Introduction to NodeJS

NodeJS (see www.nodejs.org) is a technology which allows you to develop server-side applications in JavaScript. This has the a number of advantages, for example:

- You can use the same language to develop both client- and server-side components to your web application, minimising the number of languages you have to learn;
- A similar style of programming is used (NodeJS extensively uses asynchronous development with callbacks);
- JavaScript APIs, such as JSON.parse() and bind(), will work with NodeJS as they do with in-browser JavaScript;
- Because NodeJS is JavaScript it is easy to exchange JSON with the client.

Installing and using NodeJS

NodeJS can be downloaded from the website. It is already installed on Neptune. NodeJS typically does not work with an Apache server; instead you write your own custom, lightweight HTTP server with NodeJS running on a different port to Apache (port 8000 is used in the NodeSchool tutorials). This is similar to what you did with the WebSocket server last time.

npm

Node also comes with a ***package manager*** called ***npm***. This allows you to use add-ons to Node (libraries) which perform additional tasks, not part of the core node.js distribution, such as communicating with a database. To use npm to install new packages, you enter:

```
npm install <packagename>
```

at the command-line (e.g. DOS or Linux shell prompt).

Logging onto the Neptune command-line and using the text editor

Because you need to ***run*** a node script, you need to log on to Neptune and use the Neptune command-line (Linux shell prompt) as well as using FTP. To do this you should use ***putty*** (available on the lab computers, use the search). Find putty and then login with your normal Edward2 username and password. You should see a prompt rather like this (but on Neptune, not Edward):

Linux shell

Enter the following code in Notepad++, save as ***hello.js*** and upload to your home directory (the one ***before*** public_html) using FileZilla:

```
console.log("Hello world!");
```

On PuTTY run your Node.js script on the command line by entering:

```
node hello.js
```

You should see this output:

Node

Hopefully this script is self-explanatory. ***console.log()*** simply writes the specified text to the console, so that here, for example, ***"Hello World!"*** is displayed.

Basic node.js programming

Credits: some of the exercises here are loosely based on, but not the same as, those at ***nodeschool.io***

Command-line arguments

node.js programs often need to take in information passed into them from outside. For example, if we are writing a node.js server, which we will do later, you might want to tell it which port to run on. To do this, we use ***command-line arguments***. Here is an example of a node script which uses them:

```
console.log("Number of command line arguments: " + proc
for(var i=0; i<process.argv.length; i++)
{
    console.log("Argument " + i +" is: " + process.argv
}
```

Write this script in Notepad++ and upload with FileZilla, as for the Hello World script. Then run it from the command prompt on Neptune, for example:

```
node args.js Fred
```

substituting "Fred" for your name. You should see output like:

Node command-line

Note how this is working. ***process.argv*** is an ***array*** of all the command-line arguments passed to the program. Note however that the "node" command and the filename of the script are ***included*** in this array of arguments, ***not just the user defined argument "Fred"***. The user defined arguments start at ***position 2 (starting from zero) in the array***.

Exercise

Write a node program which takes in a first name and last name (e.g. "Fred" and "Smith") as command-line arguments and displays a message such as:

```
Hello Fred Smith!
```

if the user specified: ***node arguments2.js Fred Smith***

HTTP Servers

Probably the most useful thing to learn with node.js is how to create a lightweight ***HTTP server***. With node.js, typically you do not use Apache but instead write your own server using the Node.js HTTP APIs. Here is a very basic HTTP server:

```
var http = require("http");

var server = http.createServer(
    (request, response) =>
    {
        response.write("Requested with a method of :"
        response.end();
    }
);

server.listen(8000);
```

Looking at how this works:

- First we import the http ***module***. Node comes with ***modules*** which are libraries: they provide additional capabilities to the Node environment. Many are provided by third parties. The ***npm*** command line tool allows you to install new modules.
- Next we ***create an HTTP server*** with the createServer() method. This returns a server object. Note how we ***specify a callback function (arrow function) as a parameter to the createServer() method***.

This is an example of an ***anonymous function***, which you have seen already.

