Node Servers

Contents

Intro

As we have discussed, a server is at its most basic just a computer that is set up to listen to the internet, and that will send a response when it receives an incoming request.

It's really important not to overcomplicate things here. This is the most basic function of the internet. We have various computers running programs that can send requests (often a browser on a personal computer), and other computers that will be listening, and will send a response (a web server).

Here we are going to make a really basic server:

server.js:

```
const http = require('http')

const hostname = '127.0.0.1'
const port = 3000

const server = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain')
  res.end('Hello World')
})

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`)
})
```

Let's get this server going, and then run through this code and see if we can make sense of it.

```
> node server.js
```

We can see that we have logged out that the server is running. Let's first go to this address and see what we find there. We have our server, and now we are opening our client. Start up a browser, and then we can have a look at what we find.

You have made your first server!

Note that we can also go to the alias for this IP, localhost. We will more regularly be doing that. That makes the URL `http://localhost:3000`

Code Breakdown

Let's talk through the code above in some detail. One of the skills that I would like to encourage people to develop is the ability to use the code you can see to try to make sense of what is going on behind the scenes. This is something that you can do even if you don't or even can't have full knowledge of what the code is doing. At any level you should be aiming to try to make sense of what is going on, even when that's difficult.

```
const http = require('http')
```

The first line here is where we require in the module that we need. We get the code from inside there, and it runs much as though it's been printed above our code. There is a deep dive that we can do here, but for now what you need to know is that it's a library that we can use to help us get things done.

In this case we are bringing in the http module. Right now we aren't going to go into the details, but we can see what's happened hear as the result of the require function. Something - we will soon discuss what - is being saved into the constant `http`, and we now have access to whatever functionality that has brought us.

`require` is a very unusual function, but it is a function nonetheless. You have encountered functions, but here, even though the code may not be familiar, it is important to understand that, like every function, require is returning something. We are storing that something in a `const` for use later.

```
const hostname = '127.0.0.1'
const port = 3000
```

These two lines are fairly straightforward. We are assigning values to two variables, and those have type string, and integer. If we haven't started already, it might be helpful to think in those terms - type, and value. The type is more important than the value in many ways. The type will dictate how that data can be used. The value is the particular instance of that type. Strings can do different things and be processed in different ways than integers, and so on. Of course the specifics also matter - but as devs you will often need to get used to thinking about the type as well, which is less obvious to a beginner.

```
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain')
  res.end('Hello World')
});
```

This code is complicated at first. We are not going to go into all the underlying elements here, because - to be frank - I don't know how all this works underneath. And this is part of coding. We are all standing on

the shoulders of giants here, even when writing some small snippet of code that doesn't appear to do very much. There is a great deal going on underneath, but mainly important for us to understand enough to be able to make the code do what we need, and to get things going.

Of course the docs will be a big part of that, and this is how we will interact with the functions that this code provides us, but it's also worth pulling out what we can from the syntax.

Here we will start by trying to work out what this part is (in a stripped down form):

```
const server = http.createServer(
  ...
)
```

Can anyone take a guess what it is that we are looking at here?

The most basic thing that we can say about this, is that it is a function call. We are calling the function `createServer()` with some arguments, and this function will run, and then return something - as all functions will do (although the return value may not always be particularly important). We can tell it's a function call because of the parentheses, and because it's not a function definition. But where does this function come from if we didn't define it ourselves?

Let's have a look at what else can we tell from this code. Well, we also have a better idea of what was returned out of that `require()` statement.

Can you tell me what type of thing was stored in the constant called `http`? (This can be a very broad definition.)

We now know that it was an object. We can tell this because of the syntax `http.createServer`. We now can tell that the `'http'` module we called at the top of the file must export this object. We can even start to think about how that object might look - although only a small portion of it:

```
http = {
  createServer: function(cb) {
    const req = ...
    const res = ...
    ...
    const server = cb(req, res)
    return server
  }
}
```

This is probably not exactly how the code looks - it's the broader point here about thinking through what we can know that I'm making. The main point is that the item exported is an object, and that object will have a method attached (function) that takes a callback, and provides two arguments to that callback for us to comsume - to manipulate in some way or another, and then will return something. That something I have defined as `server` because I know what is coming up soon.

```
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});
```

Here I am adding back in the callback. We can see that those mysterious `req` and `res` objects that are being provided to our callback by the `createServer` function. We have no control over what these objects are (and we will discuss what they might look a bit like soon), but we can call them anything we like. We could have written this line `http.createServer((x, y) => {` if we liked, but we call them `req` and `res` somewhat by convention, but also because these two objects are the JS representations of the incoming HTTP request object and the outgoing HTTP response object respectively.

