COMP 4537 - Web Computing Diary Book

MILESTONE 1

Bryson Lindy

Overview (Milestone 1)

The assignment asks to collate all our knowledge about web computing and serve it to an audience of a fellow student who, somehow, knows even less than I about HTML, CSS and JavaScript. This document will outline the Digital Ocean droplet I am using to host my labs/assignments for this course as a sort of guide/demonstration of what I've learned about web computing.
The first milestone will showcase the fundamentals we have learned this term regarding node and asynchronous javascript. As the term progresses this web diary will evolve to contain more information from labs/assignments, as well as information from other projects to produce the ultimate compendium of web server knowledge. Or at least a helpful tool to get a node server up and running.

Server Environment

```
4537/
├── server.js
├── modules/
│   ├── routes.js
│   └── utils.js
├── lang/
│   └── en/
│       └── en.js
├── public/
│   └── COMP4537/
│       └── labs/
│           ├── 4/
│           └── 5/
├── .env
├── .gitignore
└── package.json
```

The node server runs on a DigitalOcean Droplet running Ubuntu. The Droplet provides full root access and complete control over the server environment. It's not free but it's pretty cheap and provides lots of user-friendly tools.

Node.js is a JavaScript runtime built on Chrome's V8 engine that lets you run JavaScript outside the browser. It's single-threaded, has non-blocking I/O and is event-driven.

```
// BLOCKING (usually bad)
const data = fs.readFileSync('file.txt');

// NON-BLOCKING (usually good)
const data = await fs.promises.readFile('file.txt');
```

Node's single-threaded nature means blocking operations freeze the entire server. Non-blocking operations allow the server to continue processing other requests while waiting for I/O. We opt to use async calls whenever possible/appropriate to keep things snappy.

```
const options = {
    key: fs.readFileSync('/etc/letsencrypt/live/blindy.net/privkey.pem'),
    cert: fs.readFileSync('/etc/letsencrypt/live/blindy.net/fullchain.pem')
};

const server = https.createServer(options, async (req, res) => {
    // All requests come through here
});
```

To use HTTPS we need to have SSL/TLS (secure sockets layer/transport layer security) certification. We get the certification from Let's Encrypt and renew it automatically with certbot. This is an example of when synchronous code is good for us. We don't want any other requests to be processed until we get our certification in order. Then we build the server with the built in https node module and handle requests.

Core modules used are:
- https => to create the server
- fs => to handle file system operations
- url => parse urls and query strings
- path => system agnostic path handling

Normally we would import more modules like express.js but this course requires us to make our own routing which serves a great way to demonstrate request handling.

Routing

We use a custom Routes class instead of express.

```
  static get(path, handler)
  {
    Routes.routes.GET[path] = handler;
  }

  static post(path, handler)
  {
    Routes.routes.POST[path] = handler;
  }

  static getHandler(path, method = "GET")
  {
    if (method === "POST")
    {
      return Routes.routes.POST[path];
    }
    else // GET
    {
      return Routes.routes.GET[path];
    }
  }
```

Routes are stored as key:value pairs where the key is the path and the value is the corresponding handler function. This lets us register routes with the server and define how they should be handled when requested by the client.

```
Routes.get("/COMP4537/labs/3/getDate/", (req, res, query) => {
    const name = query.name || eng.ANONYMOUS;
    const timestamp = util.getDate();

    res.writeHead(200, { "Content-Type": "text/html" });
    res.end(`<h1 style="color: blue">${eng.GREETING(name, timestamp)}</h1>`);
});
```

Each path needs a handler function to deal with the headers and data. We pass the key parameters:
- req => the request (headers, method, url)
- res => the response (object to send back to the client)
- query => parsed query parameters from the url
- body => POST body if required

