



EDINBURGH NAPIER UNIVERSITY

SET08101 Web Tech

Lab 7 - Beginning Node.JS

Dr Simon Wells

1 Aims

At the end of the practical portion of this topic you will:

-

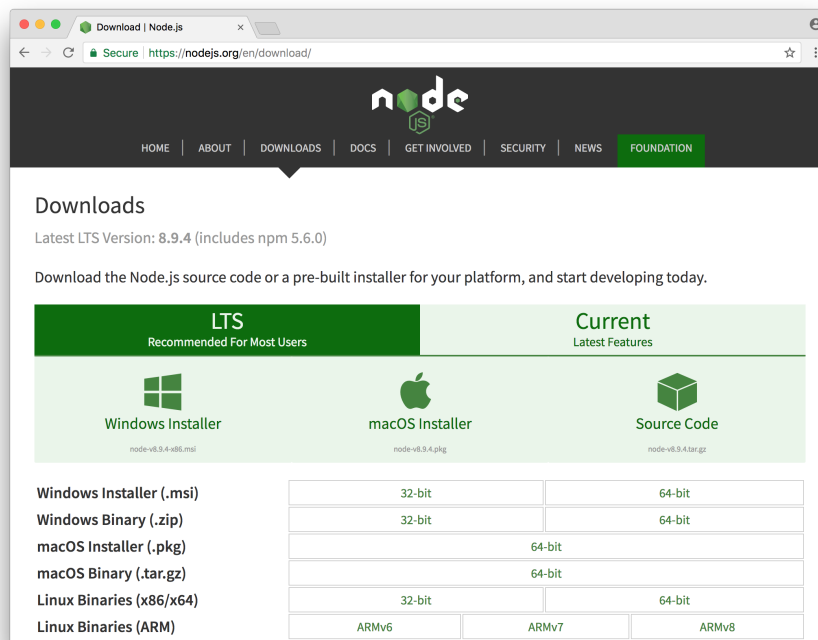
1.1 Introduction

We've concentrated so far on JavaScript in the browser or *on the client side*. Now let's look at using JS on the server as well. We'll start by running the Node console and using it as a place to type in JS code. This is quite a powerful little tool that we can use to work with JS as just another programming language rather than hosting it in a browser linked to HTML files. We'll then look at using Node as a HTTP server, to host

2 Activities

2.1 Getting Node.JS

Download the Node.JS Windows Binary distribution (zip file) from the node.js website: <https://nodejs.org/en/download/>¹. Don't use the MSI download if you're on a lab machine because this might need administrative privileges to install.



Once you've downloaded the zip, extract it to somewhere suitable such as a folder on your H: drive or on USB stick. Don't just double click zip file but instead ensure that you've properly extracted the contents of the zip file. If you have trouble extracting the contents of the zip file then use a decent zip archive tool like 7zip² instead.

Once you've extracted the Node files we should have a folder containing some useful tools. The most important here is node.exe which we can use to run the Node program, but we'll get to that in the next section...

If you are on Windows in an ENU lab then you will likely have to add Node to your path, but you can only do so temporarily using the command line, and must repeat the procedure each time we log in. Add your node folder to the Window Path from the command line like so (taking care to replace "C:\Program Files\Nodejs" with the actual path to where your node folder is):

¹Or directly download this zip: <https://nodejs.org/dist/v8.9.4/node-v8.9.4-win-x86.zip>

²7zip portable download: <https://portableapps.com/apps/utilities/7-zip-portable>

```
SET PATH=C:\Program Files\nodejs;%PATH%
```

If you're on a Mac then use homebrew to install Node, e.g.

```
$ brew install node
```

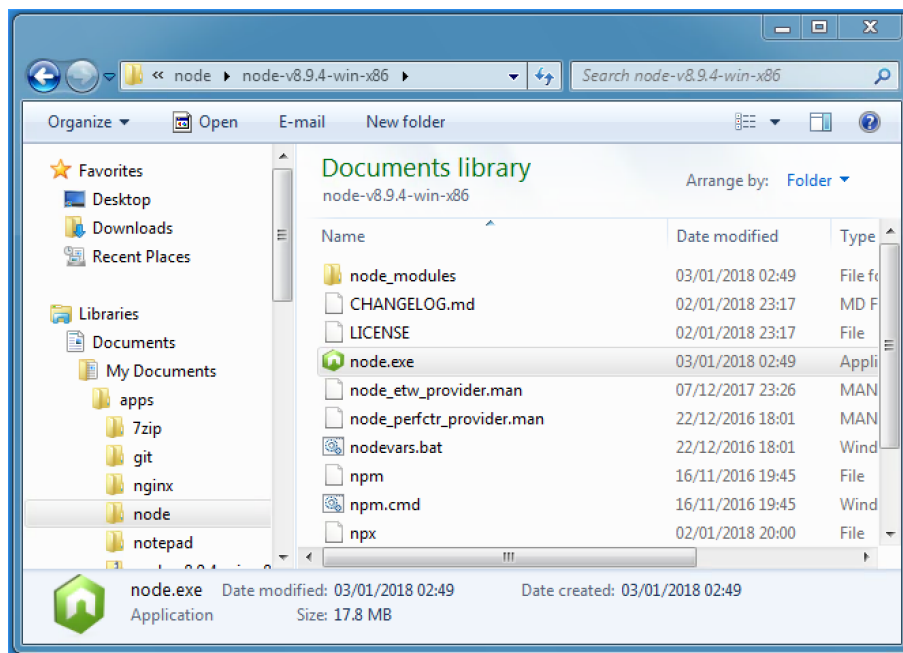
If you're on Linux then use APT, or you're distro's package manager to install Node, e.g.

```
$ apt-get install node
```

