



The Complete Developers Guide

Section 1

INSTALLING & EXPLORING NODE.JS

Installing Node.js

We can visit <https://nodejs.org/en/> homepage to download node.js for our operating system. There are two versions:

The LTS version (Long Term Support) and the Current release version.

It is advisable to download the Current release, as this will contain all the latest features of Node.js. As at August 2019, the latest version is 12.7.0 Current.

We can test by running a simple terminal command on our machine to see if node.js was actually installed correctly:

```
:~$ node -v
```

This will return a version of node that is running on your machine and will indicate that node was installed.

If we receive command not found, then this means node was not installed and we need to reinstall it.

Installing A Code Editor

A text editor is required so that we can start writing out our node.js scripts. There are a couple of open source code editors (*i.e. free*) that we can use such as Atom, Sublime, notepad++ and Visual Studio Code to name a few.

It is highly recommended to download and install Visual Studio Code as your code editor of choice as it contains many great features, themes and extensions to really customise the editor to fit your needs. We can also debug our node.js scripts inside of the editor which is also a great feature.

Links:

<https://code.visualstudio.com/>

<https://atom.io>

<https://www.sublimetext.com/>

What is Node.js?

Node.js came about when the original developers took JavaScript (*something that ran only on browsers*) and they allowed it to run as a stand alone process on our machine. This means that we can use the JavaScript programming language outside of the browsers and build things such as a web server which can access the file system and connect to databases. Therefore, JavaScript developers could now use

JavaScript as a server side language to create web servers, command line interfaces (CLI), application backends and more. Therefore, at a high level, node.js is a way to run JavaScript code on the server as opposed to being forced to run on only the client.

"Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine."

There are all sorts of JavaScript engines out there running all major browsers, however, the Chrome V8 engine is the same engine that runs the Google Chrome browser. The V8 engine is an open source project. The job of the engine is to take JavaScript code and compile it down into machine code that the machine can actually execute. The V8 engine is written in the C++ language.

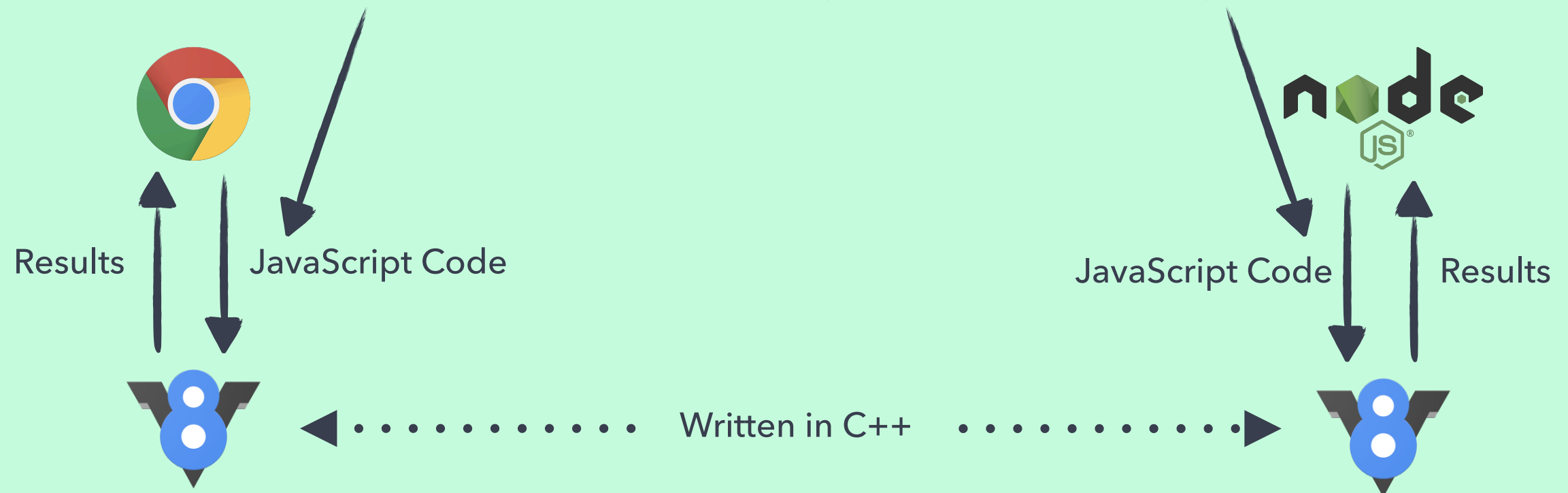
Node.js is not a programming language. The runtime is something that provides custom functionality i.e. various tools and libraries specific to an environment. So in the case of Chrome, the runtime provides V8 with various objects and functions that allow JavaScript developers in the chrome browser to do things like add a button click events or manipulate the DOM. The node runtime provides various tools that node developers need such as libraries for setting up web-servers integrating with the file system.