- The callback runs each time we get a request from a client. It takes two parameters, ***request*** and ***response***, representing the HTTP request and response respectively. Look at what it's doing: we use the ***write()*** method of the request to send some content back to the client, and then ***we must finish the response by calling response.end()***
- This simple example just displays what HTTP method (GET or POST) was used to request the server. The ***request.method*** property contains the HTTP method.

Try running this. Rather than 8000, use your own personal port number that I have given you: you will all be running your own node servers on Neptune and therefore must each use different ports. You'll find it goes into an infinite loop and does not return you to the command line: this is because it is running the server continuously listening for requests.

Now try sending an HTTP request to the server from your web browser. Enter:

```
http://ephp.solent.ac.uk:NNNN/
```

from the browser, where NNNN is your own personal port number. You should see this output appear:

```
Requested with a method of: GET
```

because you have sent a GET request to the server.

API documentation

See here for full Node API documentation on the HTTP module.

POST requests

Now try sending a POST request. Write this form and FTP it to the server with FileZilla in the normal way. Note how the form's action is pointing to your Node server. Again, replace NNNN with your port number.

```
<html>
<body>

<h1>Node POST</h1>
<form method="post" action="http://ephp.solent.ac.uk:NN
<input type="submit" value="Go!" />
</form>

</body>
```

```
</html>
```

Load this into your browser as normal, e.g:

```
http://ephp.solent.ac.uk/~jsmith/form.html
```

You'll see a blank form. Click the submit button, and now this output will appear:

```
Requested with a method of: POST
```

because you have now used a method of POST to request your Node server.

GET parameters (query strings and form fields)

We can read query string parameters and form fields in from Node. The following example shows how to do this:

```
var http = require("http");
var url = require("url");

var server = http.createServer (
        (request, response) =>
        {
            var details = url.parse(request.url, true)
            if(details.query.artist)
            {
                response.write("Artist is: " + details
            }
            else
            {
                response.write("Artist not supplied");
            }
            response.end();
        } );
server.listen(8000);
```

To explain the new parts of this:

- We now need a new module, the ***url*** module. This is a module for parsing URLs, such as loading query strings into JavaScript objects.
- In the callback function, we first use ***url.parse*** to parse the request's URL into a JavaScript object which we can use to obtain various aspects of the URL. The second parameter (***true***) specifies that we want to load

the GET parameters into a JavaScript object, which make them easy to read.

- We then test whether the parameter ***details.query.artist*** exists. ***artist*** would be a query string parameter or form field, the equivalent of $_GET ["artist"] in PHP. So the code will display the artist if it was supplied, or "Artist not supplied" if it was not.

Try running this server and requesting it with:

```
http://ephp.solent.ac.uk:NNNN/?artist=Oasis
```

and

```
http://ephp.solent.ac.uk:NNNN/
```

The former will respond with:

```
Artist is: Oasis
```

while the latter will respond with

```
Artist not supplied
```

Reading POST data

Reading POST data is a bit more complex than GET data and requires the use of the ***querystring*** Node module:

```
var http = require('http');
var url = require('url');
var qs = require('querystring');

var server = http.createServer
    ( (request,response) =>  {
            if(request.method=='POST') {
                var content = '';
                request.on('data', chunk => {
                    content += chunk;
                });

                request.on('end', ()=> {
                    var post = qs.parse(content);
                    response.write('POST: artist=' + po
                    response.end();
                });
            } else {
                var details = url.parse(request.url, tr
                response.write('artist=' +details.query
                response.end();
            }
    } );

server.listen(8000);
```

If the method is POST, we add a couple of event handlers to the ***request*** object. The 'data' event occurs when the server receives ***some*** data from the client: note that the data is sent in ***chunks*** rather than all at once. So the 'data' event handler adds the current chunk to the ***content*** variable. Then, when ***all*** the data has been sent, the 'end' event occurs. In this, we call the ***parse()*** method of the querystring module object ***qs***; this will return an object containing the POST data.