2.2 Node.JS Console

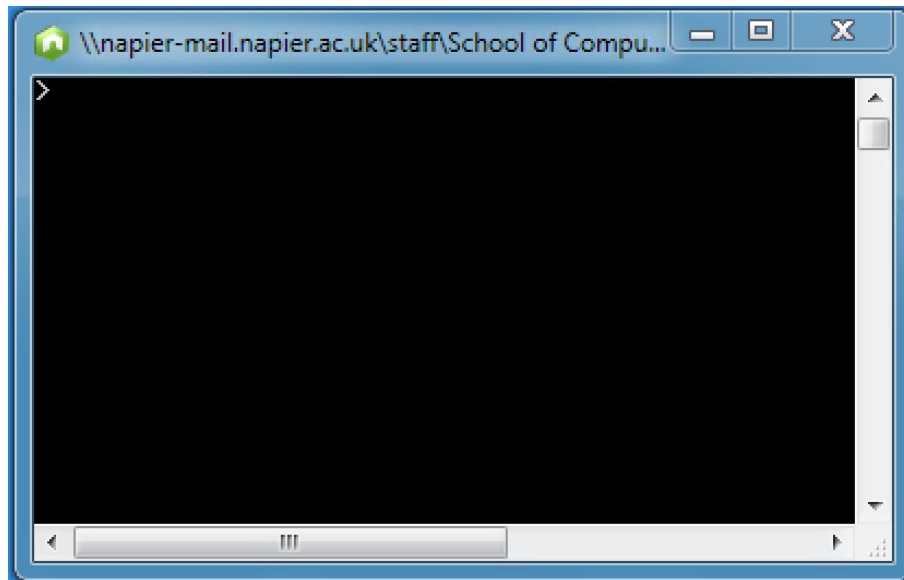
We can run node and use JS as just another programming language rather than as a *web* programming language. That is we can write some JS and execute it without having to have an HTML file to host our code and load it into the browser. In fact, we don't even need the browser at this point, although we will quickly return to using the browser as this is a web technologies module after all.

Node comes complete with a console³. A place where we can type JS code and see the output immediately. Such an environment is known as REPL, a Read-Evaluate-Print-Loop environment; it reads a command, works out what to do, prints the result, then loops back to the start and reads the next thing. We can get a node REPL by double clicking the node.exe program:

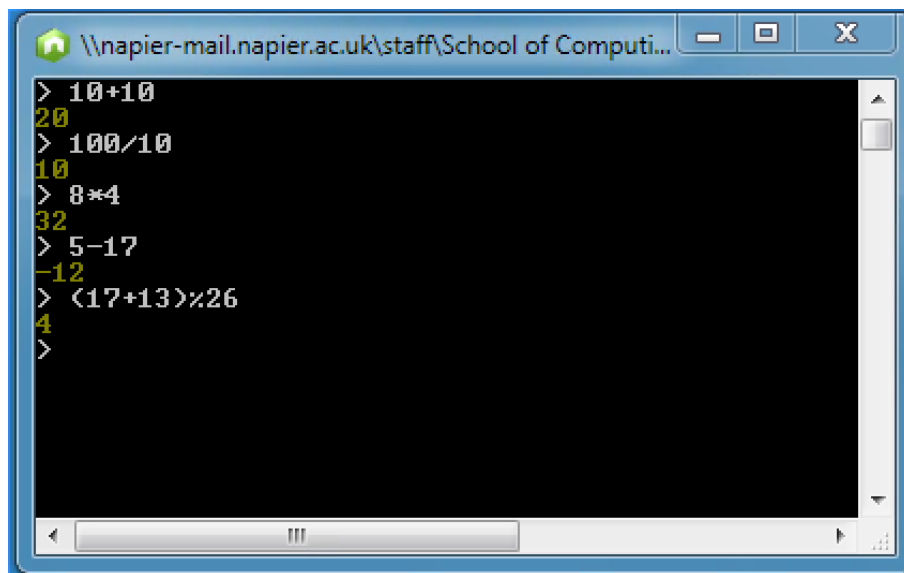


If all has gone well we should see a window like this:

