Creating a simple REST API using ***Fastify***

this is just an excuse to experiment with LaTeX and Shiki so please, don't expect anything :p

Dicha Zelianivan Arkana

June 9, 2021

Table of Contents

1 Prerequisites

- Basic understanding of Javascript.
- Text Editor.
- Browser.

2 Setup

2.1 Project Folder

First thing first, let's create a simple project folder.

```
mkdir fastify-todo-api && cd $_ && npm init -y
```

This will create a directory named `fastify-todo-api`, change the CWD to that folder, and initialise a fresh node project.

2.2 Package Installation

After setting up the project folder, we need to install the fastify package itself. Simply run this command to install it:

```
npm install fastify # 'npm i fastify' also works
```

Open your preferred text editor and edit a file called `index.js` and import the fastify package.

```
const fastify = require("fastify");
```

Now, create a new app instance by invoking fastify. Let's set the `logger` to `true` to enable logging.

```
const fastify = require("fastify");

const app = fastify({ logger: true });
```

After creating an app instance, we need to start the application and make it listen to port 3000. Of course you're free to pick which port to use.

```
const fastify = require("fastify");

const app = fastify({ logger: true });
const PORT = 3000;

(async () => {
  try {
    await app.listen(PORT);
    console.log(`App is running on: http://localhost:${PORT}`);
  } catch (err) {
    app.log.error(err);
    process.exit(1);
  }
})();
```

We can also use `.then` and `.catch` syntax as such:

```
const fastify = require("fastify");

const app = fastify({ logger: true });
const PORT = 3000;

app
  .listen(PORT)
  .then(() => {
    console.log(`App is running on: http://localhost:${PORT}`)
  })
  .catch((err) => {
    app.log.error(err);
    process.exit(1);
  });
```

..but for now let's stick to async-await syntax. To run this code, simply run `node index.js` and you should see an output similar to this on your terminal:

```
{"level":30,"time":1623233414708,"pid":172047,"hostname":"arch","msg":"Server listening
↪  at http://127.0.0.1:3000"}
App is running on: http://localhost:3000
```

2.3 Automatic Refresh with Nodemon

To make the development process easier, we can also install `nodemon`. We can install it using `npm` just like any other package:

```
npm install --save-dev nodemon # 'npm i -D nodemon' also works
```

We can then run `nodemon index.js` and it will get refreshed whenever the file is changed. The output when you run it should now look similar to this:

```
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
{"level":30,"time":1623233491233,"pid":172559,"hostname":"arch","msg":"Server listening
↪  at http://127.0.0.1:3000"}
App is running on: http://localhost:3000
```

2.4 package.json Scripts

Instead of keep running `node index.js` or `nodemon index.js`, we can create a script inside our `package.json`.

Open `package.json` and it should look something like this:

```
{
  "name": "fastify-todo-api",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "fastify": "^3.17.0"
  }
}
```

We need to edit the `"scripts"` section by removing the `"test"` script and adding `"start"` and `"dev"`. It should now look like this:

```
{
  // ...rest of the file
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js"
  },
  // ...rest of the file
}
```

After adding those, we can now run `npm start` to use node or `npm run dev` to use nodemon.

3 Routes

Each routes has its own HTTP methods. There are several ways to define routes but let's use the sorthand that fastify has provided.

3.1 GET

3.1.1 Hello World!

For our first route, let's make a GET route that simply says "Hello World". Place it before the app initialisation.

```
const fastify = require("fastify");

const app = fastify({ logger: true });
const PORT = 3000;

app.get("/", (_req, _reply) => {
  return "Hello, World!";
});

(async () => {
  try {
    await app.listen(PORT);
    console.log(`App is running on: http://localhost:${PORT}`);
  } catch (err) {
    app.log.error(err);
    process.exit(1);
  }
})();
```

I prefixed the parameter name with underscore `_` because it's not being used, of course you can omit them if they're not being used.

We can try accessing this endpoint by running `curl` like so:

```
curl http://localhost:3000
```

You'll see the output as `Hello World!` after running the command. Great! we've made our first route.

3.1.2 Getting All TODOs

Let's make an `/api/todos` route to get all of our todos. I will remove the irrelevant code for the code snippet below, but as I said before, place the route *before* the app initialisation.

```
// we'll save our todos in this array, ideally we should save it in
// a database but we'll do this for the sake of simplicity
let todos = [
  {
    id: 1,
    name: "Learn LaTeX",
    isCompleted: false,
  },
  {
    id: 2,
    name: "Learn Rust",
    isCompleted: false,
  },
];

app.get("/api/todos", (_req, _reply) => {
  return {
    status: 200,
    msg: "Successfully retrieved all todos.",
    data: todos,
  };
});
```

We can also try to access this endpoint using `curl`.

```
curl http://localhost:3000/api/todos
```

The output would look something like this:

```
{"status":200,"msg":"Successfully retrieved all todos.","data":[{"id":1,"name":"Learn
↪  LaTeX","isCompleted":false},{"id":2,"name":"Learn Rust","isCompleted":false}]}
```

