Thread ‘n Things Build Specifications

To begin the build of the Thread ‘n Things site, we need to create a *package.json* file. To do this, we create our root folder in our new file folder system and open it in Visual Studio Code. We then open a new terminal in Visual Studio Code and use the command **npm init -y**. This will populate a default *package.json* file that we need to build on. Check the name of the main file. It may be *server.js* or *index.js.* We have changed the name to *app.js* in this case. We then want to be sure to add a type specification in the *package.json* file. **“type”: “module”** is added to the *package.json* file to ensure that the code can adhere to the rules of ES6 syntax. Then, we take a moment to ensure that our file folder system is complete and correct. There should be a **public folder** and a **views folder** inside the main folder at the same level as the *app.js* and *package.json* files. From there we need to install Express and EJS as dependencies. We can do both at once utilizing the npm command **npm install express ejs**. At this point, we can also install nodemon as well. The command for this is **npm install nodemon --save-dev -g.** Going back to the *package.json* file, we will now add a line inside the **“scripts”** section. This script, **“start”: “node app.js”**, will tell the application to start using this command. Now, we can enter the **npm start** command into the terminal to start our initial application.

We now need to make our empty page files utilizing the *.ejs* file type. For this initial site build, we need *index.ejs, products.ejs, about.ejs* and *contact.ejs.* Once that is complete, we can set up the initial server code and routes. This code is as follows:

**import express from “express”;** This section brings in Express using the ES6 syntax

**const app = express();** Here we are creating an instance of Express

**const port = 3000;** This tells the application to use port 3000

**app.set(‘view engine’, ‘ejs’)** This sets the view engine and then Express loads the module internally

**app.get(‘/’, (req, res) => {** This is the first route, calling the index page at *index.ejs*

**res.render(‘pages/index’, {title: “Home”});**

**});**

**app.get('/products', (req, res) => {**

**getDbConnection()**

**.then((db) => {**

**return db.all('SELECT \* FROM supplies');**

**})** This section is the route to the

**.then((products) => {** products page. It links to the

**res.render("pages/products", {** database that will be created later

**data: products,**

**title: "Products",**

**});**

**})**

**.catch((error) => {**

**console.error(error);**

**res.status(500).send('Internal Server Error');**

**});**

**});**

**app.get('/about', (req, res) => {**

**res.render('pages/about', { title: "About Us" });**

**});** These are the routes to the About

and Contact pages

**app.get('/contact', (req, res) => {**

**res.render('pages/contact', { title: "Contact Us" });**

**});**

**app.listen(port, () => {** This tells the application to listen for requests at the given port

**console.log(`App listening at port ${port}`)**

**});**

This completes the initial server set up. Next, we add the middleware. Add the following lines into the *app.js* file in the import section below the **import express** line.

**import path from "path";**

**import { fileURLToPath } from 'url';**

**import { setupDatabase, getDbConnection, popDatabase } from './database.js';**

These lines will create ES6 syntax file references and directory references.

Underneath the **const port** line, add the following lines of code.

**const \_\_filename = fileURLToPath(import.meta.url);**

**const \_\_dirname = path.dirname(\_\_filename);**

**app.use(express.static(\_\_dirname + "/public"));**

These lines get resolved paths to the files, get the names of the current directories, and make the public folder the default. After this is completed, we can test the server by returning to our *index.ejs* file, adding a placeholder tag such as **<h1>Hello World</h1>**, and running the server with Node. This is completed by using the start command, which was **node app.js**.

Now that the server is up and running, we can begin to populate our database. First, we need a table of data to pull from. For Thread ‘n things, we created a table to hold the product id numbers (**id**), the file path to the images (**imgFolderPath**), the product names (**product\_name**), the category of each item (**category**), the prices of each item (**price**), the quantity currently in stock (**quantity\_stock**), the supplier of each item (**supplier**), and the item description tag line for the item cards (**tag\_line**).

To begin, we create a new file titled *database.js*. At the top of the file, we need an import line to create a dependency using **sqlite**.

**Import { open } from ‘sqlite’;**

Then, we can use sqlite to create a function that will build our database. For our website, the function will be as follows:

**function createTable(db) {**

**return db.exec(`**

**CREATE TABLE IF NOT EXISTS supplies (**

**id INTEGER UNIQUE,**

**imgFolderPath VARCHAR(1000),**

**product\_name TEXT,**

**category TEXT,**

**price VARCHAR,**

**quantity\_stock INTEGER,**

**supplier TEXT,**

**tag\_line TEXT**

**)**

**);**

**};**

These lines build the table for the database and assign data types to each variable in the table. After this function builds the table, we need another function to populate the table. This function will appear as follows:

**function insertInfo(db) {**

**return db.run(`**

**INSERT INTO supplies (id, imgFolderPath, product\_name, category, price, quantity\_stock, supplier, tag\_line)**

**VALUES**

**(020,**

**'/images/organizer.png',**

**'Embroidery Floss Organizer',**

**'Accessories',**

**'8.00',**

**55,**

**'CraftTools',**

**'Keep your embroidery threads tangle-free and beautifully organized with our stylish and functional embroidery floss organizer!'),**

**…**

**`);**

**};**

Each item being added to the table will follow the same template as the item listed above. The **INSERT INTO** line tells the function to add the items in the order listed, so each item variable data must match the data type identified above. These functions create and populate our database, so now we need to export the database so it can be accessed by the server.