³Documentation for the HTTP module is available here: <https://nodejs.org/api/repl.html>

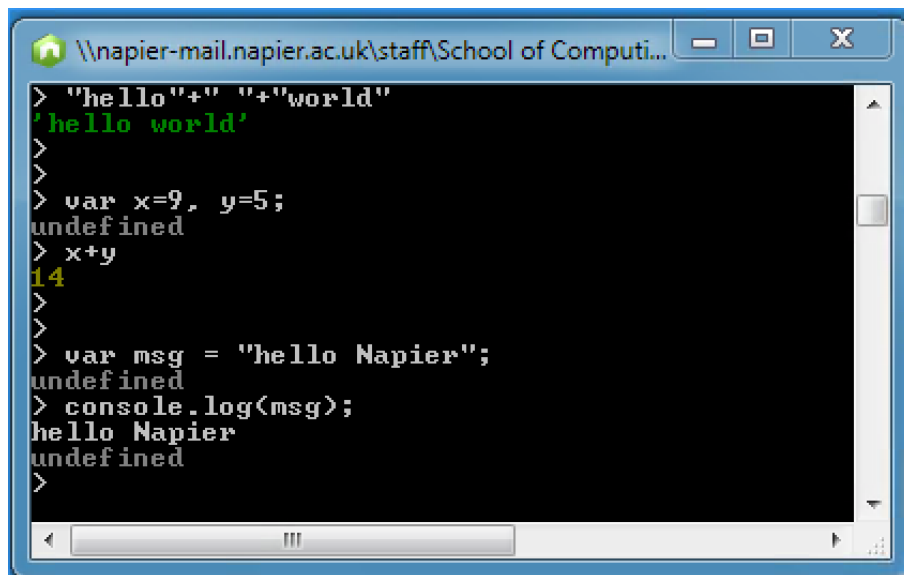


This is a place that we can type JS code into, why not give it a go? Let's start with some basic arithmetic:



We can use `ctrl+l` to clear the screen at any time and `ctrl+c` twice to exit⁴ from the REPL. As well as arithmetic we can also do standard JS programming, like the following:

⁴We can also exit the REPL by typing `.exit`

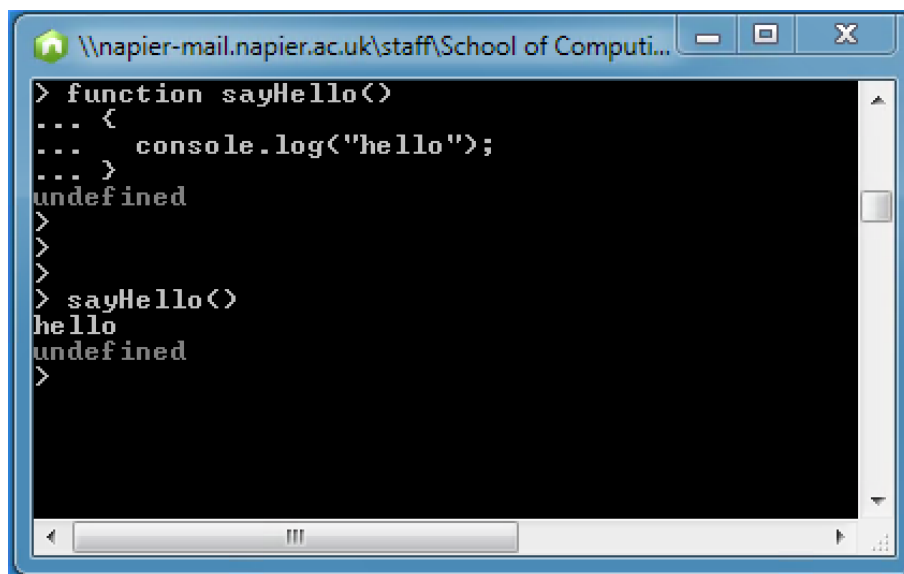


```

> "hello"+" "+"world"
'hello world'
>
>
> var x=9, y=5;
undefined
> x+y
14
>
>
> var msg = "hello Napier";
undefined
> console.log(msg);
hello Napier
undefined
>

```

We can also do multiline input by just pressing return wherever we need to. For example, to type in a function:



```

> function sayHello()
... <
...   console.log("hello");
... >
undefined
>
>
> sayHello()
hello
undefined
>

```

This is called continuity mode, because when we press return we are allowed by the REPL to write the next line as a continuation of our previous code. When in continuity mode we will see the ‘...’ as our prompt instead of ‘>’. You can leave continuity mode by typing ‘.break’ if needed.

