Northcoders

back to passport

< home < js-back-end

Express 101

Prior Knowledge

Awareness of and ability to use callback functions

Anatomy of a URL

Understanding of what a server is, and awareness of `http.createServer`

HTTP request methods

Basic HTTP status codes

Learning Objectives

Be able to create an express server

Be able to use basic express setup and best-practices for file naming e.g app.js and index.js
also brief introduction to nodemon and package.json scripts

Understand what a **parametric endpoint** is

Extract information from urls using `req.params` and `req.query`

Understand what `express.json()` does

What is REST?

Revision on what a restful api structure is:

BAD: **/fetch-ponies** and **/add-ponies** *GOOD*: **GET /ponies** and **POST /ponies**

The idea behind RESTful routing is that by just making `/ponies` available as an endpoint, depending on the HTTP Method (verb) that is used, the behaviour and response provided will be different.

So instead of having `/fetch-ponies` and `/add-ponies`, `/ponies` can use both a `GET` and a `POST` method on the same route.

Have a read of this article on freecodecamp has some further explanations.

Introducing Express

Express JS describes itself as a:

...minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

In simple terms, it does everything that HTTP Servers from `node` do, but it's a lot nicer to work with.

Here is a simple example of setting up a server using `http`:

```
// server.js
const http = require('http');
const server = http.createServer((req, res) => {});
server.listen(9090, () => {
  console.log(`listening on 9090...`);
});
```

We can use the following boilerplate code to setup our express app, which will be an instance of the express library:

```
/// app.js
const express = require('express');
const app = express();
```

Notice that we're not going to be passing a function with `req` and `res` to our server here. That is part of what express does - by default, it will pass many of these arguments for us.

In order to ensure that our http server listens for incoming requests from the client side, invoke the server's `listen` method (equivalent to `http.createServer` 's listen method).

> Specify the port you wish to listen on as the first argument.
>
> Pass an optional callback function as the second argument. This is useful for providing console feedback that the server is successfully listening.

```
// server.js
server.listen(9090);
```

In `express` , the listen method is equivalent to the listen method used in `http.createServer` .

```
// listen.js
app.listen(9090, () => {
  console.log(`Server is listening on port 9090...`);
});
```

The listen method binds the app to the relevant port so it can listen for requests on, in this example, port 9090.

NB: separate listen functionality to a separate file. Testing packages for our back-end apps will establish their own listeners - this will not work if listening happens as part of the app definition.

> ***In order to test our back-end apps later on, we will need to extract this functionality to a separate file, e.g.*** *`listen.js` **, by exporting*** *`app`* ***from*** *`app.js`* ***and requiring*** *`app`* ***in*** *`listen.js`* **.**

Basic Express Endpoint

Express has methods that will mean specific functions that we write can be invoked when we match a particular HTTP method and route combination.

```
const express = require('express');
const app = express();

app.get('/', (req, res) => res.send('Hello World!'));

app.listen(9090, () => console.log('App listening on port 9090!'));
```

The first argument passed to the express `.get()` method is the path `'/'` .

Any time we get a `GET` request on the `'/'` route, express will invoke the second argument (which in the above example is an anonymous function).

This function is called a **middleware** function. Middleware functions are functions that have access to the request object (`req`) and the response object (`res`) in the application's request-response cycle.

Express hello world example

Basic Express Routing

With `http.createServer` routing required the following kind of logic:

```
if (req.url === '/cats') {
  if (req.method === 'GET') {
    // JS code in here...
  }
}
```

In express, the same can be achieved with the following:

```
const express = require('express');
const app = express();

app.get('/cats', () => {
  // JS code in here...
});
```

Any time we get a `GET` request on the `'/cats'` route, express will invoke the second argument (which in the above example is an empty, anonymous function).

This function will have access to the `request` and `response` objects (`req` and `res`) as its parameters. Express is responsible for invoking this callback function when are request is made to the matching endpoint with the specified http method.

Sending our response

Previously, using `http.createServer` we would need to build the response manually before sending it with `response.end()`

```
if (url === '/' && method === 'GET') {
  response.setHeader('Content-Type', 'application/json');
  response.statusCode = 200;
  response.write(JSON.stringify({ msg: 'Server up and running' }));
  response.end();
}
```

The above method will update the headers in the request objects in order to inform the browser that it is being sent json. The status code is updated to 200 for a successful response. An actual message is written and the response sent back to the client.

In express, we can be far more succinct and achieve the same with the following:

```
app.get('/', (req, res) => {
  res.status(200).send({ msg: 'Welcome' });
});
```

The status code is set with the `.status` method of the response object.

The `.send` method of the response object automatically sets the `Content-Type` header to `text/html` if passed a string, and to `application/json` if passed a JS object.

`res.send` also deals with ending the response, and will send a status code of `200` by default.

