**Bootcamp Week 2:**

Day 6:

New Terminology:

- Jest

- Servers

- Ports

- Express.js

- Supertest & Cheerio

- DNS – Domain Name Server translates Domain names (urls) into IP addresses.

- DHCP

- IP - An Internet Protocol address (IP address) is a numerical label assigned to each device connected to a computer network that uses the Internet Protocol for communication. An IP address serves two principal functions: host or network interface identification and location addressing.

- \_\_dirname – directory name, the directory that the file is in.

- path.join

- sendFile & send

- req.query & req.params

- .listen – listening to the server port

- ${target} – string templates

- GET() & POST() request – server requests, get() is requesting information and posting is submitting info.

Every function in JavaScript is either synchronous or asynchronous. It's important to understand the difference because synchronicity defines how the function behaves to the code that calls it. When your code calls a *synchronous* function, the function makes your code wait until it's finished before letting the next line run (also called *blocking*). This is true for most of the functions you've called so far, and likely all of the functions you've written so far. An *asynchronous* function returns control to your code right away while it continues to do its work. This allows your code and the function to be working at the same time, asynchronously. Here are some examples:

console.log('before') synchConsoleLog('working') // synchronous function console.log('after')

As you would expect, the output to the console will look like:

before working after

Now let's consider an async example:

console.log('before') asyncConsoleLog('working') // async function console.log('after')

Depending on how long asyncConsoleLog takes to execute, the output of this could be:

before after working

Async functions are appropriate when they are going to take a long time to complete, so the calling code isn't blocked. It turns out that it takes a relatively long time to read or write to the filesystem (the hard drive) and the network (e.g. sending an HTTP request). In these cases, it's common to use async functions.

The calling code often needs to know when the async function has finished its work. It's also common for the async function to return something (a result, error code or status change) which the calling code needs. In JavaScript, there are a number of different ways to approach this issue: callbacks, promises, and generators. We'll get to promises (and maybe generators) later in the course, but callbacks are the easiest to understand and very common.

**Asynchronous** – when things happen at once. When code is being tested at virtually the same time.

**Blocking** – when code is running, there’s other code that will be blocked until the first part of code has run. Opposite to Asynchronies.

**Synchronies Callbacks:**

- Generic Callbacks (functions)

math.js:

function doMath(op1, op2, operation) {

return operation(op1 + op2)

}

Passing in the two operands and passing in the operation. doMath is going to run this operation with op1 and op2.

function add (op1, op2) {

return op1 + op2

}

defining the function add

const result = add (2, 7)

calling the function – we know this because we are executing the function with parentheses.

Const result = doMath(2, 7, add)

In this example we are calling the function doMath() but the add function is just being PASSED through the function, even though it is a function itself.

console.log(result)

Defining Functions 🡪

Calling Functions 🡪

Passing Functions 🡪

**Asynchronous Callbacks:**

I/O – input/output (Asynchronies)

Const A = 4 (this is saved in memory (RAM) and takes microseconds to access this)

When you’re accessing something from the fileSystem (hard drive) it takes milliseconds.

When you’re accessing something from the Network it takes seconds.

filestuff.js

**fs.readdir(path[, options], callback)#**

* path [<string>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#String_type) | [<Buffer>](https://nodejs.org/api/buffer.html#buffer_class_buffer) | [<URL>](https://nodejs.org/api/url.html#url_the_whatwg_url_api)
* options [<string>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#String_type) | [<Object>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object)
  + encoding [<string>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#String_type) **Default:** 'utf8'
* callback [<Function>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function)
  + err [<Error>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error)
  + files [<string[]>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#String_type) | [<Buffer[]>](https://nodejs.org/api/buffer.html#buffer_class_buffer)

Asynchronous [readdir(3)](http://man7.org/linux/man-pages/man3/readdir.3.html). Reads the contents of a directory. The callback gets two arguments (err, files) where files is an array of the names of the files in the directory excluding '.' and '..'.

The optional options argument can be a string specifying an encoding, or an object with an encoding property specifying the character encoding to use for the filenames passed to the callback. If the encoding is set to 'buffer', the filenames returned will be passed as Buffer objects.

const fs = require(‘fs’)

fs.readdir(\_\_dirname, showFiles) //this happens first and keeps going then reads console.log immediately after. Unlike Synchronous operations.

console.log(‘I am here’)

function showFiles (err, files) {

if (err) {

return console.error(err)

}

files.forEach(file => console.log(file))

}

It’ll call showFiles once it has finished reading the directory.

