CADT IDT

W2 PRACTICE

Native HTTP and Manual Routing

EXERCISE 1 – REVIEW and ANALYZE

Goal

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

You are provided with a minimal `server.js` file. Read and run the code. 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 error message is `TypeError: res.endd is not a function`. This occurs because `endd()` is a typo, and it should be `end()`.

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

- `res.write()` sends data to the client, but doesn't finish the response. `res.end()` completes the response and sends the data to the client, closing the connection.

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

- If `res.end()` is not called, the response will not be completed, and the client will not receive a response. The request will hang.

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

- `http.createServer()` creates an HTTP server and allows Node.js to handle incoming requests. It sets up a listening server that runs asynchronously and can handle multiple requests.

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

- The server can be made more resilient by adding error handling (e.g., using try-catch blocks, validating inputs, or using middleware for logging errors).

EXERCISE 2 – MANIPULATE

Goal

✓ Practice using `req.url` and `req.method`. ✓ Understand how manual routing mimics what frameworks (like Express) automate. ✓ 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 vai 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 vai email…
```

TASK 2 – As we can see the complexitiy 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?
   - The server will respond with an empty response or a default error message, as no route is defined for that URL.
2. Why do we check both the `req.url` and `req.method`?
   - We check `req.url` to identify the requested route and `req.method` to ensure the correct HTTP method (GET, POST) is used for that route.
3. What MIME type (`Content-Type`) do you set when returning HTML instead of plain text?
   - The MIME type for HTML is `text/html`.
4. How might this routing logic become harder to manage as routes grow?
   - As the number of routes increases, the conditional logic becomes harder to maintain and read. Managing complex conditions for multiple routes can lead to errors.
5. What benefits might a framework offer to simplify this logic?
   - A framework like Express automates routing, error handling, middleware, and more, making the code cleaner and easier to maintain.

EXERCISE 3 – CREATE

Goal

✓ Practice handling `POST` requests. ✓ Parse URL-encoded form data manually. ✓ Write and append to local files using Node.js' `fs` module. ✓ 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`?
   - We listen for the `data` event to collect the body of the POST request in chunks, and the `end` event to know when the full request body has been received.
2. What would happen if we didn't buffer the body correctly?
   - If we didn't buffer the body, we might not receive the full POST data, leading to incomplete or corrupted form submissions.
3. What is the format of form submissions when using the default browser form POST?
   - The default format is `application/x-www-form-urlencoded`, where form data is encoded as key-value pairs in the body.
4. Why do we use `fs.appendFile` instead of `fs.writeFile`?
   - `fs.appendFile` adds new content to the end of a file, while `fs.writeFile` overwrites the file. We use `appendFile` to preserve previous submissions.
5. How could this be improved or made more secure?
   - To improve security, you could validate the input data (e.g., ensuring no script injection), sanitize form inputs, and implement CSRF protection.

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.