Furthermore, we can see that the request and response objects we have been disucssing are in fact JS objects here.

Why do we know this?

Again, we know this because of the `.` we can see in lines like this: `res.statusCode = 200`. In the example at hand we are not making use of the incoming request object. But we are making some alterations to the response object as seen in this line where we are setting the response code to 200. We can see that this `res` object also has at least two functions attached to it, and these are `setHeader(a, b)`, which can take two string arguments (at least), and `end(a)` which can take at least one string argument.

Exercise 1: RESponse?

Using what you learned yesterday and the code you can see above, and have a go at putting together what the HTTP response might look like. We can discuss this after you have had a go.

We can now see that the callback is modifying that response object in some ways. Please have a go at changing some of the values in this code. You can play around with the status code, the header, and the 'content'.

Demonstrate and look at the Chrome 'Network' tab

But looking at the code again, we need to keep going and see where this all ends. We can see here that the return value of this function is stored in `server`:

```
const server = http.createServer(
  ...
)
```

So we have to see where that constant is being used.

```
server.listen(port, hostname, function() {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Here I'm not going to dwell on the details as before. We can see that the `server` element returned from `createServer()` is an object, and it has a function attached to it call `listen`. This function takes two regular arguments, and a callback. The two regular arguments that we send it are the string and the integer that we defined earlier. In the callback we are running a `console.log` to get some feedback when the program starts, otherwise we will succeed silently, and it's a nice touch to know that things have run the way we had hoped.

Here, as with `createServer` and the `http` object that we imported, there is only so much that we can say, and the magic is happening behind the scenes in the http module. This is where our little Node program really gets going, and turns our computer into a server - albeit one that only runs locally. It is now listening on port 3000, we can send a request to it from the browser by entering the correct URL, and our server sends a response.

Conclusion (Basic Server)

No need to do this level of analysis every time you run some code. Most often you will just be trying to get something to work, and playing around with the arguments to functions. But every dev will hit points where they no longer understand what is happening, and just poking at arguments is not working anymore. At these times it can be useful to be able to break apart the code a bit, and demystify what you can so that you might be able to narrow down the search, or at least feel a little less lost in the code maze.

It's also important to mention here that if some of that was hard to understand, that's to be expected, and some of the elements from all of these lessons will become clearer over time. Feel free to ask me any questions that you might have.

Express

We have our nice little simple server, but as with much that goes on in development of any kind, if we want things to get more sophisticated, then rolling our own code for all the elements that we need to make a complicated server will be a lot of work. In modern web development we have access to the work of those that came before us.

It's worth quickly noting here that while importing tried and true code from the internet and using them in our programs is very often a godsend, and saves us a lot of time, everything has tradeoffs, and as you will find from time to time, you will occasionally waste more time trying to get the code from a particular package to bend to your will, than you stand to save by importing it.

This is not one of those instances - at least in my experience, and definitely for our purposes in this course. Express is a well tested and very convenient package that we can use to construct a modular server. Let's get started:

```
const express = require('express')
const app = express()
const port = 3000

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

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

To get this to work we are going to have to import the Express package. We learned earlier how to load that package in to where we need it:

```
> yarn add express
```

While we are adding packages, we are going to add one more, and digress briefly to talk about scripts in the `package.json`.

NodeMon

```
> yarn add nodemon
```

Nodemon is a program we can use to watch for changes to our project (NODE MONitor), and it will rerun the script whenever we save those changes, meaning that we don't have to stop and restart the server every time to make a change to the file - nodemon will do that for us (in this instance).

Once it is installed, we can run it from the terminal:

```
> nodemon server.js
```

Have a go and see how it's going. Make a change to the file, and save it, and see the outcome.

package.json scripts

We are going to expand our knowledge of the `package.json` file's capabilities. Hopefully your `package.json` looks something like this:

```
{
  "name": "week-3-code",
  "version": "1.0.0",
  "description": "code for the week three of class",
  "main": "index.js",
  "license": "MIT",
  "dependencies": {
    "express": "^4.17.1",
    "nodemon": "^2.0.4"
  }
}
```

We are going to ammend the file as follows:

```
{
  "name": "week-3-code",
  "version": "1.0.0",
  "description": "code for the week three of class",
  "main": "index.js",
  "license": "MIT",
  "dependencies": {
    "express": "^4.17.1",
    "nodemon": "^2.0.4"
  },
  "scripts": {
    "dev": "nodemon server.js"
  }
}
```

You can see above that we have added in a section for scripts, and a particular script that we want to run. `dev` is the 'name' of this script, and we can make use of this like so:

```
> yarn run dev
```

Here we are asking yarn to run the script in the `package.json` file called dev, which is the same as running:

```
> nodemon server.js
```

This has two beneifts.

