Intro to Node

Summary

- What is Node?
- Install Node
- Running some Node files
- Clients and servers
- Protocols and formats
- The HTTP Protocol
- Parsing
- URL
- Packages (Yarn)
- Challenges

Introduction

Node is JS for the backend. While JS is the development language of the browser, NodeJS is the language of the OS (meaning it runs in the terminal). Node was created so that developers who already knew how to code using JS, could now use essentially the same language, but do so within the OS, rather than only in the browser (or needing to learn another back-end language).

The main differences are that Node won't have access to those browser specific APIs (the functions you've been using, like `addEventListener()`) that you have been working with, and like most other backend languages, it will have functionality for working with files and implementing APIs. There are a few other differences, but we will discover them as we go.

Downloading Node

First we will need to download Node.

https://nodejs.org/en/download/

Running NodeJS

Here we are making a very simple Node file:

```
console.log('this is my first program in node')
console.log('it runs from the command line')

function getRandomInt(max) {
  return Math.floor(Math.random() * Math.floor(max));
}

const randVal = getRandomInt(23)
console.log(randVal)
```

And now we can run it from the command line.

`> node myFirstProgram.js` We can see here that the output is now on the command line. The main thing that you'll have to get used to for now, is that the browser isn't involved, and so your terminal is your locus of control.

Let's have a look at another example:

```
const fs = require('fs')

// fs.readFile takes the file path and the callback
fs.readFile('hello.md', function(err, data) {

    // if there's an error, log it and return
    if (err) {
        console.error(err)
        return
    }
    // Print the string representation of the data
    console.log(data.toString())
})
```

Here we have quite a bit going on, but we can work through it slowly. The first line `const fs = require('fs')` is us taking advantage of the work of others. It's a bit of efficient laziness. Let's not get too deep in the weeds here, but we can talk about the syntax that we can see, and what might be happening. We will dig deeper later. The main point is that we are importing some code into our program, and can make use of the work done for us.

We can see that `require` is a function from the parentheses, and it is returning something into the const fs (which we could call anything, but because it's file system options, 'fs' makes sense). We can also see that `fs` is an object from this code: `fs.readFile(..)` as the dot gives it away.

`.readfile` is a function on that `fs` object, and we can see it takes two arguments, a string, and a callback. That callback is given two arguments, an error, `err`, and the file contents in the `data` variable. This `data` is actually a buffer, so we need to turn it into a string for us to read it. We aren't going to get into buffers now, but the short story is that they are an area in memory. We could also specify an encoding when we call `readfile`, and can return to this later.

Some basic Node challenges

To get into just being comfortable with Node, here are a few challenges:

`challenges/basic-node-challenges.js`

Useful terminal commands

- `> ls` : list all the files and directories in the current directory
- `> mkdir <dir-name>`: make a new directory
- `> cd <dir-name>`: change location into this new directory
- `> touch <file-name>`: create a new file called file-name.
- `> New-Item <file-name>`: (Windows) create a new file called file-name.

- `> cd ..`: go back up the filesystem one level.
- `> rmdir <dir-name>`: remove a directory (directory much be empty).
- `> rm <file-name>`: remove a file (cannot be undone).

Clients and Servers

Front End and Back End

Before we continue with a few little exercises in Node, we are going to discuss a few concepts that will be useful, and then we will see if we can combine the concepts and Node, and make a little parser (or two).

Now is a good time to introduce a few analogous ideas: client and server, and front-end and back-end.

It might be a little confusing at first but these ideas are different ways of saying the same thing. JS runs in the browser, and the browser is also called the client, or the front-end. It's the 'visible' aspect of web development, and this code ends up running in the browser of the user (most of the time).

The backend is also often the server. This is where we will be running our Node code, which runs on our operating system. There are a few interchangable terms for the backend, and but let's concentrate on server, or server-side.

To understand more fully, we need to have an idea what a server is. It's important not to overcomplicate this (there are lots of tricky details about the web, but the overview is simple).

Server?

A web server is merely a computer that has been set up to listen to incoming HTTP traffic (there are a few different types of server, but from here on server will mean web server). That's really all. It's a computer that is running a program, and that program is keeping a port open to the internet and listening to the traffic. Of course this is more complicated underneath, but for now, as web devs, this is the part that interests us.

To understand a little about what is happening with a web server, we need to talk a little about the internet more broadly. There is a lot that could be said here, and many levels to dive into. But for our purposes I think we can keep it relatively simple to start with, and dive a bit deeper down the track when the basics have sunk in.

The Internet? HTTP?