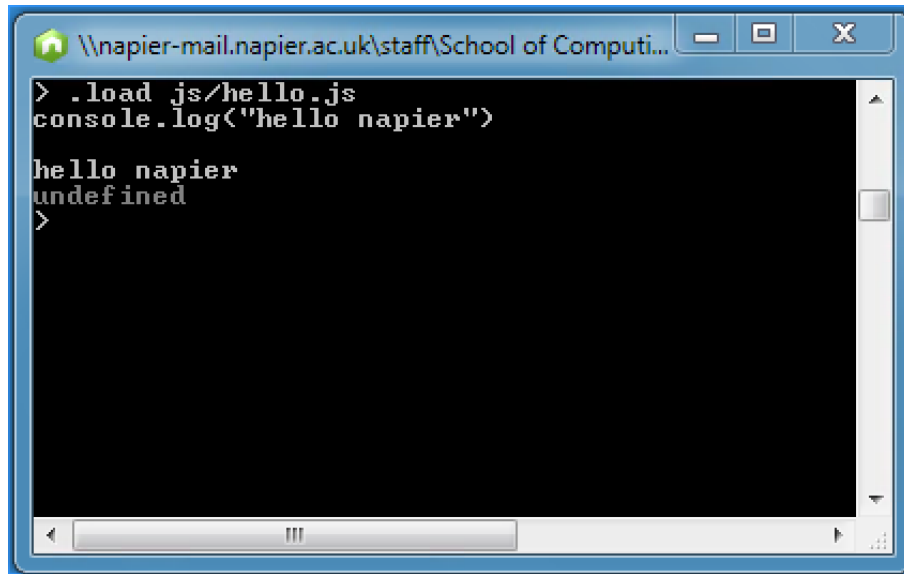
Node has a bunch of useful commands that you can explore, for example, *.help* which gives you information about some of the basic REPL commands. The tab key will display all the available commands. You can also use the up and down arrow keys to cycle through previous commands and lines of code that you’ve already typed in to stop your have to re-type things. The *.save* command will enable you to save your current REPL session, and *.load* will load the specified file into the Node REPL session. Ctrl+C (once) will terminate the current command and ctrl+c (twice) will exit the REPL.

2.3 Node.JS as a JavaScript execution environment

We can load and run JS written in external files in two ways. Firstly by loading them within the REPL and secondly by passing the JS file to be executed to node.exe when we start it. Let’s try both out. For convenience let’s first create a simple JS file called “hello.js” and add the following code to it using a preferred editor:

```
1 console.log('Hello Napier');
```

I saved this to a subfolder of the node folder called `js` so that I could easily access examples. You can save your JS files wherever you like but you'll have to specify where they are to Node. Now you can start node and use the `.load` command to load and execute your external JS file, e.g.



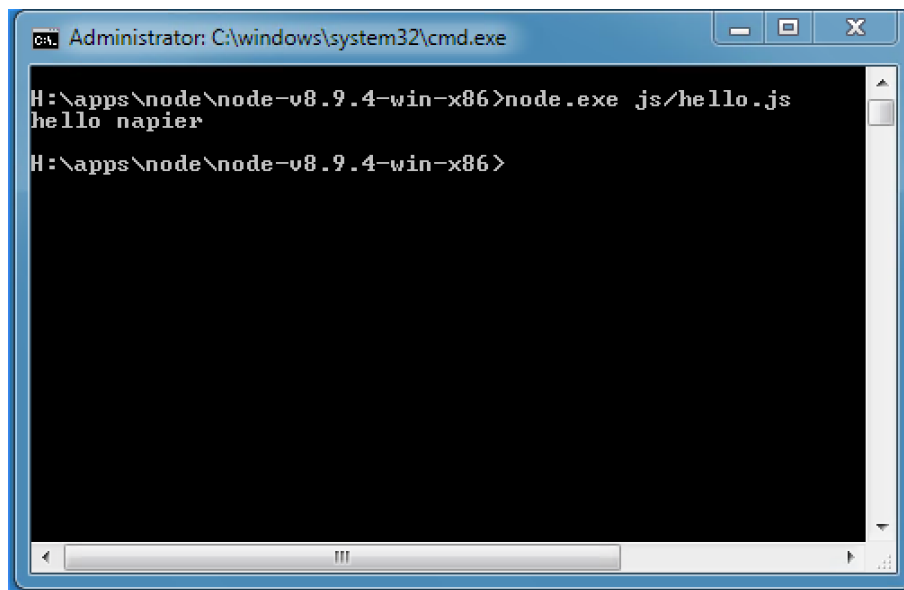
```

> .load js/hello.js
console.log("hello napier")

hello napier
undefined
>

```

Alternatively, instead of clicking `node.exe` we can start Node from the command line (`cmd.exe`)⁵ and tell it which JS file to load as an argument to the executable. For example:



```

Administrator: C:\windows\system32\cmd.exe

H:\apps\node\node-v8.9.4-win-x86>node.exe js/hello.js
hello napier

H:\apps\node\node-v8.9.4-win-x86>

```

Using these techniques we can pretty much use `node.exe` as our own little programming platform. But there's more, so much more we can do with Node. We've barely scratched the surface...