When it hits readDir, because its asynchronous it makes everything happen at the same time – so it branches off to fileSystem while everything is still going (eg. The console.log()) and then the branch merges in the same time as the original run finishes. So theres different things happening at once. Whereas in synchronous operation, when it hit readDir, it would stop the operation to run that and then keep going, rather than continuing to run other stuff at the same time. Asynchronous is does a lot more in the same amount of time as Synchronous.

**fs.readdirSync(path[, options])**

* path [<string>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#String_type) | [<Buffer>](https://nodejs.org/api/buffer.html#buffer_class_buffer) | [<URL>](https://nodejs.org/api/url.html#url_the_whatwg_url_api)
* options [<string>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#String_type) | [<Object>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object)
  + encoding [<string>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#String_type) **Default:** 'utf8'

Synchronous [readdir(3)](http://man7.org/linux/man-pages/man3/readdir.3.html). Returns an array of filenames excluding '.' and '..'.

The optional options argument can be a string specifying an encoding, or an object with an encoding property specifying the character encoding to use for the filenames passed to the callback. If the encoding is set to 'buffer', the filenames returned will be passed as Buffer objects.

In synchronous function (blocking function) its going to use the return keyword and get a return, and what it returns we need to save into a variable. In Asynchronous you don’t get this, theres no assignment operator – we get it in the callback, it’ll pass the value you want into the callback and then you will get it that way.

How to tell if your function is Async or Sync?

When you have a callback you generally assume its an async function.

**Callback Functions:**

A *callback* is a function we pass to another function as a parameter. When the receiving function wants to send something *back* to the first function, it *calls* the callback.

Remember Array.map()? It takes a callback function as a parameter:

// Returns [2, 3, 4] [1, 2, 3].map(function (n) { return n + 1 })

The anonymous function passed to map is called by map: we don't call it ourselves, we just tell it what to do once it gets called. This is a very common pattern in functional programming.

"Hello, Metaphorical Pizza?"

We often use callbacks when working with *asynchronous functions*. An async function will go off and do its work, then call the callback you passed to it when it has some results to give you (or an error to report). In the meantime, the rest of your program continues on its merry way, not waiting for the operation to complete.

Say you're hungry, and you order a pizza. You don't know ahead of time how long the pizza is going to take to arrive (30 min guarantees notwithstanding) and you're certainly not going to stand there doing nothing while you wait. You go on doing whatever you were doing, but you've provided your address to the pizza place so that when your order is ready, they can deliver it to you. They can also report an error: "We ran out of anchovies", or "The oven exploded".

function deliver(err, pizza) { if (!err) { answerTheDoor() payFor(pizza) enjoy(pizza) } } orderAPizza('pepperoni', deliver) makeCoffee() callFriends()

In this example, deliver is a function being used as a callback. We immediately go on to doing other things after we've called orderAPizza, and the code inside the callback will only execute when the pizza place calls deliver(null, pizza). (See below for why we used null here.)

Async functions might take a little getting used to in the beginning, but you'll use them a *lot* in your JavaScript programming and they will become second nature.

Reading from a file

Node programs often pass callback functions that handle the data you expect to receive from APIs, from databases, or from the filesystem. The filesystem is an easily accessible way of practicing this technique.

var fs = require('fs') fs.readFile('animals.txt', 'utf8', function (err, animals) { if (err) { console.error("Couldn't read file:", err.message) } else { console.log(animals) } })

This program reads a file from the directory in which it was run, and outputs the result to the terminal. Try it! When you're done, try modifying the string 'animals.txt' to a filename you know *doesn't* exist, and run the program again. You should see an error that Node can't open the file you asked for.

The first thing to notice about this example is that everything interesting happens *inside the callback*. Remember, we're not making things happen ourselves: we're *defining* what *will* happen once readFilehas finished its work: once it has either succeeded or failed to deliver the contents of the file.

The next thing to notice is that the callback has a very particular structure. It's an *error-first callback*, a common convention in the Node world which assumes that the first parameter of the callback will always be either an error object or null. We should always check the error to make certain it's null before we try to use the animals parameter.

This is a really common pattern in JavaScript programs. You'll be seeing it a lot, so get some practice in!