The JavaScript is provided to the V8 engine as part of the runtime. JavaScript does not know the methods e.g. read file from disk or get item from local storage but C++ does know the methods. So we have a series of methods that can be used in our JavaScript code which are, in reality, just running C++ code behind the scenes.

Therefore, we have C++ bindings to V8 which allows JavaScript, in essence, to do anything that C++ can do. Below is a visualisation to demonstrate exactly what is happening with the V8 JavaScript Engine:

JavaScript (Chrome)	C++
localStorage.getItem	Some C++ function
document.querySelector	Some C++ function

JavaScript (Node.js)	C++
fs.readFile	Some C++ function
os.platform	Some C++ function



We can run the following code in our terminal

```
:~$ node
```

What we get by running this command is a little place where we can run individual node JavaScript statements which is also known as repl (*read eval print loop*). These are not bash commands. All of the core JavaScript features we use to are still available when using node.js because those are provided by the V8 engine itself.

It is important to note that some of the JavaScript features available in the browser are not available in node. For example, Chrome has a object called window which provides all the different methods and properties we have access to. This makes sense in the context of JavaScript in the browser because we actually have a window to work with. This will not work on node because window is something specifically provided by the Chrome runtime when JavaScript is running in the application. Node does not have a window and it does not need window and therefore window is not provided.

With node we have something similar to window; we have a variable called global:

```
:~$ node  
[> global
```

Global stores a lot of the global things we can access such as the methods and properties available to us. The browser does not have access to the global variable.

Another difference between the browser runtime and the node runtime, is that the browser has access to something called a document. The document allows us to work with the DOM (document object manager) which allows us to manipulate the document objects on the page. Again, this makes sense for a browser where we have a DOM and it does not make sense for node where we do not have a DOM. In node we have something kind of similar to document called process. This provides various methods and properties for the node process that is running (e.g. `exit()` allows us to exit the current node process).

```
[> process.exit( )
```

Why Should You Use Node.js?

The node.js skillset is in high demand with companies like Netflix, LinkedIn and Paypal all using node in production. Node is also useful for developers anywhere on the stack i.e. front end developers and back end engineers.

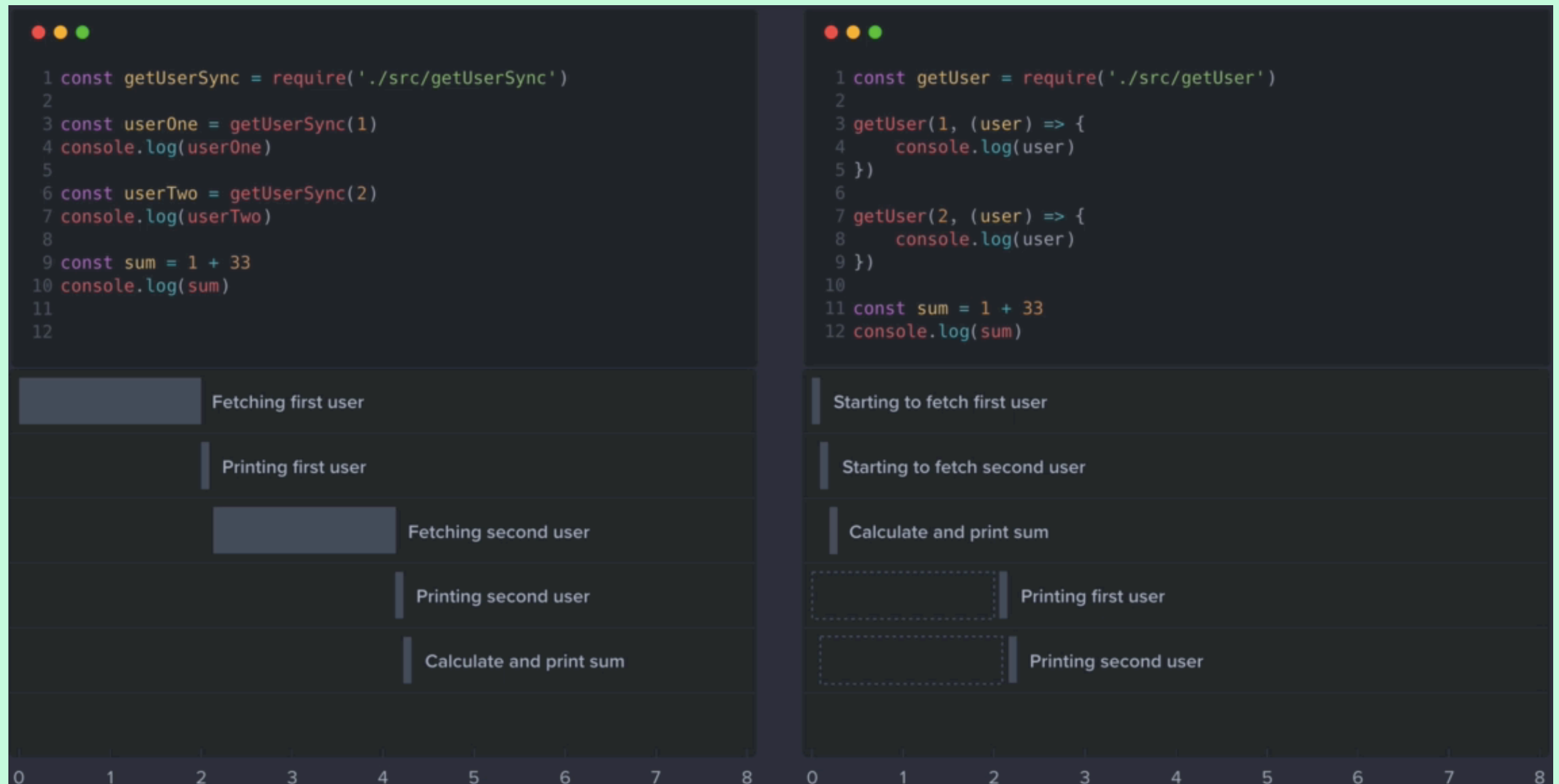
"Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js package ecosystem, npm, is the largest ecosystem of open source libraries in the world."