The internet is run on the basis of the HTTP and TCP protocols. You may be familiar with these terms. Again, keeping it short, the TCP protocol is the way the data is sent, and HTTP tells us how to interpret the data that is sent via TCP. You can think of TCP as the water that the data swims through. TCP is the communication instructions, and HTTP is the format of the data being sent via that medium.

To understand that a little more, we need an understanding of what a protocol is. A protocol is essentially just a shared understanding about the shape of some data, or the method to asssemble that data. If we are going to send any information, we need to have some way for both parties involved, the sender and the receiver, to be able to know how to assemble and then how to read that information.

Protocols and Formats

We could make up a way to send some data that is called asterisk-ampersand protocol. We can decide ahead of time that the data that we send is going to be deliniated by '*', and each line of data will end with '&'. A file containing this data might look something like this:

```
kate*smith*32*2.6*3900&
chris*baker*77*2.2*2020&
agnes*murphy*23*1.9*3889&
```

As long as the person on the other end knows the symbols that help define the structure of the protocol, then they can use coding tools to simply arrange the data on the other end. Here the file we have is very similar to a format that you might be familiar with, and that is a CSV. Another common format is JSON, which we will get to soon enough. The main point to take away here is that we are trying to send data from one person to another, and we need a way to decide ahead of time how to structure the data so that both parties can make sense of it on the other end. We could have said that we would delineate columns with 'cat' and the end of a row with 'dog'. That would have it's issues, but could be done as long as people on each end of the communication understand the format. This last point is the most important: it is a shared understanding.

Looking at the example above, we can see that this won't vanquish all of the potential issues that we might have around sending data. For example, the data itself is rather meaningless without more information. We might be able to deduce that the first column is a given name, and the second is a surname. The next column might be age, but also might not. And so on. So we might want to add to this protocol.

We could have a set of headers send that give a little more infomation about the data itself. Here we might use ^, and % to be our markers. Now the file might look something like this:

```
fname^lname^age^score^studentno%
kate*smith*32*2.6*3900&
chris*baker*77*2.2*2020&
agnes*murphy*23*1.9*3889&
```

We have added a header to the file that helps to give some colour to the data therein.

All of this is to go the long way around to helping understand HTTP, but the more important message is to understand that communication and data storage will most often involve a format, or a understood data structure, sometimes also called a protocol.

The Structure of HTTP

The way HTTP operates is that the browser sends a request, and the server responds with a response.

The format for HTTP is:

1. A start-line describing the requests to be implemented, or its status of whether successful or a failure. This start-line is always a single line.

2. An optional set of HTTP headers specifying the request, or describing the body included in the message.
3. A blank line indicating all meta-information for the request has been sent.
4. An optional body containing data associated with the request (like content of an HTML form), or the document associated with a response. The presence of the body and its size is specified by the start-line and HTTP headers.

This will have variations in terms of the request and the response, but the main thing to recognise here is that the browser will be making requests, some of which will have extra bits of information (the body), and similarly for the response from the server, which may also contain extra data.

We can talk through the details shortly.

First I want to talk about the other side of this coin, and that is what to do when we have a particular format incoming. This is where the concept of a parser comes into things.

Parser?

A parser is a program that takes an object that has a known format (or protocol or shape), and pulls apart the constituent elements. For the file above:

```
fname^lname^age^score^studentno%
kate*smith*32*2.6*3900&
chris*baker*77*2.2*2020&
agnes*murphy*23*1.9*3889&
```

you might end up with something like

```
data = [
    {
        firstName: 'Kate',
        lastName: 'Smith',
        age: 32,
        score: 2.6,
        studentNo: 3900
    },
    // ...
]
```

We have taken the raw data, and turned it into something that we can now code with in Node. We can use the full power of the language to search through the array, or whatever else we need. This is the power of knowing the format, and then being able to pull it in to our coding world. There are many different ways that we could deal with this data, and it would depend in part on the end goal. There is no right way or wrong way to process it (although there will be better and worse ways, depending on the goal), and even then the way the data has been processed will have to be made clear to anyone else using the program that we have created.

Structure of a URL

A URL is a more complicated element than we often bother to think about. It is the first way that we will discuss that a browser will send information to a server, which is again something we don't think a lot about. The URL carries information that the server will use to determine the content that will be sent back, and for this information to make sense and be useful it must be in a particular format.

Here I am going to throw it back to you. Have a look at the following link:

https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_URL

What I would like you to attempt (in pairs if possible), is to use Node to write a URL parser.