`res.json` can be used to explicitly set the headers so that `Content-Type` is `application/json` . It uses `JSON.stringify()` in the background so we do not have to stringify before passing data into the `.send()` or `.json()` methods.

Parametric end-points (variables)

`http.createServer` requires the use of conditional logic, and awkward parsing/testing of url paths, sometimes with regular expressions. This is especially true when it comes to variable endpoints such as `GET /api/cats/:catId` .

```
if (req.url.includes('/cats/') {
  if (req.method === 'GET') {
    const catId = req.url.split('/cats/')[1];
    // ...
  }
}
```

Express allows us to pass in a variable value into our path marked by the `":"` - it will combine all of these **parameters** into a params object, and place it on the request. The key will be the phrase that follows the colon; the value will be the section of the path that Express actually received in the request.

```
app.get('/cats/:catId', (req, res) => {
  console.log(req.params); // { catId: value-from-the-request-url-here }
});
```

The value given to the variable in the path will be the key in our `params` object and the actual variable username the client has sent in the url will be the value.

```
app.get('/cats/:catId', (req, res) => {
  const { catId } = req.params;
  fs.readFile(`data/cats/${catId}.json`, 'utf8', (err, catString) => {
    if (err) console.log(err);
    else {
      const parsedCat = JSON.parse(catString);
      res.status(200).send({ cat: parsedCat });
    }
  });
});
```

Queries

Express also makes handling queries much more straightforward than it would be using `http.createServer`.

Unlike `http.createServer` where we would have had to parse the different sections of the query string ourselves, queries are made available by express on the request object (`req.query`) and do not require any additional input or changes to the express route set-up in `app.js`.

For example a request to `GET /api/cats?grumpy=true` could be handled as follows:

```
// app.js
app.get('/cats', (req, res) => {
  const { grumpy } = req.query;
  // we would then get our grumpy or not grumpy cats from the data
  // and then send the response
});
```

The `req.query` object is built up from the key-value pairs defined in the query string. Any additional queries will be added to the object as key-value pairs.

Dealing with the body

Express is a "**minimal** and flexible Node.js web application framework" and will not automatically handle the parsing of the request body for us, as Express cannot ***assume*** the data type that is being received on the body of a request.

With `http.createServer` streams/packets of data have to be handled individually to assemble the request body.

```
const addCat = (req, res) => {
  let body = '';
  req.on('data', (d) => (body += `${d}`));
```

```
  req.on('end', () => {
    // do something with the body
    // ...
  });
})
```

Express was designed with minimal functionality, stripping out anything that was not crucial to its use. That includes, by default, the ability to parse the body of the HTTP request.

The following will console.log `undefined`:

```
// app.js
app.post('/cats', (req, res) => {
  console.log(req.body);
});
```

In order to access the body of the request that is made to the sever, we need to use **application-level middleware**.

Application-level middleware can be thought of as additional function that is invoked before the request and response objects are passed on to the controllers.

`express.json()` is a built in middleware function that will parse incoming request JSON bodies, making them available under the `req.body` property in the controller functions.

`express.json()` is used as middleware by the express app with the following lines of code:

```
// app.js
const express = require('express');
const app = express();

app.use(express.json());

app.post('/', (req, res) => {
  console.log(req.body);
});
```

Sending a body of the following in a POST request using Insomnia would attach this object to the `body` key of the request object (`req.body`), which could then be used to create the new item in the dataset.

Running the server & nodemon

It is tedious and unnecessary to have to kill and restart the listen process each time we save changes to the application. Instead we can use an npm package called `nodemon` . `nodemon` will ensure that the server gets restarted any time a JavaScript file is changed and saved.

To install `nodemon` run the command: `npm i -D nodemon` .

NB: `-D` means that the package will be saved as a ***developer*** dependency - a dependency that only the developer will need to use whilst building the application. Similar to `jest` , an end user should never have any need to use `nodemon` .

To run the server using `nodemon` : `nodemon listen.js`

NPM Scripts

Scripts are used to automate repetitive tasks.

Most scripts will be configured in the package.json file that `npm init` creates.

The scripts object in the `package.json` file is where NPM scripts will be defined. NPM scripts are written as JSON key-value pairs where the key is the name of the script and the value contains the script to be executed.

A common one that you will have seen already in the sprints is:

```
"scripts": {
     "test": "jest"
   }
```

This script specifies that when `npm test` is ran in the command line, to use the `jest` npm package that is installed in the `node_modules` to run the `spec` directory.

A script to use `nodemon` to run the `listen.js` file would look like this:

```
"scripts": {
     "dev": "nodemon listen.js"
   }
```

This script would be run with the following command: `npm run dev` .

The word `run` is needed here, because `dev` is not a predefined npm script in the way that `test` is/

More on npm scripts