Not the prettiest output, but hey, it's the correct output! You can use tools like `jq` for example to make it prettier. Running the following command will give you an easier to read output.

```
curl -s http://localhost:3000/api/todos | jq
```

Here's how the output from the previous command will look like:

```
{
  "status": 200,
  "msg": "Successfully retrieved all todos.",
  "data": [
    {
      "id": 1,
      "name": "Learn LaTeX",
      "isCompleted": false
    },
    {
      "id": 2,
      "name": "Learn Rust",
      "isCompleted": false
    }
  ]
}
```

3.1.3 Getting a Single TODO

We can also get a specific todo by creating an endpoint which will receive an ID and return a todo item with that specific ID. Don't forget to put this route **above** the previous route because otherwise this wouldn't work.

```
app.get("/api/todos/:id", (req, _reply) => {
  const { id } = req.params;
  const todo = todos.filter(t => t.id === +id);
  return {
    status: 200,
    msg: "Successfully retrieved a todo with an id of " + id,
    data: todo,
  };
});
```

Using a colon and giving it a name like `:id` will allow us to access the value of that URL. In this case, if we go to `http://localhost:3000/api/todos/1` then we'll have access to `req.params.id` which is going to be `1`.

Since the value is going to be a `string`, when we compare it with the `id` of our todos using triple equal sign (called "strict equal") which means it also checks the data type, it will always return `false`. Here, we use `+id` to cast the datatype to `number`. We can also use `parseInt()` but using a plus symbol is shorter and looks nicer.

3.2 POST

3.2.1 Creating TODO

We can get all of our todos but we can't create one, yet. Let's add a route to add a todo into our app.

```
app.post("/api/todos", (req, _reply) => {
  const { name } = req.body;

  // generate a random ID and add the new todo to the list
  const id = Math.floor(Math.random() * 100);
  const item = { id, name, isCompleted: false };
  todos.push(item);

  return {
    status: 201,
    msg: "Successfully created a todo with an id of " + id,
    data: [],
  };
});
```

Here, we use a randomised number from 1 - 100. In a real world you'd use a better way like using `nanoid` or `uuid`, but for the sake of simplicity, we'll use this super simple method instead.

To check if this endpoint is working, we can use this command to check if it works or not.

```
curl -X POST -H "Content-Type: application/json" -d '{"name": "stuff"}' -s
↪  http://localhost:3000/api/todos | jq
```

The output is going to look like this:

```
{
  "status": 201,
  "msg": "Successfully created a todo with an id of 36",
  "data": []
}
```

You can omit piping it to `jq` if you don't need a pretty-formatted output.

3.3 PUT

3.3.1 Updating TODO

If we want to update one of our TODOs, we'll need a route for a `PUT` method. Creating it is quite, the code is as follow.

```
app.put("/api/todos/:id", (req, _reply) => {
  const { id } = req.params;
  const { name, isCompleted } = req.body;

  // find the todo to modify
  const old = todos.find(t => t.id === +id);

  // create a new todo item with fallback value if no new value is provided
  // it's using a nullish coalescing operator to check if the lhs value is
  // null, then use the rhs value
  const item = {
    id: old.id,
    name: name ?? old.name,
    isCompleted: isCompleted ?? old.isCompleted,
  };

  // replace the old items with the new one with modified field(s)
  todos = todos.filter(t => t.id !== +id).concat(item);

  return {
    status: 200,
    msg: "Successfully updated a todo with an id of " + id,
    data: [],
  };
});
```

You can use this `curl` command to update one of our existing TODOs.

```
curl -X PUT -H "Content-Type: application/json" -d '{"name": "asdasdasdasd",
↪  "isCompleted": true}' -s localhost:3000/api/todos/1 | jq
```

Here we update a TODO with an ID of 1, change its `name` to `asdasdasdasd`, and `isCompleted` to `true`. The response from that command will look like this.

```
{
  "status": 200,
  "msg": "Successfully updated a todo with an id of 1",
  "data": []
}
```

3.4 DELETE

3.4.1 Deleting a TODO

To delete an item, we would need a route that accepts a `DELETE` request. It simply sets the `todos` array that we have before to a new array that doesn't have the targetted item. Here's how the code looks:

```
app.delete("/api/todos/:id", (req, _reply) => {
  const { id } = req.params;

  // replace the old items *without* the deleted item
  todos = todos.filter(t => t.id !== +id);

  return {
    status: 200,
    msg: "Successfully deleted a todo with an id of " + id,
    data: [],
  };
});
```

Try deleting a TODO using `curl`, you can use this command to test it.

```
curl -X DELETE -s localhost:3000/api/todos/1 | jq
```

The response would look like this:

```
{
  "status": 200,
  "msg": "Successfully deleted a todo with an id of 1",
  "data": []
}
```

4 Error Handling

If you noticed, we haven't handle any edge cases that might cause an error. In a real world situation, you ***should*** handle edge cases for your app. I'd leave it up to you to write since my initial intention to write this is just to try out LaTeX and Shiki.

You can find the original code inside `fastify-todo-api` directory in this repository.

Good Luck! :)