Node packages

The exercise you are about to embark on may have some tricky elements that might benefit from the ecosystem that Node brings with it in the form of packages. These are not available in the JS environment (although this issue is complicated). Keeping things simple for now, we are going to have a brief run through how to import packages into your program, and how to then make use of them.

Yarn

Head here to install yarn:

https://classic.yarnpkg.com/en/docs/install/#mac-stable

We now have the power to install some code that others have written that we can make use of. We are using Yarn to download thesse packages, but also to keep track of them. When we use them in our code they are called dependencies (because our code now depends on them). Yarn will help us to keep track of this by adding them into a file that is called `package.json`.

The `package.json` file is the accumulation of all the dependencies for our project (and has a few other tricks up its sleve, which we will get to in time). We need to initialise it with this command:

```
> yarn init
```

and answer the questions that are asked of us. Many can be ignored, but it is worth paying attention to what they are to get a feel for what is happening.

This website goes into some more detail:

https://classic.yarnpkg.com/en/docs/usage

You will also notice that a folder called `node_modules` has turned up in the current directory. This is where the code that you are importing will be kept. We will talk a little bit more about this down the track, but for now you can ignore it (or investigate as you like), but do not make changes to this file.

The other main aspect to this way of storing dependencies is that the project can now be shipped to other devs and they are able to install all of the dependencies locally with one command:

```
> yarn install
```

which will save everyone a considerable amount of time.

Exercise 1: Package search

Search out a few interesting packages, and install them. Share any cool packages with your fellow students. Take note of the changes to the `package.json` file.

Exercise 2: Callback Boilerplate

Give them a chance to have a look at `3-callback-boilerplate.js`

EXERCISE 3: Parse a URL

The aim of this exercise is to drill home the idea of formats, protocols, and parsing. A URL is a nice simple one to have a go at, but also a little more complicated than people might think at first. This can be done in Node and the code can be talked through.

To do this you will first need to understand the format of a URL. Then you will need to think about how to break it apart into its constituent pieces, and have a plan for how to put that into meaningful Node structure. As you go through this, try to think of edge cases that could trip you up.

To make things a little bit simpler, I suggest hard coding a URL into your program at the top, and then attempting to break it down. As your program develops, you should attempt to parse URLs that fit the format, but that have unusual features (these would be your edge cases).

- Working in teams chat (if needed)
- Edge cases chat (if needed)

Exercise 4: CSV parser (optional)

Write a program to parse CSV files. This is more practice with Node, with formats, and with forming meaningful Node (JS) structures. This exercise will also require some work with file system functions, as in this scenario we will draw from the `data.csv` that will be provided. This reinforces the back end nature of Node (processing files on the computer), and demonstrate a different format to be parsed.

Exercise 5: Super extension

Start writing a parser for an incoming HTTP request. This will involve investigating what an incoming request looks like, and breaking it into its constituent parts. It's worth noting here that HTTP requests are sent as strings.

If you get this far you could do the same for an HTTP response, and see how you go.

As a final uber challenge you might think about how you would parse the HTML that could be sent in the response. This is a very very difficult task, so I would recommend just putting a little time into thinking about how you might go about this, and what type of functions you might need to make this work. With the difficulty and time available I think that pseudo code might be the way to go here.

Extension (optional for now): Response Codes

Have a look at the various response codes. Can also do a bit more diving into the structure of various aspects of the HTTP objects, and various.

Summary

We covered a lot of content here, and many concepts, some of which will take time to sink in fully. But let's try to draw all of what we have done together so that we might have a sense of the overview from today.

Node

First we talked about Node, which is backend JS. It is just a way to write JS code that can be run from the OS itself, rather than needing to be run in a browser. Node comes with libraries that we don't have in JS (file system libs, for example), but also lacks the functions for dealing with browser. Because Node can run from the terminal within the OS, we can also make a server using Node - something we will do tomorrow.

Front and backends, formats and protocols, HTTP and the web

The front-end, or client, for our purposes, is the browser. The back-end, that we now have the ability to control with Node, is often a server (which we are about to get to). The most basic pattern to understand about the internet is that browsers make HTTP requests, and servers (computers that are connected to the internet and listening to HTTP traffic), are waiting to send an HTTP response.

But to be able to efficiently understand the messages being send to and fro, it helps to have a set format, or protocol. In this case we have the HTTP protocol. But the concept of protocols and formats is a more general one, and we can see several instances of this same concept (CSVs, URLs, and HTTP).

In this class we put them all together. We installed and ran Node, and then used Node to *parse* data in various formats, which means to take that format and turn it into something that is useful within the language we are using.

Next

Next up we see these elements in action on the internet..