

***W2*** *PRACTICE*

# Native HTTP and Manual Routing

 *At the end of this practice, you can*

* **Create** and run a native Node.js HTTP server ü **Manually implement** route handling using conditionals.

* Serve **static files** using fs. ü Parse form data from POST requests.
* Debug and enhance server code using console outputs.

 *Get ready before this practice!*

* **Read** the following documents to understand Nodejs built-in HTTP module:

<https://nodejs.org/api/http.html>

* **Read** the following documents to understand Anatomy of an HTTP Transaction: <https://nodejs.org/en/learn/modules/anatomy-of-an-http-transaction>

 *How to submit this practice?*

* Once finished, push your **code to GITHUB**
* Join the **URL of your GITHUB** repository on LMS



## EXERCISE 1 **ANALYZE**

### Goal

ü Identify and fix the bug. ü Understand the request-response cycle in Node.js using the http module. ü Explain the role of res.write() and res.end() in sending data back to the client.

 For this exercise, you are provided with a minimal server.js file. Read and run the code and observe how it behaves.

|  |
| --- |
| // server.js |
| const http = require('http');  const server = http.createServer((req, res) => { res.write('Hello, World!'); return res.endd();  }); server.listen(3000, () => { console.log('Server running on http://localhost:3000'); }); |

**Q1 –** What error message do you see in the terminal when you access http://localhost:3000? What line of code causes it?

The message was: Connection was forcibly closed by a peer, which cases by the line return res.endd();

**Q2 –** What is the purpose of res.write() and how is it different from res.end()?

restwrite() send a chuck of data to the client, and it does not end the response (the connection stays on), whereas rest.end() signal that you’re done sending the response.

**Q3 –** What do you think will happen if res.end() is not called at all?

* The response stays open, the client keeps waiting.
* The browser shows a loading spinner and doesn’t finish the request
* Then, it may time out, showing an error like this site can’t be reached.

**Q4 –** Why do we use http.createServer() instead of just calling a function directly?

Because calling a function directly only runs once immediately, it doesn’t wait for real web requests. But what http.createSever () does is that:

* It sets up a listener for HTTP requests on a port
* It waits for connections from Chrome, Postman, curl, etc
* It calls your callback function every time a request comes in.

**Q5 –** How can the server be made more resilient to such errors during development?

* Use try…catch and Error Handling
* Use nodemon for Auto-Restarting
* Set timeouts and Limits to protect against hanging connections

## EXERCISE 2 **MANIPULATE**

### Goal

* Practice using req.url and req.method.
* Understand how manual routing mimics what frameworks (like Express) automate. ü Serve both plain text and raw HTML manually.

 For this exercise you will start with a **START CODE (EX-2)**

**TASK 1 -** Update the code above to add custom responses for these routes:

|  |  |  |
| --- | --- | --- |
| **ROUTE** | **HTTP METHOD** | **RESPONSE** |
| /about | **GET** | About us: at CADT, we love node.js! |
| /contact-us | **GET** | You can reach us via email… |
| /products | **GET** | Buy one get one… |
| /projects | **GET** | Here are our awesome projects |

Use [VS Code’s Thunder Client](https://www.thunderclient.com/)) (or other tools (POSTMAN, INSOMIA) of your choice or curl on your terminal to make request.

Example output

curl http://localhost:3000/about -------------------------->

About us: at CADT, we love node.js!

curl http://localhost:3000/contact-us -------------------------

-> You can reach us via email…

**TASK 2 –** As we can see the complexity grow as we add more routes. Use switch statement to arrange the code into more organized structure.

### ❓ Reflective Questions

1. What happens when you visit a URL that doesn’t match any of the three defined?
2. Why do we check both the req.url and req.method?
3. What MIME type (Content-Type) do you set when returning HTML instead of plain text?
4. How might this routing logic become harder to manage as routes grow?
5. What benefits might a framework offer to simplify this logic?

**Answers**

1. IF the URL doesn’t match any of the defined it pops the message 404 Not Found.
2. We check both the req.url and req.method because a single URL can respond differently depending on the request method.
3. The MIME type (Content-Type) text/html is set when returning HTML instead of plain text.
4. This routing logic becomes harder to manage as routes grow because of:

* Too many conditions: the code blocks become long, hard to read, and error-prone.
* No separate of concerns: all the logic is crammed into one place ( server.js ), which lead to difficulty in testing, reuse, collaborate.
* Poor Scalability: when adding features (like middleware, authentication, sessions), this simple structure becomes unmanageable

1. A framework (like Express.js for Node.js) can simplify routing by offering:

* **Cleaner syntax and route definitions** (e.g., app.get('/home', handler) instead of manual if checks).
* **Middleware support** to handle repeated tasks like logging, authentication, and error handling.
* **Modular structure** to split routes and logic into separate files, improving readability and maintainability.
* **Built-in utilities** for handling requests, responses, routing, and even templating, reducing boilerplate code.

## EXERCISE 3 **CREATE**

### Goal

ü Practice handling POST requests. ü Parse URL-encoded form data manually. ü Write and append to local files using Node.js' fs module. ü Handle async operations and errors gracefully.

 For this exercise you will start with a **START CODE EX-3**

**TASK 1 -** Extend your Node.js HTTP server to handle a **POST request** submitted from the contact form. When a user submits their name, the server should:

1. **Capture the form data** (from the request body).
2. **Log it to the console**.
3. **Write it to a local file** named submissions.txt.

Testing, go to /contact on broswer and test **Requirements** • Handle POST /contact requests.

* + Parse raw application/x-www-form-urlencoded data from the request body.
  + Write the name to a new line in submissions.txt.
  + Send a success response to the client (HTML or plain text).

### ❓ Discussion Questions

1. Why do we listen for data and end events when handling POST?
2. What would happen if we didn’t buffer the body correctly?
3. What is the format of form submissions when using the default browser form POST?
4. Why do we use fs.appendFile instead of fs.writeFile?
5. How could this be improved or made more secure?

Answer:

1. When handling a Post request, the request body is streamed in chunks, meaning it doesn’t arrive all at once.

* The data event captures each chunk of incoming data
* The end event signals when all the data has been received, allowing us to process the full request body.

1. If we didn’t store the chunks of data, the request body would be incomplete or corrupted. This could lead to:

* Partial or missing user input
* Errors when parsing the data
* Security vulnerabilities if handling large payloads improperly.

1. By default, browser forms submit data in URL-encoded format, similar to query parameters in a URL. This format is called application/x-www-form-urlencoded. If the form includes file uploads, it uses multipart/form-data.
2. We use fs.appendFile instead of fs.writeFile because

* Fs.appendFile adds new data to the file without erasing existing content
* Fs.writeFile overwrites the file completely.

Since we want to store multi form submissions, fs.appendFile ensures new data is added instead of replacing previous entries.

1. This can be improved it by:

* Validate user input
* Sanitize data
* Use a database
* HTTPS encryption
* Rate limiting

### Bonus Challenge (Optional)

* Validate that the name field is not empty before saving.
* Send back a small confirmation HTML page instead of plain text.
* Try saving submissions in JSON format instead of plain text.