2.4 Node.JS as a hosting platform

In addition to hosting your server-side JavaScript, node can also host your regular HTML files. This can be a useful way to test out your site locally or to simplify your deployment. For example, to avoid having to install a full additional web-server if you already have node installed, or else to serve up a small number of static pages in addition to a web-app generation from JS.

We'll need to install the libraries that Node will use. We'll do this by opening the Windows command prompt, then navigating to the location of `node.exe`, but this time we're going to use the `"npm"` command:

⁵Go to the Start menu and type `"cmd.exe"` into the `"Search programs and files"` box.

```
$ npm install connect serve-static
```

The Node Package Manager (NPM)⁶ is a way to package up and distribute JS code for use with Node. We just installed two packages, one called *connect* and the other called *serve-static* which we'll use to serve up our existin HTML.

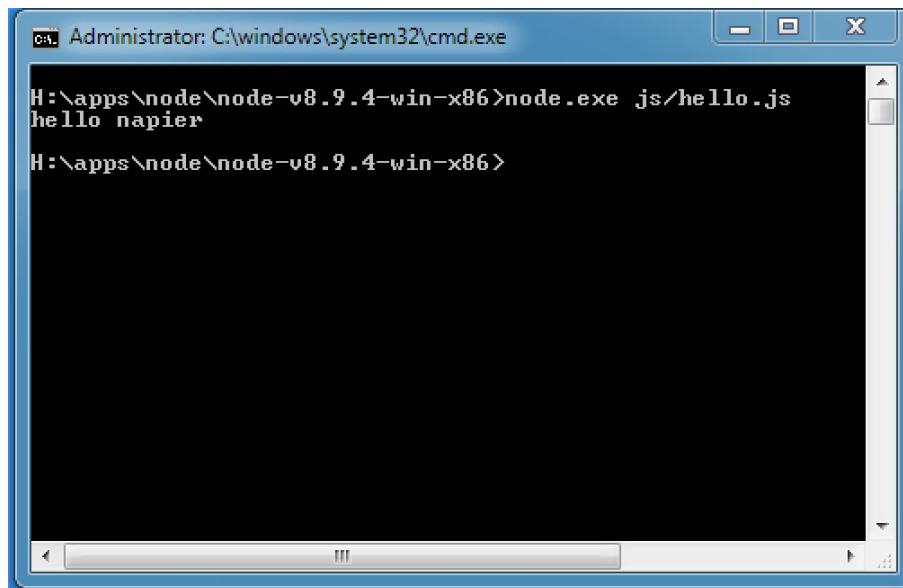
Now we need to create a small JS file, let's call it 'server.js', to tell node how to serve our HTML:

```
1 var connect = require('connect');
2 var serveStatic = require('serve-static');
3 connect().use(serveStatic(_dirname)).listen(8080, function(){
4     console.log('Server running on 8080...');
5 });
```

Now we also need a small index.html to serve up, e.g.

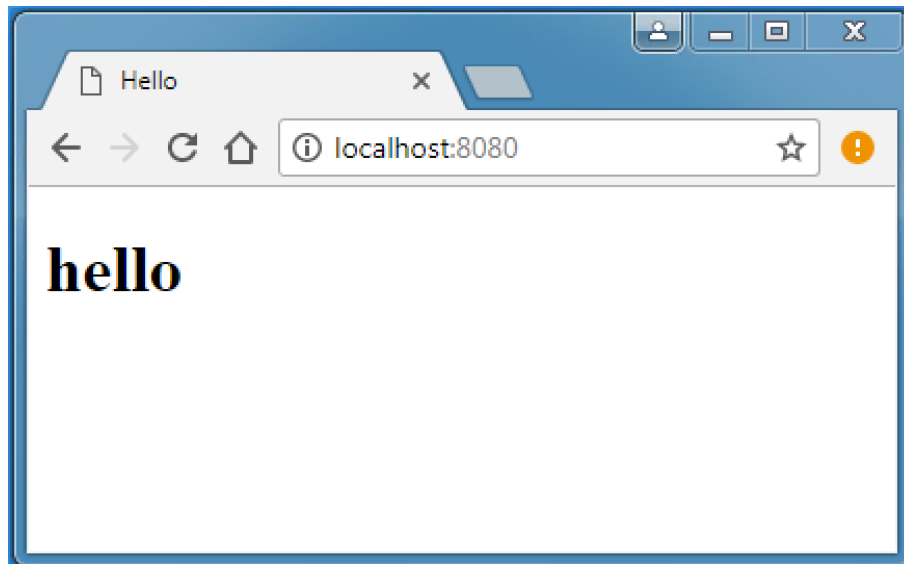
```
1 <!DOCTYPE html>
2 <html>
3   <head><title>Hello</title></head>
4   <body>
5     <h1>hello</h1>
6   </body>
7 </html>
```

Put both index.html and server.js into the js folder then invoke node.js to run server.js, e.g.



