REST Exercises

Setup

1. Use starter pack provided for this course and unzip it.
2. Create your own project folder (`<student-id>-rest-api`) and copy over all the unzipped files (`README.md` optional). Make sure the 2 'hidden' files like `.editorconfig` and `.gitignore` are also copied over.
3. Download the following:
    - Postman to quickly send REST API requests and save them in a collection for testing.
    - JSONView Chrome Extension to better view your JSON responses in the browser.

Goal

In these exercises, we will learn how to create a REST API using Express. We will also generate documentation for our API using Swagger.

There are a few approaches:

1. Write the API, create a `swagger.json` file that will be used by the Swagger UI.
2. Write the API, add documentation in the code to auto-generate the Swagger docs & UI.

We will be using the first approach, but a section below will also provide some detail on how the other approach can be done.

Endpoints

At the end of this exercise, we should have the following endpoints:

- GET /doctors
- GET /doctors/:id
- POST /doctors
- GET /patients
- GET /patients/:id
- POST /doctors
- GET /visits (with query parameters)

Exercises

Hello World Basic Setup

Goal: Set up Express server with a Hello World example. Set up middleware and data.

First, let's get an Express server up and running. Express is a a minimal and flexible Node.js web application framework that will be serving up our REST endpoints.

1. In your local folder set up from the previous section, run `npm init -y` (defaults) to set up our `package.json` file.

2. Install the packages we need by running the following in your terminal:

    ```
    npm install esm express cors body-parser
    ```

    - `body-parser` : Documentation. Parses the HTTP request body, making it easier to access `request.body` in routes.

    - `cors` : Documentation. Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional HTTP headers to tell a browser to let a web application running at one origin (domain) have permission to access selected resources from a server at a different origin. A web application makes a cross-origin HTTP request when it requests a resource that has a different origin (domain, protocol, and port) than its own origin. To avoid running into cors issues when trying to call your REST service from the React UI we'll be creating later, ensure that your service allows cors.

    These 2 middleware will essentially run before the request gets to our endpoints. We also need:

    - `esm` : Documentation. This package allows us to use ES6 syntax in our code for importing and exporting modules.

    - `express` : Documentation. This is the framework we're using for our REST API.

3. In your folder (created from the Local Setup section), create a folder `src` , then within the folder create a file called `index.js` . Add the following code (comments are added for explanation). This snippet was based on Express documentation in Getting Started.

    ```
    import express from "express"; // importing the module
    const app = express(); // creating an Express app
    const { PORT = 3000 } = process.env;

    // set up route for '/', http://expressjs.com/en/5x/api.html#res.send
    ```

```
// this will show up on `localhost:3000` in the browser
app.get("/", (request, response) => response.send("Hello World!"));

// server will start listening for requests, the function is called immediately
once the server is ready. Console.logs show up in your terminal.
app.listen(PORT, () =>
  console.log(`Hello World, I'm listening on port ${PORT}!`)
);
```

4. In `package.json`, add your start script

```
"start": "node -r esm src/index.js"
```

Alternatively, you can install `nodemon` to watch for changes and avoid having to stop and start the server every time you make changes. Add a script to run in development that runs nodemon.

```
npm install nodemon
```

```
"dev": "nodemon -r esm src/index.js"
```

5. Run your app and navigate to `localhost:3000`. You should see your first "Hello World" example.

Adding middleware

1. Our middleware (body-parser and cors) should run before the request goes to our endpoints. To set this up, add the following before the `app.get()`. You can read more about using middlewares with `app.use` in the documentation.

```
import bodyParser from "body-parser";

import cors from "cors";

app.use(bodyParser.json()).use(cors());

app.get("/", (request, response) => response.send("Hello World!"));
```

2. In the next exercises, you can try commenting out the `app.use` line to see how your routes are affected.

Set up data as a module

1. In `data.js`, assign the object to a variable called data. Export the variable.

```
const data = { ... }
export default data;
```

2. In `src/index.js`, import the data module at the top.

```
import data from "../data"; // the `./` denotes the folder structure
```

Get a list of doctors

Goal: Set up endpoint to retrieve a list of doctors through `GET /doctors`.