Writing out HTML

By default, the content type of a response generated by a Node server is "text/plain", so it generates plain text rather than HTML. To change the content type to something else, we use the ***response.writeHead()*** method, e.g:

```
response.writeHead (200, { 'Content-Type' : 'text/html
```

Note how ***writeHead()*** takes two parameters:

- The status code;
- An object containing instructions to be included in the HTTP response header. Here, we only have one instruction, the Content-Type.

Writing out HTML - exercise

1. Modify the previous example to write out HTML. Prove it works by putting the artist within <strong> tags so you get an output such as: You entered the artist: **Oasis**
2. Modify the example so that a 400 Bad Request is sent back if the artist was not supplied.

API endpoints

In Node you can define different functions of your web server via so-called API endpoints: basically different URLs representing different functionality. For example, a HitTastic! Node server could use the URL

```
http://ephp.solent.ac.uk:NNNN/search
```

to do a search, or

```
http://ephp.solent.ac.uk:NNNN/buy
```

to buy an item of music. So this URL

```
http://ephp.solent.ac.uk:NNNN/search?artist=Oasis
```

could search for all Oasis hits, and

```
http://ephp.solent.ac.uk:NNNN/buy
```

with an ID parameter of 123 could buy the song with the ID of 123.

The same server would handle all API endpoints. To find out which API endpoint was specified, you use the ***pathname*** property of the object returned from ***url.parse***. For example:

```
var http = require("http");
var url = require("url");

var server =http.createServer (
    (request, response) =>
    {
        var details = url.parse(request.url, true);
        response.write("API endpoint: " + details.pathn
        response.end();
    } );
server.listen(8000);
```

Try this out and request the server with the following URLs:

```
http://ephp.solent.ac.uk:NNNN/search
http://ephp.solent.ac.uk:NNNN/buy
```

You should see the outputs:

```
API endpoint: /search
```

and

```
API endpoint: /buy
```

Exercise

1. Modify the example so that if the API endpoint is ***search***, it displays the artist specified in the query string as the parameter ***artist*** e.g:

   ```
   http://ephp.solent.ac.uk:NNNN/search?artist=Oasis
   ```

   would give:

   ```
   You are searching for songs by: Oasis
   ```

   while if the API endpoint is ***buy***, it displays the song ID specified in the query string as the ***ID*** parameter, e.g:

   ```
   http://ephp.solent.ac.uk:NNNN/buy?ID=123
   ```

   would give:

   ```
   You are buying the song with the ID of: 123
   ```

   (technically you should use POST, not GET, for a buy operation as you are updating something, but using GET for now makes it easier to test).
2. Modify the example further so that a 400 is sent back if:
   - a search operation is done without supplying an artist;
   - a buy operation is done without supplying an ID, or
   - the API endpoint is something other than ***/search*** or ***/buy***.

More advanced topic: sending HTTP requests

This is not compulsory but you should look at it if you finish the above exercises.

From a node script you can send HTTP requests to a remote web server. To do this you need to include the ***http*** module. Here is a script which makes a request to a remote web service:

```
var http = require("http");
http.get("http://edward2.solent.ac.uk/wad/xmlhits.php",


    (response) =>
    {
        var result="";
        response.on("data", data =>
                    {
                        result += data.toString();
                    }
                );
```

```
            response.on("end", () =>
                        {
                            console.log("Data received: "
                        }
                    );

    }
);
```

Note how this is working as it might be a bit more complex than you expect.

- We firstly create an ***http*** object (similar to creating an ***fs*** object) and then call its ***get()*** method to perform a GET request. We specify a callback to run when we get a response back.
- One thing to bear in mind is that ***not all the response might be received at once***. We might get what we call a "chunked" response, where the response is delivered in small chunks. In our callback function, we receive a response object and attach two further callbacks to it:
    - the "data" callback. This runs when we get a chunk of response back;
    - the "end" callback. This runs when the entire data has been received.

  Note how in the "data" callback we add this chunk of data to the variable ***result***, and only in the "end" callback, when we know everything is received, do we display the result.

Exercise

Write a node.js script to contact your HitTastic! web service (from week 1). Take the artist from the command-line and use DOM parsing to display the results in a user-friendly way.

Further resources

- API documentation
- NodeSchool, practical walk-through exercises which cover the subjects presented here but in more detail.