COMP 2406A
Fall 2020 - Tutorials #6 and #7

Objectives

- Use Express to develop an HTTP web application
- Begin using RESTful design in web applications
- Practice providing functionality to create and retrieve resources from a server
- Increase web service scalability by storing data in files
- Use NPM to organize dependencies and allow easy installation

Problem 1 (Creating an Express Application)

In this tutorial, you will be creating an Express-based application to store/serve recipes. This application will allow users to create new recipes, browse existing recipes, and view recipes. To start, download the T06-T07-BaseCode.zip file from cuLearn. This contains some base HTML and Javascript, along with an Express-based static server that you can use as a starting point. Before running the server code, you will have to install the required modules from NPM. First, execute the command `npm init` from the terminal within the directory where your tutorial code is. This will create a package.json file to store your project's dependency information. You will then have to install Express using the command `npm install express`. If this was successful, Express should be added into your package.json file as a dependency. Run the server code and ensure that the home page shows up properly when you access http://localhost:3000/ and http://localhost:3000/index.html. Inspect the code and ensure you understand how it is working before moving on to the next problem.

Problem 2 (Creating New Recipes)

Initially, there will be only three recipes on your server, which are included in the *database* object within the server code. Open the `create.html` and `addrecipe.js` files in the `public` directory and inspect the code. The `create.html` page allows a user to enter recipe information. When the Save Recipe button is clicked, the `addrecipe.js` file sends the recipe data to the server using a POST request to the resource /recipes.

Within the server code, all recipes will be stored in a single object called `database`. The keys of this object will be unique IDs and the values will be the recipes associated with those IDs. Add a route within the server code to handle POST requests to the `/recipes` resource. The handler for this route should:

1. Extract the recipe object included in the POST request body. You can use the built-in `express.json()` middleware for this.
2. Generate a unique ID for the new recipe. See notes on "Using the '`uuid`' Package" at the end of this problem.
3. Add a new entry into the recipes object with the key being the unique ID and the value being the recipe object. There is a chance the ID you generate will overlap with an existing ID, but the chance of matching is so *extremely* low that you can ignore the possibility.

The `create.html` file provides a button to randomly generate recipes. Test your code by using this functionality to add a few recipes to your server. For now, you can log the contents of the recipes object to see that it is storing the correct data. We will add recipe browsing functionality in the next problem.

Using the '`uuid`' Package:
The `uuid` package provides a method for generating 'universally unique identifiers'. You can use this package to generate unique IDs for resources you will store on your server (recipes in this case). Alternatively, you could write your own code to ensure that IDs are unique (i.e., by using the recipe name). If you will not have billions of objects, the uuid package may be overkill, but it is an easy way to generate unique IDs.
To use the package, first install it via npm:

```
npm install uuid
```

You can then require the package with:

```
const uuidv4 = require('uuid/v4');
```

And create new UUID values with:

```
uuidv4();
```

Problem 3 (Browsing Recipes)

The `index.html` page also includes an HTML link to the resource "/recipes". Create a GET route for the resource /recipes on the server. Add a template engine to your server and use it to generate an HTML response that contains a list of all recipes stored on the server. Each entry in this HTML list should have the text of the recipe name and a link to the address "/recipes/*uuid*", where *uuid* is the unique ID assigned to that specific recipe. For example, a single entry may look something like:

    <a href="/recipes/61e8b921-bd94-4526-b619-0b0444e390b9">Hamburgers</a>

Problem 4 (Serving Recipes)

Now, add a route to the server that handles GET requests to the parameterized address /recipes/:uuid. In this case, the :uuid parameter will indicate the unique ID of the recipe the user is trying to retrieve. If the specified ID indicates a recipe that exists on the server, then the handler should generate the HTML to represent the specified recipe and send it as the response to the user (like the previous problem, use the template engine to do the generation). As with the list, this HTML does not have to be anything fancy, but should include all the data about the recipe (name, prep/cook time, description, ingredients list). If the ID does not exist, then the server should respond with a 404 error.

At this point, you should be able to add and retrieve recipes. One of the advantages of using NPM to manage your project dependencies is that it can simplify the install process. Additionally, you no longer need to include the node_modules directory with your code in order to allow somebody else to use that code. Copy your server code and resources, along with your package.json file to another directory (leave the node_modules directory out). Navigate to the new directory in the terminal and run the command `npm install`. This should install all the dependencies for your project and you should then be able to run your server again without any additional steps. It will be required that your third assignment is submitted in this way, without the node_modules directory.

