry with show ID: 00000000-0000-0000-0000-000000000000 with a new title and critics_score = 12.

/deleteShow/:show_id - endpoint to delete a specific show_id. Takes in show_id as a parameter.