We should now be able to navigate to either <http://localhost:8080> or <http://127.0.0.1:8080> in our web browser and see our HTML file:

⁶It's well worth visiting the NPM site <https://www.npmjs.com/> and doing some background reading on this technology. There are thousands of ready made packages available though the NPM.



2.5 Building a Web Server in Node from (nearly) scratch

Let's now implement our own web framework using only the built-in *http* module⁷ that Node supplies for handling HTTP related functionality. Note that there are a lot of frameworks out there that do a similar thing such as *express*, but by implementing our own, we should get a much clearer understanding of what these frameworks actually do for us.

The *http* module has a *createServer* function that is very useful. We need to create an HTTP server, tell it to listen for events, and to respond to those events in an appropriate way.

```

1  const { createServer } = require("http");
2
3  const PORT = process.env.PORT || 5000;
4
5  const server = createServer();
6
7  server.on("request", (request, response) => {
8    response.end("Hello, world!");
9  });
10
11 server.listen(PORT, () => {
12   console.log(`starting server at port ${PORT}`);
13 });

```

What we've done is create a new instance of an HTTP server using the *createServer* function then add an event listener that waits for request events and prints a traditional message to our users when such an event occurs. Finally we tell the server which port to listen on for those incoming requests. In the real world we listen on port 80 because this is the default port that web browsers connect to. Browsers can connect to other ports, but the default happens automatically.

When a request event occurs we send a response, in this case using the *response.end* function to return a string. We can do more with *response*, for example, in this case we use multiple *response.write* calls followed by a final *response.end* call to construct our response:

```

1  const { createServer } = require("http");
2
3  const PORT = process.env.PORT || 5000;
4
5  const server = createServer();
6
7  server.on("request", (request, response) => {
8    response.write("Hello");
9    response.write(", ");
10   response.write("World!");
11   response.end();

```

⁷Documentation for the HTTP module is available here: <https://nodejs.org/api/http.html>


```

12 });
13
14 server.listen(PORT, () => {
15   console.log('starting server at port ${PORT}');
16 });

```

Consider how we could use this to construct a web page, perhaps having different functions to generate parts of our page. One thing to consider is that response is a stream, so we can keep adding data to a response until it is closed using the `response.end()` call. However, if we fail to close the stream, the client will keep waiting for more data and, from our server's perspective this constitutes a memory leak as we will have allocated resources that aren't reclaimed.

Now that we have a basic HTTP server that listens to requests and sends back responses, we probably want to do some of the other things that an HTTP server does. For example, most HTTP server don't just respond with the web page that you see in the browser. They also send header information and status codes so that client can decide how to interpret the content. Servers also don't just send the same response, they examine the request that they receive and return different pages depending on which page is requested. This last thing, responding with the actual content requested according to the address specified, is known as *routing*. We'll look at each in turn...

2.5.1 HTTP & HTTP Headers

This protocol is a text-based agreement for communicating Hypertext documents between clients and servers. It is request-response based, as we've just seen. But it also does a whole lot more. Here is what a typical HTTP request looks like:

```

1 GET / HTTP/1.1
2 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_2) AppleWebKit/537.36 (
  KHTML, like Gecko)
3 Host: blog.bloomca.me
4 Accept-Language: en-us
5 Accept-Encoding: gzip, deflate
6 Connection: Keep-Alive

```

There can also be a load more other information supplied as well, for example, specific headers to communicate access tokens as part of the server's access control, cookies, perhaps containing log-in data. Importantly, the request will always include a method, a path, and headers, which are a bunch of key-value pairs.

We can access the headers from within our server. Let's try that out.

```

1 const { createServer } = require("http");
2
3 const PORT = process.env.PORT || 5000;
4
5 const server = createServer();
6
7 server.on("request", (request, response) =>
8 {
9   const languages = request.headers["accept-language"];
10  response.end(languages);
11 });
12
13 server.listen(PORT, () =>
14 {
15   console.log('starting server at port ${PORT}');
16 });

```

When run this should show us which languages are accepted by the client.

```

1 en-GB,en;q=0.5