Problem 5 (Moving to a File-Based Design)

There are a couple significant issues with the decision to store all recipe data in a local variable (i.e., in the RAM of the computer that's running the server). The first is that when the server is stopped, all recipe data that has been added is lost. Each time you start the server, you start with the default database that is included in the server code. Additionally, storing the database in RAM significantly limits the number of recipes you can store. To address these problems, you can store the information in local files on the computer. You can then create/read files to answer queries for information from the users. Later in the course, we will further improve on this approach by incorporating databases into our applications.

To start, create a new folder called 'recipes' within your directory structure. You can then store all your recipe data within this directory. For each recipe, you will store a single file with the name *`uuid`*`.json`, where *`uuid`* represents the ID of that recipe. Note that, just like sending data through HTTP, you can write/read JS objects to/from files, allowing you to easily save/load objects in files.

Delete the database variable from your code and modify your POST handler for the `/recipes` route so that it uses a file-based approach. When each request is received, instead of saving it into the local variable, write a new file with the appropriate path/name.

Modify your GET handler for the `/recipes` route to generate the HTML response by reading the names of files within the recipes directory. A simple way to do this is to use the ‘fs’ module’s `readdir` method to get an array of the filenames within the recipes directory. When the client requests the information for a specific recipe, you can then read the recipe object from each file to get the necessary information. For this tutorial, you can use the `readFileSync` method in the ‘fs’ module. This will read the files synchronously and should simplify your code. **Note, however, that reading the entire recipe and reading the files synchronously are both bad design decisions**. See the additional problems if you are interested in improving the performance of the system.

Modify the GET handler for the `/recipes/:uuid` route to use the file-based approach. As you are given the ID of the recipe, and the filename matches the ID, it should be straightforward to read the contents of the recipe if the file exists.

Once you have done this, you should have a recipe database that can store many recipes on a local disk.

Problem 6 (Further Challenges)

If you are looking for more interesting extensions to this code, consider the problems below.

1. If you followed the basic advice in problem #5, you will have read the entire database (names, IDs, ingredients, descriptions, etc.) in order to generate the list of recipes in response to GET requests for /recipes. The only information you require to generate the list of recipes, though, is the name and ID. Can you think of a way to modify your solution so it does not have to read unnecessary information? What trade-offs need to be considered?
2. Reading files synchronously is easier from a coding perspective but is a ***very*** bad practice in general. Imagine trying to run a popular website where the entire server stops responding to requests whenever any user anywhere requests a resource. Can you modify the code to read everything asynchronously? The difficult part will be deciding when you have finished reading all the information. For example, if there are 10 files, you need to ensure you have read information from all 10 before ending the response.

This is not straightforward, as asynchronous method calls return immediately. Consider adding a variable to track how many reads have completed and use this variable within your callback to decide if all files have been read.

3. Another function you may support is the editing of recipes. Modify the HTML representation that a user gets in response to recipe requests so that they can edit the recipe data and send the new data back to the server using a PUT request. Add a parameterized PUT handler for /recipes/:uuid so a client can PUT a new representation of the recipe containing modified data and have those changes remembered by the server.
4. If you have many recipes, it will also be infeasible to list all of them when the user makes a GET request to /recipes. Instead, implement a pagination mechanism that will show, for example, the first 25 recipes for an initial request. Provide next/previous buttons to allow the user to navigate the complete list of recipes.
5. In general, one resource may be requested frequently, while many others are rarely requested. For example, there may be a few very popular recipes. While you can't store all recipes in RAM, you can store some number of them (e.g., 1000, or X in general). Add a local variable to store requested recipes. This technique is known as caching. A simple cache can just store the last X recipes that have been requested. If a recipe is in the cache, you can serve it from RAM instead of reading the file. You can also add more complexity into the cache by trying to remember requests that are more frequent. Try adding a value to track how many times each recipe is requested. Modify the cache so it remembers the most frequently requested recipes (note: there are additional modules that could help with this, like '`heap`').
6. The RESTful design principles discussed in class specify that the server should mark data as cacheable or not. It is logical to assume that the recipe data on your server will not change often. Add the appropriate headers to your response to enable clients to cache recipe data locally. Request the same recipe a number of times to verify that this mechanism is working.