**export const getDbConnection = () => {**

**return open({**

**filename: './public/database/products.db',**

**driver: sqlite3.Database**

**});**

**};**

This export function sends the information contained in the new table and creates a new file titled *products.db*.

**export const setupDatabase = () => {**

**return new Promise((resolve, reject) => {**

**getDbConnection()**

**.then(db => {**

**console.log("Creating Database Table");**

**return createTable(db);**

**})**

**.then(() => {**

**resolve("Database setup complete.");**

**})**

**.catch(error => {**

**console.error("Error setting up database:", error);**

**reject(error);**

**});**

**});**

**};**

This export function utilizes a promise to check the database initialization at each step and alert the user of its completion status; pass or fail.

**export const popDatabase = () => {**

**return new Promise((resolve, reject) => {**

**getDbConnection()**

**.then(db => {**

**console.log("Populating DB");**

**return insertInfo(db);**

**})**

**.then(() => {**

**console.log("Info Population complete");**

**resolve("Info population complete.");**

**})**

**.catch(error => {**

**console.error("Error populating database:", error);**

**reject(error);**

**});**

**});**

**};**

This export function utilizes a promise to check the population of the database, and alerts the user at the completion of each step; pass or fail.

We now go back to the *app.js* file, where we add the functions to call the database initialization and population.

**setupDatabase()**

**.then((msg) => {**

**console.log(msg);**

**})**

**.catch((mishap) => {**

**console.error(mishap);**

**})**

**.finally(() => {**

**console.log("Finished setup.");**

**})**

**popDatabase()**

**.then((msg) => {**

**console.log(msg);**

**})**

**.catch((mishap) => {**

**console.error(mishap);**

**})**

**.finally(() => {**

**console.log("Finished population of db");**

**})**

**global.param1 = null;**

Now that the server and the database are set up and functional, we can move on to webpage creation. Our first step in this process is to create our partial files. These will be saved in the **views/partials** folder. For the Thread ‘n things site, we utilize the partials *head.ejs, menu,ejs, footer.ejs,* and *productmodal.ejs*. These files will contain information that is referenced in one or more pages, and will allow us to update information in one place and have it automatically be applied to every referenced page.

The *head.ejs* file will contain link to all the scripts and styles that are needed to run the site. These files include a local CSS file, references out to external CSS files, script links to the j Query api files, and script sources that allow us to utilize Bootstrap for functionality and design.

The *menu.ejs* file contains the code for a Bootstrap navigation menu. The files uses Bootstrap syntax for styling, inline styling to adjust colors to match the business color scheme, and links to our local data. A logo is added for the business that also functions as a hyperlink back to the index home page. The navigation buttons themselves are linked to their respective pages using the tags set up in the routes defined previously.

The *footer.ejs* file contains placeholder code for a social media navigation bar. Currently, the links do not go anywhere, but as the social media sites are created, the proper links can be added into the blank **href** sections of their respective buttons. There is also a copyright tag included at the bottom. For now, we have left it in to attribute the design to Bootstrap.

The *productmodal.ejs* file contains the code that is needed to create the pop up that will be functional on the Products page. This code is obtained from Bootstrap and has empty tags for the information that will fill it later on. This is to provide a place for the function that will be created later to display the product information. There are also **X** and **Close** buttons included to allow the user to close the modal once it is open on their screen.

Each website page will include EJS **include** tags to reference these partials. The *productmodal.ejs* file will only be referenced on the *products.ejs* page. The syntax will be as follows:

**<%- include('../partials/head.ejs') %>**

**<%- include('../partials/menu.ejs') %>**

**<%- include('../partials/productmodal.ejs') %>**

**<%- include('../partials/footer.ejs') %>**

This tells the page to access the partial file and pull in that code as if it were all written directly onto the file.

We can now move on to the Products page that will use the information from the database we created earlier. Aside from the partial inclusions listed previously, The Products page will have a div container populated with product listing cards. The code set up for this div and the cards is originated with Bootstrap. The cards are set to have a max width of 300px, but no height specification to allow for longer or shorter product information in the listings. The cards will be created dynamically in the code by using a **forEach** function. The **forEach** function creates a div for each card to be styled the same way. In each card, the function pulls data from the database to add the products image, the product name, the category, and the tag line for each item in the table. After that, there is a button also create in the function for each item that will activate a modal to pop up and let the user know the price and quantity of an item.

After the **forEach** function, the *products.ejs* page includes a short script to provide functionality to the modal buttons in each card. This script takes the date from the table and translates it into strings that can be read by the program. Is also utilizes a **forEach** function to identity the button it needs based on which button was clicked. The empty divs from the *productmodal.ejs* file are identified. Then, the product needed to fill the divs is identified using the id number for the product selected. The information is sent to the empty divs and used to display the product name, price, quantity, and supplier of each item on the pop-up modal.

The *index.ejs, about.ejs,* and *contact.ejs* pages are currently populated with filler text and sample images to display what a fully functional website for the company would look like. The *contact.ejs* page has a sample form placed to demonstrate how customers could reach out to give comments or suggestions for our business.