Let's set up an endpoint to retrieve a list of doctors `GET /doctors`. Here is the documentation for the methods we're using:

- Response.json()
- App.get().

It's convention to prefix routes with `/api/v1` to indicate the version.

In `src/index.js`, add the following after the first `app.get` line:

```
app.get("/api/v1/doctors", (req, res) => res.json(data.doctors));
```

Navigate to your browser and go to `localhost:3000/api/v1/doctors` to see a list of doctors.

Get a doctor by id

Goal: Set up endpoint to retrieve a list of doctors through `GET /doctors/:id`.

Let's set up an endpoint to retrieve a doctor by id `GET /doctors/:id`. We're using the same methods as above, but now we can have access to the request parameters (id).

- Request.params

1. Set up the route. Parameters are denoted with a `:` prefix. Let's try logging what's in `req.params`.

   ```
   app.get("/api/v1/doctors/:id", (req, res) => {
     console.log(req.params); // returns an object with { id: '1' }
   });
   ```

2. Find the doctor from the `data` object and return it. We'll implement error handling later, follow the happy path for now. Note that the params passed are of String type, while our data has Number ids.

   ```
   app.get("/api/v1/doctors/:id", (req, res) => {
     const id = parseInt(req.params.id);
     const doctor = data.doctors.find((doc) => doc.id === id);
     return res.json(doctor);
   });
   ```

3. Go to your browser and test out the route for `localhost:3000/api/v1/doctors/1`. Try another id.

Using Postman

Goal: Use Postman to test your endpoints.

So far, we've been going to the browser to test the endpoints. We can also use Postman (and we will need it for POST requests in later sections).

1. Open up Postman. You should not need to sign in or create an account.
2. Create a New Collection, name it **REST API Exercises**.
3. Click on New -> New Request. Let's do the first GET /doctors request, so fill in the form with the request name, a description, and make sure it saves to the correct collection.
4. Choose the `GET` request through the dropdown and fill in the URL. Click **Send** to send the request. You should see the response JSON.

5. Create another request for the `GET /doctors/id` request using the same steps above.

Adding a doctor

Goal: Set up endpoint to add a doctor through `POST /doctors`.

Let's set up an endpoint to add a doctor to our list. We're using these methods:

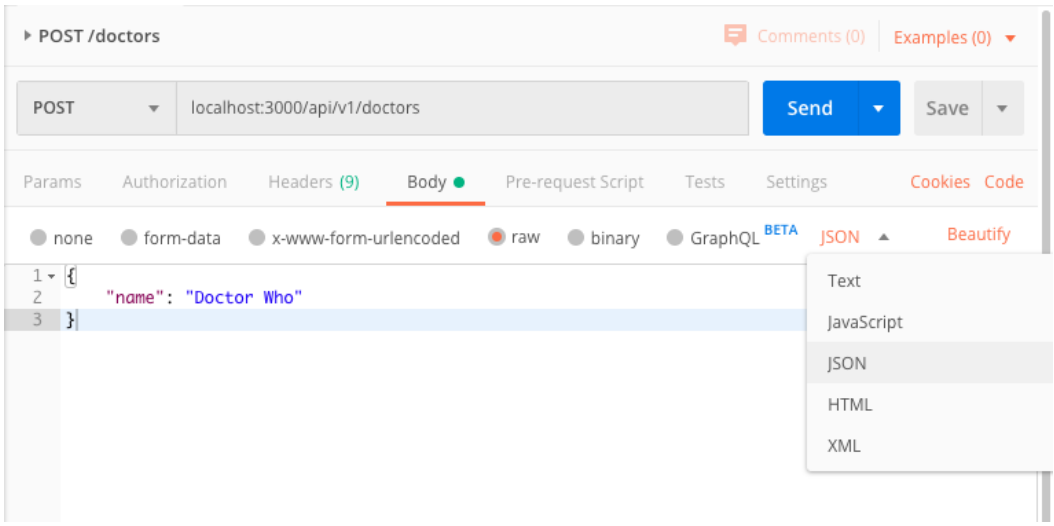
- App.post()
- Request.body
- Response.status()
- HTTP Status Codes