```

Just as the client can communicate metadata to the server through the request headers, when we create a response, we can also specify particular headers to communicate back to the client. A response is actually structured, again as plain text, as a set of headers followed by two newline characters which are in turn followed by the body of the response. The body is the bit that we usually think of as the web-page. The body is the bit where our HTML would be. Let's investigate the headers first though. We can set headers using either of the following `response.setHeader()` and `response.writeHead()` functions.

```

1 const { createServer } = require("http");
2
3 const PORT = process.env.PORT || 5000;
4
5 const server = createServer();
6
7 server.on("request", (request, response) =>
8 {
9     response.setHeader("content-type", "application/json");
10
11     response.end(JSON.stringify({ name: "simon" }));
12 });
13
14 server.listen(PORT, () =>
15 {
16     console.log(`starting server at port ${PORT}`);
17 });

```

When we run this we will get a JSON file returned, instead of a web-page, like this:

```

1 {"name": "simon"}

```

Consider how we could use this to perhaps build an API that listens for HTTP requests and then returns data to the client, for example, returning data that is loaded into a web-page, or that is consumed by a mobile application. Some approaches to developing dynamic web-sites involve storing all of the content as JSON which is rendered into HTML entirely in the browser.

If we inspect the headers that were returned alongside the JSON file, we'll also see that our "content-type" header has also been set.

```

1 Connection keep-alive
2 Content-Length 16
3 Date Fri, 01 Feb 2019 15:51:09 GMT
4 content-type application/json

```

2.5.2 HTTP Status Codes

Whilst web developers don't all agree on the semantics of how HTTP status codes should be used in some cases, there is broad consensus regarding their use for most common interactions on the web. For example, the use of 200 when everything is "OK", using 404 when an address is specified but there isn't actually a resource to return for that address. Similarly, if a route is protected and a user supplies incorrect credentials then a 403 might be used. If a user access a protected route without supplying credentials then a 401 unauthorized might happen. If a user make a bad request, perhaps omitting an essential URL parameter or supplying incorrect parameters then a 400 might be raised. One status code that you will probably be familiar with, especially if you are doing web development of dynamic sites, is the 500 code which is used to indicate an internal server error.

Status code responses can be set with either the `response.statusCode()` or `response.writeHead()` function, for example, using a custom status code...

```

1 const { createServer } = require("http");
2
3 const PORT = process.env.PORT || 5000;
4
5 const server = createServer();
6
7 server.on("request", (request, response) =>

```

```

8 {
9   response.writeHead(666, "OH Hell!!!");
10  response.end();
11 };
12
13 server.listen(PORT, () =>
14 {
15   console.log('starting server at port ${PORT}');
16 });

```

We can inspect the headers that are returned using our browser's developer tools. Usually this is via the network tab⁸.

2.5.3 Routing

One of the basis ideas with the Web based version of hypertext is that different resources are available from different addresses. Our webserver Node app has a single request handler, but needs to be able to repond to requests for different URLs. This is achieved quite simply by mapping different responses to different URL requests. Fore example, using the `request.url` property we can access the URL from the request, and then use a simple programming construct to handle each URL differently.

```

1  const { createServer } = require("http");
2
3  const PORT = process.env.PORT || 5000;
4
5  const server = createServer();
6
7  server.on("request", (request, response) =>
8  {
9    switch (request.url)
10   {
11     case "/":
12       response.end("Our beautiful homepage");
13       break;
14     case "/about":
15       response.end("An equally beautiful about page");
16       break;
17     default:
18       response.statusCode = 404;
19       response.end("Page not found!");
20   }
21 });
22
23 server.listen(PORT, () =>
24 {
25   console.log('starting server at port ${PORT}');
26 });

```

Efficiently routing requests and returning the correct page is an important function that is actually central to all web frameworks.

2.5.4 HTTP Methods

Just like with HTTP status codes, methods are a contentious area. Or rather, the correct use of HTTP methods can lead to significant discussion about their semantics. Whilst the semantics of GET is straightforward, the difference between PUT and POST are less clear and lead to implementaion subtleties as sites, in terms of hypertext applications, get larger and more complex. Generally, GET is used to retrieve a web resource, whilst PUT and POST are used to create new resources. They differ however in their semantics. Usually PUT is considered to be *idempotent* whilst POST is not.

```

1  const { createServer } = require("http");
2
3  const PORT = process.env.PORT || 5000;
4

```

⁸On the network tab refresh the page then click the page that you want to inspect headers for. These will then be displayed.

```

5 | const server = createServer();
6 |
7 | server.on("request", (request, response) =>
8 | {
9 |   if (request.method === "GET")
10 |   {
11 |     return response.end("Got a GET");
12 |   }
13 |   else if (request.method === "POST")
14 |   {
15 |     return response.end("Got a POST");
16 |   }
17 |   else
18 |   {
19 |     response.statusCode(400);
20 |     return response.end("Method not supported");
21 |   }
22 | });
23 |
24 | server.listen(PORT, () =>
25 | {
26 |   console.log('starting server at port ${PORT}');
27 | });

```

If we visit `localhost:5000/` in our browser then we will see the “Got a GET” message because our browser issues GET requests by default. It’s not straightforward to issue other other types of HTTP method. The only other message that you can do from a browser using HTML is POST from a form. Otherwise we need either some Javascript, or a tool that can issue other kinds of HTTP requests. For example, the command line “curl” tool will contact a server and can issue many other kinds of request. This makes tools like curl really useful for testing out your APIs and for inspecting server responses.