The first is that while in this instance we aren't saving a heap of time, we are saving a little bit by having an alias for starting the program. However, sometimes the command to run a particular aspect of your program might be very complicated. In that instance we are saving considerable time not typing it our searching back through the terminal commands.

The second aspect is that you may forget the command (yes, this happens), and in most instances you are unlikely to be the only person working on the project.

Run the command through the `package.json` via yarn, and then check the response in the browser.

Exercise 2: Other scripts

To get the feel for these scripts, write in a few more possible terminal commands into this section, and then use yarn to execute them.

Code breakdown

This time around we aren't going to go into this in as much detail, and many elements here are similar. In fact, the chances are that the Express code uses the `http` module in the background. But there will be a few things to put out here.

After requiring in Express, we now have something stored in the constant `express`. Given the following code, what type of thing is `express`?

```
const app = express()
```

Maybe that was too obvious. Express is a function, and so a function must have been exported out of the Express package. We can see this because of the parentheses.

We know that `app` that we named must be an object because of the following code:

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

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

We can see that the `app` object has at least two functions attached to it: `get`, and `listen`. We aren't going to worry about `listen` much here, as we discussed a similar method above. Here we are more interested in the `get` function.

First off, what can you tell me about this function, defined on the `app` object?

firstly we can see it takes two arguments (at least), one that is a string, and the other is a callback function. Inside that callback function we can see code that bears similarities to the code we came across in the simple server. The big difference that we see is that we are adding a 'route', and responding in a particular way to this route.

To make this clearer though, let's expand the code, and then talk about what is going on here:

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World (via your first Express app)!')
})

app.get('/grasshopper', (req, res) => {
  res.send('You hit the grasshopper route!')
})

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

I've made the second route `/grasshopper` because it is totally random, to demonstrate that the routes that we create are arbritrary, and we are in control of whatever we want our server to respond to. The first route in the file is the 'root route', and although it doesn't look like much, it is very important. Basically, this is the Express route that will be hit when `http://localhost:3000` is hit without any further path specified. Express will see this as the same as `http://localhost:3000/` and this will be pattern matched to this function `app.get('/', (req, res) => {...`.

Outside of the root route though, it's all up to us. We could call a route `'/here-is-a-big-long-route'` or we could use `'/e'`. For now we have only the twoo routes above. Go to each in the browser. Let me know if you have any difficulty, or questions.

Now try going to a route that we haven't accounted for on the backend, say `http://localhost:3000/xyz`. What is happening in the browser?

Express is helping us out a bit here. It is sending a response for us when the path designated in the browser does not match the available routes in our `server.js`.

What we are seeing here is the first way that the browser can send some information to the backend. By going to `http://localhost:3000` it is connecting to our server. But then by adding a path, the browser is sending some more information to the server, and we now have the power to react to that on the back end.

Exercise 3: Getting more routes.

We are going to discuss the other alternatives later, but for now we are only going to use `app.get()` functions. Make at least ten more routes in the server. See what happens when you `console.log` something out within the method. Where does that appear? Play around with the different responses that you can send. Use the Express docs to see what is possible here. Send some responses as JSON. Send back some responses as some very basic HTML.

Discussion of the exercises and questions.

Self sufficiency

While it is always good to have good instruction, what we are doing here is also helping you to know how to move forward on your own. This is a key skill in web development. I need to look up even basic instructions often, and my knowledge is limited. There are many better developers than me out there, but even the best of the best will need to google things many times in their day. Stack Overflow is visited by all of us. And even more difficult is the times when we are out on our own and need to find a way to break new ground. We are going to have a go at having you investigate the next part a little on your own.

Exercise 4: Query Strings

To add a bit more functionality to our back end. We know from yesterday that a URL can contain what is known as a query string. We also know that Express is doing us a favour and creating a request object for us. Use the docs for URLs and for Express (or any resource you choose), and see if you can complete the following tasks:

1. Send a query string along with the `/grasshopper` path, and have it set the key `number` to 3. (Please ask me if this isn't clear.) In the `'/grasshopper'` route in your Express app, send back a

small HTML list with three items if number is set to 3, and an empty list otherwise.

Summary

- Node server
- Code breakdown
- Express server
- package.json scripts (nodemon)
- Routes
- Query strings
- Challenges