Server-side Introduction in Node.js

Today’s Goals

- Setting up a Node.js project from scratch
- Writing an HTTP Server in Node.js using:
  - Node.js’s built-in HTTP module
  - A minimalist framework such as Express.js
- An introduction to server-side rendering
- Handling HTML <form> POST data

Setting up a TypeScript Server-side Project from Scratch

- Setup a directory for your project:
  - mkdir 10-node-from-scratch
  - cd 10-node-from-scratch
- Initialize a package.json file:
  - npm init -y
- Ignore the node_modules folder (where node installs libraries) in your repo:
  - echo “node_modules” >> .gitignore
- Install development dependencies (these are the libraries our project needs):
  - TypeScript, ts-node (runs TypeScript without precompiling), TypeScript Node.js Type Definitions
  - npm install --save-dev typescript ts-node @types/node
- Add a "start" script to npm:
  - "start": "ts-node index.ts"
- Add an "index.ts" file to the project with the contents of: `console.log("Hello, world");`
- Try running the project's start script! **npm run start**

Our first Server-side Application

```
import { createServer } from 'http';

let server = createServer((request, response) => {
    response.statusCode = 200;
    response.setHeader("Content-Type", "text/text");
    response.write("Hello, world");
    response.end();
});

server.listen(1234, () => console.log("Listening on 1234"))
      .on("error", (e) => console.error(e))
```

Let's Add Some Example Resources

- /random - generate a random number
- /json - respond in content-type application/json
- /redirect - return a 302 redirect to location /json
- /not-found - respond with a 404 error

Using Node's HTTP Library Directly is Uncommon

- It imposes no structure on your server's application design
  - The intent of Node's built-in HTTP library is to provide "low-level" primitives
- After 20 years of back-end development, common needs identified:
  - Routing requests tends to be organized by *resource* (URL)
  - Per resource, HTTP methods have different outcomes (GET vs POST)
  - There are cross-cutting concerns you'd like to share across handlers
    - Such as user identification, logging of requests, and so on, "middleware"
- Framework's structure address common needs so you don't reinvent the wheel

Adding Express Framework

- The Express framework is one of Node's most popular on the server-side
  - We're choosing it because it's minimal and learning its structure translates well to popular frameworks in many other languages:
  - Ruby: Sinatra, Rails (Batteries included)
  - PHP: Silex, Slim
  - Python: Flask
- To add it to your project we need to install it as a full dependency:
  - npm install --save express
- Since we're developing in TypeScript, we'll also need to install its types:
  - npm install --save-dev @types/express

Our first Express Application

```
import * as express from "express";

let app = express();

app.get("/", (req, res) => {
    res.send("Hello, world!!!");
});

app.listen(1234, () => console.log("Listining on port 1234"))
   .on('error', (e) => console.error(e));
```

Try Adding Some Routes

- For now, these will all bet routes accessible with the GET method:
- /time - Respond with "The current time is " + new Date()
- /redirect - Respond by 302 redirecting to "/time"
    - Search for how to redirect in Express
- /hits - Declare a global variable named hitCounter and initialize it to 0.
    - Each time /hits is accessed, increment the hitCounter variable by 1 and respond with the string `The current hit count is ${hitCounter}`

Let's Add Middleware

- What if for *every* incoming request we want to:
  - Log its method and URL
  - Update the hitCounter variable by 1
- We can *use* a middleware function to achieve these *cross-cutting* concerns
- Generally, middleware is used to abstract out common pre- or post-processing steps to requests/responses across *many* or *all* routes.

Simple Middleware

```
app.use((req, res, next) => {
    console.log(`${req.method} ${req.url}`);
    hitCounter += 1;
    next();
});
```

- A middleware function is registered *before* routes and makes use of a third parameter named *next.*
- The *next* callback is a function that tells Express: "Pass these request/ response objects on along to the next middleware/route. I did not handle it."

Adding a Template Engine

- If we want to respond with HTML from our back-end, which is common, it is best practice to use an HTML template engine rather than building up HTML response strings manually.
- There are a *ton* of HTML Template Libraries
- We'll choose Handlebars because it's reasonably simple
- To add it to our project we have two production dependencies:
  - npm install --save handlebars express-handlebars
- And one development dependency:
  - npm install --save-dev @types/handlebars
- Setup directories for views, views/layouts, views/partials
- Register view engine: https://www.npmjs.com/package/express-handlebars