Lab 3: Task List Server

Due Date: Before Lab on Week 4

Overview

In this lab, we'll be extending our Task List from Lab 2 into a server that returns tasks. Additionally, we'll be saving the responses to a local file.

Objectives

1. Serializing and saving objects to memory
2. Writing an HTTP request handler
3. Testing out your server.

Task 1: Read Code

My solution to Lab 2 is included in the task-list folder. You won't be able to access this until you submit your 1 submission for Lab 2. You can read through it to see how you compared with mine.

utils

The first big chunk of code is in the utils/send-response.js function. This sends an HTTP response given the response object, a status code, and an optional response body. Not all request types will need to send a body.

The next util function is readBody. This is used to read the body of a request. We haven't talked yet about Events in Node.js but this is how events are consumed. The main thing to be aware of is that this function returns a Promise, so you won't have the value immediately. How do you access the value returned by the readBody Promise?

```
```

server.js

The bulk of your code will live in the server.js file. We create a handleRequest function which takes in two initialization parameters - the tasklist and a file (to write out to / read from). If you're wondering how it'll work, I highly recommend looking through some examples and this reference. Notice that the handleRequest function returns a function. This creates a closure over the tasklist and file variables whose references will stick around after the function has been executed once.

Below that large function we initialize the server. Don't worry too much about the require.main piece, it's a Node.js specific detail to make sure autograding doesn't accidentally create a bunch of HTTP servers. Let's breakdown the other piece of code though

```
const tl = TaskList(); // create an empty task list
let server = http.createServer(handleRequest(tl, "tasks")); // register request handler
server.listen(8080); // listen on port 8080 so you can send requests to localhost:8080
```

Task 2: Write the request handler

I recommend most of your logic goes into the function returned from handleRequest because of the closure created. You don't have to do some weird returning functions thing that we'll go over in class later in the quarter. You'll have to handle three different types of paths which might have different request methods:

1. GET /tasks - lists all tasks
2. POST /task - create a new task
3. GET/PUT/DELETE /task/:id
    a. GET - reads the task with the given id
    b. PUT - marks the task with the given id as completed
    c. DELETE - deletes the given id

If this seems very similar to what you did in Lab 2, that's good. Each of these "endpoints" corresponds to a function that we created in Lab 2.

Knowledge Check

Before coding, make sure to note down all the error codes you plan to use and check them with the TA

1. GET /tasks

2. POST /task
3. GET /task/:id
4. PUT /task/:id
5. DELETE /task/:id

GET /tasks

This endpoint should only respond to GET requests. If any other request method comes in to this endpoint, you should return a correct status code.

POST /task

This endpoint should respond to POST requests and can be combined with the /task/:id endpoint or stay separate. If you do choose to keep this separate from the /task/:id endpoints, make sure to return the correct status code for other types of request methods.

Note that this endpoint will use Promise logic because you have to "read the body" of the request using one of the utils.

GET / PUT / DELETE /task/:id

For these endpoints, they can and should be grouped logically. Make sure return the correct status codes for each of these endpoints as well as a NOT FOUND error anytime the ID is not found.

Task 3: Loading and Saving Data

The slightly harder part of this lab is to load and save your task list from a file. This file lives in the directory you're running your server from in a file called "tasks" (with no file extension).

Updating the TaskList capabilities

In the tasklist.js file, you'll see two new functions: serializeTasks and loadTasks. These correspond to saving and loading the tasks.

1. Serialize the tasks by returning a formatter string representing the task list. Think about what common format we use in Web Programming to send data. You should return something which is an Array of tasks from this function.

2. Loading the tasks should verify that each task given is actually a task. Then create a new Task for each element in the given array. Make sure that if other fields exist (like a createdDate) that those are loaded as well.

Saving the TaskList

After each function which *mutates* the task list, write out to the file. The easiest way to do this (with such small amounts of data) is to overwrite the file each time. We're not going to be focused on file manipulation in this course so don't worry too much about performance when writing a file. The performance you *should* care about is how long does it take to send the response to the user. With that in mind, when do you think we should write to the file? Before or after the response is sent? What's your reasoning?

Loading the TaskList

Once you can write tasks out, you also need a way to read the task list. Before you start listening for requests, you should *load* the tasklist itself from the file. Think about the error scenarios that may occur when reading from the file:

1. Does the file actually exist?
2. Is the file properly formatted?