Full-Stack Bookstore

By Joseph Rodenas

Link To: Project Code **NOTE: Please fork the code base!**

Purpose:

This project was made to practice and explore full stack development concepts and implementation. The project features a webstore, shopping cart, server, and database. While it is full stack, there is no cross-origin support so only local files and requests are served.

Implementation:

The project was accomplished by dividing it into each layer of the stack and developing each separately. For instance, the front end consists of a responsive single page application, and a separate shopping cart page. The server is created with NodeJS and Express. The database is a sqlite3 relational database hosted on the server.

Front-End:

The front end was developed using the bootstrap framework and a template provided by W3Schools, both of which have thorough documentation already. It consists of two separate HTML pages, a homepage and cart. There is nothing particularly interesting about the format or design of these pages, they simply exist to house data points for a user to select and submit. Because the cart page was separate and I wanted the cart to be persistent, localStorage is used to house both the users item choices and the item count to display on the cart icon. The cart page grabs these items from storage, displays them, and, when needed, POSTs them to the server. The JavaScript required for the homepage is primarily just button handlers and parsing the users cart choices into localStorage as a single string. For the cart page, the script is a bit denser as there are multiple dynamic functions at play. For example: loading string from storage, parsing the string into separate arrays and displaying the contents/ price of the items, logic handling for deleting individual items, clearing the cart, manipulating POST form data before submission, fetching GET response data and displaying the data in tables.

Server:

The NodeJS and Express server were one of the easier parts to implement. For this project, only two routes were needed: post an order, and get all orders. Included in the server index file is JavaScript to handle parsing of the received data into separate arrays of strings and a single string value for the order total. Also included is script to create the database, as well as functions for writing to and reading from the database. Of note here are static references for html resources, so that NodeJS can route source requests in HTML (for images, JavaScript, and CSS).

Database:

The database is an SQLite3 relational database consisting of one table: purchase. The only thing of note is that the table is only created if it does not already exist, as this prevents the error message every time the server is started. The table columns are an autoincrementing id, text for the list of book names, next for the list of book prices, and a real to hold the order total price. Two operations are performed on the data: inserting new order data (after parsing) and selecting all rows from the table.

Obstacles:

This project was very challenging but incredibly rewarding and informative as well. Despite developing the layers separately, each one provided their own unique challenges. On the front-end, deciding to store the cart data as a single string gave me a hint at what was to come with the database. localStorage converts all datatypes to strings so I figured it would be easier to use one data point rather than trying to manage a dynamic number generated by cart items. Surprisingly difficult to figure out was the cart symbol. I needed to read items remaining in storage whenever the page was loaded, as maybe the user was returning from the cart. Related to this was adding the functionality to delete individual cart items, as I needed to learn bubble handling for a dynamic number of buttons, as well as repeatedly converting between local variables/stored values/strings/arrays etc. For the server, I spent a good amount of time reading documentation on NodeJS and Express, as the structure was confusing in the beginning. Namely, routing and launching on a port gave me lots of trouble. The most difficult obstacle by far was retrieving data sent from the server. Because this is a local project, I intended to use html forms for getting and posting data, however, did not want to load a new page to display retrieved database information. When I finally figured out how to use fetch in JS to capture JSON response data, I thought I had it solved. I then learned that SQLite queries are all handled asynchronously, so I spent a lot of time playing with promises and callbacks so that my function would return all database rows, operate on the data, and send it as a response to the client. This was incredibly confusing as asynchronous functions behave differently, and Express routing needs the response object in order to reply. My eventual solution was to just perform the asynchronous operation directly in the routing function. This is not as clean of a fix as I would like but it gave me the effect I wanted. In the future (the final), I will likely just display the output as its own page, rather than returning it to the client and rendering it dynamically.