When a request arrives the server tries to match it to a registered route.

```
    let body = '';
    if (req.method === 'POST')
    {
        for await (const chunk of req)
        {
            body += chunk.toString();
        }
    }
    let handler = Routes.getHandler(pathname, req.method);

    if (!handler && !pathname.endsWith('/')) {
        handler = Routes.getHandler(pathname + '/', req.method);
    }

    if (handler)
    {
        handler(req, res, query, body);
    }
    else if (await Routes.serveStatic(pathname, res))
    {
        console.log(`${req.method} ${pathname} 200 (static)`)
    }
    else // 404 response
    {
```

Gather the body if it's a POST request. Use the getHandler function to check the registered routes for exact matches with or without a trailing slash. If the handler exists, call it. Otherwise check for static files to be served. Usually with something like express this would be handled with middleware that checks for static files BEFORE the handler, but it's all the same to check afterwards. If the route isn't registered and the request isn't for static files, serve a 404 response.

Static Files

The public directory has all the static assets and client code that need to be served. We have to deal with the MIME type and do some checks to make sure nefarious fellows cannot make dangerous requests.

```
static getMimeType(filePath) {
    const ext = path.extname(filePath).substring(1).toLowerCase();
    const mimeTypes = {
        default: "application/octet-stream",
        html: "text/html; charset=UTF-8",
        js: "application/javascript",
        css: "text/css",
        json: "application/json",
        png: "image/png",
    };

    return mimeTypes[ext] || mimeTypes.default;
}
```

We check the extension of the file and return the corresponding type from a map. There are many more mime types but these are what the server expects to handle.

Before serving a file it’s badass to check for path traversal attacks. I learned this from Mozilla's explanation of how to serve static files without express.

```
  static async prepareFile(url)
  {
    const paths = [Routes.public, url];
    if (url.endsWith("/")) { paths.push("index.html"); }

    const filePath = path.join(...paths);
    const resolvedPath = path.resolve(filePath);

    const pathTraversal = !resolvedPath.startsWith(Routes.public);
    const exists = await fs.promises.access(resolvedPath).then(...Routes.toBool);
    const isFile = exists && (await fs.promises.stat(resolvedPath)).isFile();

    const found = !pathTraversal && exists && isFile;
    const streamPath = found ? resolvedPath : null;
    const ext = streamPath ? path.extname(streamPath).substring(1).toLowerCase() : null;

    return { found, ext, streamPath };
  }
```

If the requested path doesn’t begin with public, then it’s not found. Always have to be vigilant to protect data against these kinds of attacks.

Once a file is validated, serve it up to the client.

```
static async serveStatic(pathname, res) {
    if (!Routes.public) { return false; }

    const file = await Routes.prepareFile(pathname);

    if (!file.found) { return false; }

    const mimeType = Routes.getMimeType(file.streamPath);
    const stream = fs.createReadStream(file.streamPath);

    res.writeHead(200, { "Content-Type": mimeType });
    stream.pipe(res);

    return true;
}
```

Streaming is a great choice because it allows large files to be served and begins sending immediately. The whole process here uses async calls to access and check files so nothing blocks.

Asynchronous JavaScript

Async code is important for node to deal with the single thread.

```
Routes.get("/COMP4537/labs/3/writeFile/", async (req, res, query) => {
    const text = query.text;

    if (!text) {
        res.writeHead(218, { "Content-Type": "text/html" });
        res.end(`<h1 style="color: red">${eng.TEXT_MISSING}</h1>`);
        return;
    }

    const dirPath = path.join(__dirname, "..", "data");
    const filePath = path.join(dirPath, "file.txt");

    try {
        await fs.promises.mkdir(dirPath, { recursive: true });
        await fs.promises.appendFile(filePath, text + "\n");
        res.writeHead(200, { "Content-Type": "text/html" });
        res.end(`<h1 style="color: green">${eng.WRITE_SUCCESS(text)}</h1>`);
    }
    catch (err) {
        res.writeHead(500, { "Content-Type": "text/html" });
        res.end(`<h1 style="color: red">${eng.WRITE_ERROR(err)}</h1>`);
    }
});
```

The async keyword makes the code run asynchronously (believe it or not), and await pauses execution until the promise resolves. Once paused, the event loop is free to process other requests. Try/catch blocks handle rejected promises and synchronous errors.