2.6 Cookies

We’ve seen cookies before in many other contexts, but we would expect a modern web framework to be able to set and use cookies. So let’s see how to do this.

```

1 | const { createServer } = require("http");
2 |
3 | const PORT = process.env.PORT || 5000;
4 |
5 | const server = createServer();
6 |
7 | server.on("request", (request, response) =>
8 | {
9 |   response.setHeader("Set-Cookie", ["daka=dakadaka", "type=ninja", "name=simon"]);
10 |   response.end('Your cookies are: ${request.headers.cookie}');
11 | });
12 |
13 | server.listen(PORT, () =>
14 | {
15 |   console.log('starting server at port ${PORT}');
16 | });

```

In this example we used the `setHeader` function to add a cookie line to the headers returned in our response. Remember cookies are just a string of text so the cookie header just contains our keys and values that must be encoded into that string. Cookies can get quite complex, in particular cookie values should be URL-encoded so that if your cookie’s value contains, for example, the ‘=’ character, then this won’t be misinterpreted as the key-value assignment operator. Note that when testing cookies it is a good idea to use your browser in private mode so that the cookies are automatical cleared. You sometimes also have to refresh your browser a couple of times before the cookies are picked up because the cookies are set are the response which also renders the page so we only see them at the next request.

2.6.1 Query Parameters

As HTTP is designed to be stateless, each request is independent so, just as we use cookies to persist information between requests, we can also use query parameters (also known as URL

or address parameters) to send small amounts of data to the server as part of our request. The server can then parse these parameters and use them in constructing the response. Consider, for example, a blog site that has multiple pages of posts, you might want to access a post on a given page so a query parameter can be used to specify which “page” of results to return, for example `localhost:5000/posts?page=3` would be used to request the third page of posts.

To access the query parameters we use the `request.url` property as we did earlier. Now however we check for a specific parameter in the query, e.g. “name” and then do something with that and its associated value, if it’s present.

```

1  const { createServer } = require("http");
2
3  const PORT = process.env.PORT || 5000;
4
5  const server = createServer();
6
7  server.on("request", (request, response) =>
8  {
9      const { query } = require("url").parse(request.url, true);
10     if (query.name)
11     {
12         response.end(`You requested parameter name with value ${query.name}`);
13     }
14     else
15     {
16         response.end("Hello!");
17     }
18 });
19
20 server.listen(PORT, () =>
21 {
22     console.log(`starting server at port ${PORT}`);
23 });

```

`http://localhost:5000/about?name=simon`

Query parameters are a useful way to make our URL schemes, the organisation of our hierarchy of addresses, more flexible.

2.6.2 Request Payload

Sometimes a request to our server will include a “payload”, some data, perhaps a JSON document that we want to make available to the server.

```

1  const { createServer } = require("http");
2
3  const PORT = process.env.PORT || 5000;
4
5  const server = createServer();
6
7  server.on("request", (request, response) =>
8  {
9
10     if (request.method === "POST") {
11         let data = "";
12         request.on("data", chunk => {
13             data += chunk;
14         });
15
16         request.on("end", () => {
17             try {
18                 const requestData = JSON.parse(data);
19                 requestData.ourMessage = "success";
20                 response.setHeader("Content-Type", "application/json");
21                 response.end(JSON.stringify(requestData));
22             } catch (e) {
23                 response.statusCode = 400;
24                 response.end("Invalid JSON");
25             }
26         });
27     } else {

```

```
28     response.statusCode = 400;
29     response.end("Please POST a JSON object");
30   }
31 }
32 });
33
34 server.listen(PORT, () =>
35 {
36     console.log('starting server at port ${PORT}');
37 });
```

As this is a POST request we need to call it using curl, for example, `$curl-XPOST-d '{"message": "helloworld"}' http://localhost:5000`

The most important thing to take away from this investigation is the great number of things that the server is doing when we make a request. It's tempting to underestimate the work that is done to service a single request, just because we clicked on a hyperlink.

3 Next

There are a whole bunch of Node.JS libraries that we can build on so that Node can act as a complete server-side web language. These libraries will help us to construct our own web-apps and APIs so that we can generate our user interface on-the-fly based upon what our user wants. This is how many highly interactive websites function and developing skills in this area will enable us to begin the process of going beyond the (mostly-) static websites that we've developed so far.