I/O stands for input/output and our node application is going to use I/O anytime it is trying to communicate with the outside world. So if our node app needs to communicate with a machine its running on, that is an I/O for example, reading some data from a file on the file system. If our node app needs to communicate with some other server that is also a I/O for example, querying a database to fetch some records for a given user which is a I/O operation. I/O operations take time. With node.js we get non-blocking I/O which means that while our node application is waiting for a response, it can continue to process other code and make other requests (i.e. Asynchronous).

Below is diagram representation of two I/O (one blocking and one non-blocking) attempting to do the same thing of taking in a user id and go off to a database to fetch the user profile and print the information to the console for two different users and then finally run a calculation of adding two numbers and print that sum to the console. The timeline is going to help us compare the relative operating speeds of the blocking and non-blocking I/O codes.

Blocking I/O

Non-Blocking I/O



The non-blocking I/O is twice as fast at running the code compared to the blocking I/O. This is because we were able to overlap the part of our application that took the longest, which was waiting for the I/O to finish and continue to complete other tasks in the code. This is what makes node.js ideal to develop with.

The node package manager (npm) was also installed on our machine when installing node. This package manager allows us to download open source pre-written packages from the web that we can use inside of our applications as dependencies. We can find all sorts of available packages on <https://www.npmjs.com> and we can use the terminal to install these within our application directory. These libraries and packages makes our life much easier when developing with node.js and we will make heavy use of npm packages which real developers do when they are building out their node applications.

Your First Node.js Script

Using Visual Studio Code, we can create a new file within a directory and name the file with an extension of .js at the end. For example, we can create a file called hello.js which will be our first node.js script file. This is a JavaScript file similar to what we would write if we were to run it in the browser. Our first line of code will use the console.log() to print a message to the console for anyone who runs this script.

hello.js

```
1. console.log('Hello world!');
```

We can use the node documentation to view all the different modules that we can work with based on the version of node we are running on our machine (<https://nodejs.org/en/docs/>).

We can run the following command in the terminal (*VS Code has an integrated terminal we can also use*):

```
:~directoryPathContainingNodeScript$ node hello.js
```

We must ensure within the terminal that we are within the directory path of the file we want to run.

Section 2

NODE.JS MODULE SYSTEM

The Node.js Module System

The node.js module system is the most fundamental feature of node. This is going to allow us to take advantage of all of the interesting things that node provides.

There are all sorts of modules available in node.js and some, for example the Console, are available to us globally. This means that we do not need to do anything special in order to use them. We access console in order to use it. However, other modules require us to actually load them in before they can be used in our scripts for example, the File System module.

The File System module allows us to access the operating system's file system. We will be able to read and write, append files and check if a given file or directory exists and many more interesting features. We will learn more about this module system.

Importing Node.js Core Modules

We can create a new node app directory and name this anything we want and create a .js file within it. In the below example we are going to explore the File System module function called `writeFileSync()`. This function takes in two arguments both being strings. The first argument is name of the file and the second is the data/content to write in that file.

notes-app > app.js:

```
1. fs.writeFileSync('notes.txt', 'This file was created by node.js')
```

We can use terminal to execute node to run our script. However, we will run into a problem as this is not going to work and throw an error message of fs is not defined. The Files System module has to be required by us in our script we are using it in. Before we use the File System module, we need to import it in our script.