1. In `src/index.js`, add the POST request. We are adding the doctor details into our `data.doctors` list and assigning it an id.

```
app.post("/api/v1/doctors", (req, res) => {
  const nextId = data.doctors.length + 1;
  const doctor = { id: nextId, ...req.body };

  data.doctors.push(doctor);
  res.status(201).json(doctor); // 201 means Resource Created
});
```

2. In Postman, set up a new POST request for the endpoint. Click on the Body tag (this is where your doctor information will be sent), click on the `raw` selection and click on the dropdown to select JSON. The screenshot below shows how it should be set up. Add a JSON object in the textarea with a `name` property.



3. Send the request, then check your doctors list with a GET request again. You should see the newly added doctor at the end of the list.

Error Handling and Parameter Validation

Goal: Add parameter validation for getting a doctor by ID and adding a doctor.

We implemented the happy paths for our routes, but what could go wrong?

- In GET /doctors/:id, the id passed in the route could be letters
- In GET /doctors/:id , the id doesn't exist in `data`
- In POST /doctors, we don't check what details we're allowing.
- When there is an error, what should the response send back?

Adding validation for the doctor id

Let's update the code to check if the id param sent in the request is a valid id. We're going to make the assumption that ids are numbers.

1. In your route for `app.get("/api/v1/doctors/:id")` , add a function to check for an invalid id. We're using the following methods:

   - Number.isNaN()

   We're keeping the function inside the route for now to make the code-along easy to follow (everything in one file), but for separation of responsibilities, you may want to extract this logic to a Validator object.

   In src folder, create a subfolder `utils` , then create a file `functions.js`

   ```
   export const isInvalidId = (id) => {
     return Number.isNaN(parseInt(id, 10));
   };
   ```

2. In src/index.js, use the validator function, to check if it's an invalid ID. If not, return an error. We're using the following methods:

   - Response.status()
   - HTTP Status Codes

   ```
   import * as utilities from "./utils/functions";
   ....
   if (utilities.isInvalidId(req.params.id)) {
     return res.status(400).json({ error: "Invalid id." });
   }
   ```

3. We also want to check if the doctor with that id exists in the data. If not, return an error.

   ```
   const id = parseInt(req.params.id, 10);
   const doctor = data.doctors.find((doctor) => doctor.id === id);

   if (!doctor) {
     return res.status(404).json({ error: "Doctor not found." });
   }
   ```

4. The route should now look like this:

   ```
   app.get("/api/v1/doctors/:id", (req, res) => {
     if (utilities.isInvalidId(req.params.id)) {
       return res.status(400).json({ error: "Invalid id." });
     }
     const id = parseInt(req.params.id);
     const doctor = data.doctors.find((doc) => doc.id === id);
     if (!doctor) {
       return res.status(404).json({ error: "Doctor not found." });
   ```

```
  }

  return res.json(doctor);
});
```

Allowing specific properties for POST /doctors

Currently, our POST route is adding everything passed through the request body. We want to guard against bad data, as well as make sure the required information is there (such as the doctor's name).

1. Check for the existence of name in the body. If it doesn't exist, or it's empty, return an error.

```
if (!req.body.name) {
  return res.status(400).json({ error: "Doctor needs a name parameter." });
}
```

2. Replace the spread operator `...req.body` with the properties you're allowing.

```
const doctor = { id: nextId, name: req.body.name };
```