Full-Stack Book Storage RESTful API

By Joseph Rodenas

Link To:

Client: https://replit.com/@rodeje894/SimpleAPIClient#index.html

*Server/Local Client: https://replit.com/@rodeje894/SimpleAPIServer#index.js

***IF YOU WISH TO FORK THE SERVER PROJECTS, YOU MUST CHANGE THE 'URL' IN CLIENT SCRIPT.JS**

However, forking should not be necessary for these projects, as a cross-origin request turns the server on, and it appears whatever issue with Replit 'preview' page has been resolved.

If server requests aren't working, ensure the server is running first- Replit occasionally turns them off for some reason, and automatically turning them back on doesn't always work.

Purpose:

The main purpose of this project is to develop a RESTful API and use the practice to better understand full stack development and concepts. This project features local and external clients, a webserver, routing, and a database. The ultimate goal was to provide a RESTful API that could be consumed by a cross-origin client, while also supporting local SCRUD operations.

Implementation:

This project, like most other full stack applications, was developed in layers to provide functionality for each portion of the stack. I had a decent understanding of what was needed for the front end functionality and what information would actually be sent, so rather than developing that first, I worked on the back-end to better understand what needed to happen for requests to be routed correctly. Admittedly, much of my design layout was referenced from Jaden Leake and Sen Zhang, though my project sought to combine the clearest functionality from both of their projects into a coherent, different solution. The front-end consists of HTML/JS/CSS, while the back-end is provided by Node and Express, while hosting a SQLite3 relational database.

Front-End:

The front-end of this project consists of both a local and external client. Those client pages are generated using HTML/JS/CSS and Bootstrap to help layout and design the UX. The main purpose of the clients is to offer support for SCRUD operations using the developed API endpoints. To accomplish this, 5 buttons are added for each SCRUD operation. For create, an HTML form submits a POST request to the server and sends relevant data to create a new entry. The other buttons are listened to and handled to send Fetch requests for POST, GET, PUT, and DELETE. For requests that require a response, read and select, asynchronous requests are needed because of access to the database. Select actually requires an async POST request that

sends data and waits for a response. Though fetch is used for each, different techniques are employed: some using await, some promises, some default (get), others specified (put, delete).

Server:

The server is actually relatively straightforward. The index.js file provides the necessary modules like CORS and body-parser, links to a static public folder for the local client files, and defines a subroute '/books' to host endpoints. The routes.js file creates a connection to the database class, and creates each necessary route that coincides with a database method: post, get, put, delete. Create and Update both check if the input is the error values for bad input, and return if they are. Different techniques are used to send data to the database methods: some send the request object, some parse the data beforehand, and some save a variable and send that. Unlike Dr. Zhangs example, my clients both go directly to the endpoints and then the database from one place, where his creates a different connection for local or external requests.

Database:

The database is a large portion of this project. Unlike other projects that simply create and operate on the database using the endpoints, this application separates the logic for each portion. A database class is defined that declares separate methods for each query operation. Inside routes.js, createTable method is called to generate the table if now already done. The attributes are: integer id, text title, text author, integer rating, and price as a real (float). Then, each server endpoint coincides with a query method. For example, the route 'put, /books/changeOne' calls the method 'updateOne' that receives all fields input by the used to then perform an update query. Assuming an input passes validation, one last conversion is applied to ensure that any data stored to the table is of the intended type. When necessary, the database methods are passed a response object that is then used to return the query result to the client.

<u>Obstacles:</u>

This was by far the most challenging project of the year, but also the most rewarding. Many different obstacles were encountered but each solved in turn. For example, near the end of the project I became concerned with input/data validation. I had already implemented type conversions before storage to the database, but they were assuming the types could be converted to the required type. To prevent the database from storing unintended data, I decided to use client-side validation (which is usually not best practice). Some database requests generate errors, so in these cases little/no validation was required, although others, like delete, still try to delete an entry even if the input didn't match what was expected, so I very simply checked if the input matched some basic requirements. Another issue with my code is that responses are not provided for requests that do not require one: (delete, update, create). The solution would be to add responses to those routes, making each request

asynchronous, and providing parsing on the client to display the response data. This would require much more code and logic for what I thought was not worth the result for the user. The absolute most challenging obstacle was understanding the back-end file structure. Both Jaden and Sen used the same structure that I was unfamiliar with. My solution was to understand what both of their projects intended to do and find the similarities between them that made them functional. Once I could separate the logic into distinct files, it became easier to identify what was missing and where things needed to happen: for example, index.js points to a subroute, endpoints are provided in the subroute, those endpoints call a database js file that perform queries on a database instance. After developing the structure (and bug fixing), it was easy to build clients to consume the resources provided by my API.