Could also use promise chaining

```
  static toBool = [() => true, () => false];
  static async prepareFile(url)
  {
    const paths = [Routes.public, url];
    if (url.endsWith("/")) { paths.push("index.html"); }

    const filePath = path.join(...paths);
    const resolvedPath = path.resolve(filePath);

    const pathTraversal = !resolvedPath.startsWith(Routes.public);
    const exists = await fs.promises.access(resolvedPath).then(...Routes.toBool);
    const isFile = exists && (await fs.promises.stat(resolvedPath)).isFile();

    const found = !pathTraversal && exists && isFile;
    const streamPath = found ? resolvedPath : null;
    const ext = streamPath ? path.extname(streamPath).substring(1).toLowerCase() : null;

    return { found, ext, streamPath };
  }

  static async serveStatic(pathname, res)
  {
    if (!Routes.public) { return false; }

    const file = await Routes.prepareFile(pathname);

    if (!file.found) { return false; }

    const mimeType = Routes.getMimeType(file.streamPath);
    const stream = fs.createReadStream(file.streamPath);

    res.writeHead(200, { "Content-Type": mimeType });
    stream.pipe(res);

    return true;
  }
}
```
This pattern is stolen from Mozilla from the same static file serving link as above. fs.promises.access() throws an error if a file doesn't exist, so .then(...Routes.toBool) spreads the array into the format promises expects. Success will be true, failure is false. This makes the process of checking files nice and clean by providing boolean variables for subsequent checks.

Security

The SSL/TLS encryption mentioned in the server environment is crucial security to make sure we can use HTTPS. The client requests the domain, the server sends the public certificate (fullchain.pem). The client uses the public cert to generate a session key which gets encrypted by the public certificate. Only the private key that the server has access to (privkey.pem) can decrypt the session key. This ensures that devious man-in-the-middle attackers get encrypted gibberish instead of actual data.

Try/catch error handling makes sure the server doesn't expose file structure information on failures.
```
try {
    await fs.promises.mkdir(dirPath, { recursive: true });
    await fs.promises.appendFile(filePath, text + "\n");
    res.writeHead(200, { "Content-Type": "text/html" });
    res.end(`<h1 style="color: green">${eng.WRITE_SUCCESS(text)}</h1>`);
}
catch (err) {
    res.writeHead(500, { "Content-Type": "text/html" });
    res.end(`<h1 style="color: red">${eng.WRITE_ERROR(err)}</h1>`);
}
```
We control the error messaging instead of exposing server internals.

Input is validated whether it is the query parameters or the raw SQL we were asked to send for lab 5. It's important to sanitize and validate data to prevent elite hackers from gaining access to the server.

Deployment

Without CI/CD, it can be very tedious to keep a server updated during development. So use GitHub actions because they are amazing.

```yaml
name: Deploy to Server
on:
  push:
    branches: [ main ]

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v5

    - name: deploy to server
      uses: appleboy/ssh-action@v0.1.7
      with:
        host: ${{ secrets.SERVER_HOST }}
        username: ${{ secrets.SERVER_USER }}
        key: ${{ secrets.SSH_PRIVATE_KEY }}
        script: |
          cd ~/COMP4537/labs/3/4537-lab3
          git pull https://${{ secrets.PAT }}@github.com/brysonomicon/4537-lab3
          pm2 restart lab3-server
```

When code is pushed to the main branch, we spin up a virtual machine and use it to ssh into the server automatically. We set some keys in the github environment and execute a simple script to pull changes and restart the server. This way when we push any changes to github, the server updates immediately. Little bit of overhead saves you having to ssh into the server to manually pull and update.