Named callbacks

The callback doesn't have to be an anonymous function. It's often better to use a named function, both because it can be re-used and because it can be more readable:

fs.readFile('birds.txt', 'utf8', displayFileContents) fs.readFile('trees.txt', 'utf8', displayFileContents) function displayFileContents (err, contents) { if (err) { console.error("Couldn't read file:", err.message) } else { console.log(contents) } })

Right away this makes our code a bit more DRY. We have to do the same thing with both files so why write the function twice? It can also make the code easier to debug since we only have to look in one location for the problem.

**Testing Asynchronous Functions:**

**index.js:**

module.exports = {

readFolder,

showFiles

}

function readFolder (path, cb) {

fs.readdir(path, cb)

}

function showFiles (err, files) {

if (err) return console.error(err)

files.forEach(file => console.log(file))

}

readFolder(\_\_dirname, showFiles)

**index.test.js:**

/\* global test expect \*/

const index = require(‘index’)

test(‘Test harness is working’, () => {

expect(true).toBeTruthy()

})

test(‘readFolder calls the callback’, ()

index.readFolder(‘p’, assert)

function assert (err, files) {

expect(err).toBeFalsy()

expect(files.length).toBe(5)

}

})

**Bootcamp Day 7:**

New Terminology:

- Asynchronous Functions – you have to put your assertions in your callback for Async functions. You don’t access the FS with Async.

- Synchronous Functions - .map, .filter, .reduce are synchronous functions that take callback functions. Blocking.

- Callbacks

- I/O

- Time-slicing

- fs.readdir – specific to node

- fs.readfile - specific to node

- fs.writefile - specific to node

-fs.appendfile - specific to node

- prompt.

- Dependency Injections:

dependency injection is a technique whereby one object supplies the dependencies of another object. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it. The service is made part of the client's state.[1] Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

**Handlebar Framework:**

To install:

Npm init –y

Yarn add express express-handlebars

**index.js:**

const server = require(‘./server’)

const port = 3000

server.listen(port, () => {

console.log(Server is listening on port’, port)

})

**server.js:**

const express = require(‘express’)

const hbs = require(‘express-handlebars’)

const server = express ()

module.exports = server

server.engine(‘hbs’, hbs({

extname: ‘hbs’

}))

server.set(‘view engine’, ‘hbs’)

server.get(‘/’, (req, res) => {

res.send(‘Hello world!’)

})

// run node index to test if Server is listening on port 3000

rendering on Server.js:

render = template and data put together

the handlebars template is stored in the views folder.

{{….}} = placeholder

**server.js:**

const express = require(‘express’)

const hbs = require(‘express-handlebars’)

const server = express ()

module.exports = server

server.engine(‘hbs’, hbs({

extname: ‘hbs’

}))

server.set(‘view engine’, ‘hbs’)

This is defining the object

// server.get(‘/’, (req, res) => {

// const data = {recipient: ‘Stina”}

// res.render(‘home’, data)

This is an anonymous object. You don’t have to define it but it is easier.

// server.get(‘/’, (req, res) => {

// res.render(‘home’, {recipient: ‘Stina”})

// })

server.get(‘/’, (req, res) => {

const data = {

recipient: {name: req.query.name}

}

res.render(‘home’, data)

})

**Bootcamp Day 8:**

New Terminology:

POST requests – The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.

Partial Views/Nested Partials - The folder within views that breaks down the HTML file into footer/header/nav etc and allows you to be able to pull in these files in multiple other files.

- res.render

- server.set

- server.engine

- {{#each}} – handlebars loop.

- {{this}} – when you are looping over an array of strings, use {{this}} to reference a specific string.

More Server-Side Rendering (Puppy Exercise):

const fs = require(‘fs’)

const path = require(‘path’)

const express = require(‘express’)

const hbs = require(‘express-handlebars’)

const server = express ()

module.exports = server

server.engine(‘hbs’, hbs({

extname: ‘hbs’

}))

server.set(‘view engine’, ‘hbs’)

server.get(‘/’, (req, res) => {

const filePath = path.join(\_\_dirname, ‘puppies.json’)

fs.readfile(filePath, (err, fileContents) => {

if (err) {

res.status(500)

}

const puppies = JSON.parse(fileContents)

res.render(‘home’, puppies)

})

})

module.exports = server