This is done using the `require` function that node provides which is at the very core of the module system. We use `require` to load in other things whether it's a core node module, another file we created or a npm module we chosen to install and use in our projects. The `require` function takes in a single string.

notes-app > app.js:

```
1. const fs = require('fs')  
2. fs.writeFileSync('notes.txt', 'This file was created by node.js')
```

When we are looking to load in a core node module, we just provide the name of the module. For the File System module this is 'fs'. The require function returns all of the things from that module and we have to store this in a variable.

In the above example, we created a const called fs (this could be named anything we want) and its value is going to come from the return value of calling require function (i.e. the node module). We then call on the fs variable followed by the method of writeFileSync to write some text to the notes.txt file.

To run this script we can use the terminal or VS Code integrated terminal and cd into the directory containing our node script. We would call on the node command followed by the name of the script we wish to run:

```
:~.../notes-app$ node app.js
```

When we run the code we will see that we are brought back to the command prompt asking us to do something else, but we will notice in the tree view for our directory, we have a brand new file called notes.txt which is the file created by the script that is now sitting along side of our app.js file.

The writeFile and writeFileSync methods are responsible for writing data to a file and if the file does not exist it will be created. If the file does exist, its text content will be overridden with the new text content.

We have now learnt how to load in our first core node module and we used it to do something interesting. To know how to load in a core module and what variable naming convention to use or even if the module requires to be loaded in at all, the node documentation is going to be our best friend. We can see the File System module documentation on the following link. Scrolling past the table of content, one of the first things we are going to see is the code to use in order to load the core module i.e. what node calls the module, the common used variable name and whether it needs to be imported in:

<https://nodejs.org/api/fs.html>

Important Note: we do not need to stick to the common variable name; however, it is advisable/best practice to do so, as the naming convention allows developers who are working with our project know which core modules they are working with when importing the modules.

CHALLENGE:

Append a message to the existing notes.txt file using appendFileSync.

1. Use appendFileSync to append to the file
2. Run the script
3. Check your work by opening the file and viewing the appended text.

SOLUTION:

```
notes-app > app.js:
```

1. `const fs = require('fs')`
2. `fs.appendFileSync('notes.txt', '- This is a appended text.')`

```
:~.../notes-app$ node app.js
```

Importing Your Own Files

When we pass a file to node, only that file executes. This would mean that we would need to put all of our code in a single file if we want to run the code. However, this is not ideal especially if our application grows larger and more complex. This also makes it difficult to expand and/or maintain our application code.

Therefore, we would ideally create our project using multiple files so that we can stay organised and our application is modular. For example, we can define a function in one file and then require it in another file in order to use that function.

To import our own files into node we have to require it in order for the file to get loaded in when we run node. We would continue to use the require function and pass in a single string value. The string value we pass in is a relative path from the file we are loading it in from.

Below is an example of loading in our own file called `utils.js` which has a single function that logs to the console the name of the file. We will import this file into our `app.js` and call on this function:

```
notes-app > utils.js:
```

```
1. console.log('utils.js')
```

```
notes-app > app.js:
```

```
1. require('./utils.js')  
2. const name = 'John Doe'  
3. console.log(name)
```

The `./` will take us to the relative path from `app.js` which is the `notes-app` directory. We can then select the `utils.js` file that is located within that directory in order to require the file and import it into our `app.js` file.

When we load in a JavaScript file, it will execute that file when we run our node command on our `app.js` script. Therefore in the console we should see printed:

```
utils.js
```

```
John Doe
```

The `utils.js` file will be printed first because this code runs as soon as we require/load it in our `app.js` file and then the `name` variable is printed as it is further down in the code. We now have a simple application that

takes advantage of multiple files.

We can take the above example further and try to define variables within our `utils.js` file and try to use the variable within our `app.js` file:

```
notes-app > utils.js:
```

```
1.  const name = 'John Doe'
```

```
notes-app > app.js:
```

```
1.  require('./utils.js')
```

```
2.  console.log(name)
```

If we were to run `app.js` script, we will see the error of *'name is not defined'*. This is one very important aspect of the node module system. All of our files, which we can refer to as modules, have their own scope. Therefore, `app.js` has its own scope with its own variables and `utils.js` has its own scope with its own variables. The `app.js` file cannot access the variables from `utils.js` even though it was loaded in with `require`.

To share the variables and functions within the `utils.js`, we need to explicitly export all of the code within the file to be able to share with the outside world i.e. share outside of its own scope. To do this we take advantage of another aspect of the module system which is called `module.export`. This is where we can define all of the things the file should share with other files.

Using the above example and taking it one step further, is to try and define variables within our `utils.js` file and export that variable to use it within our `app.js` file:

`notes-app > utils.js:`

1. `const name = 'John Doe'`
2. `module.exports = name`

`notes-app > app.js:`

1. `const firstName = require('./utils.js')`
2. `console.log(firstName)`

In the above example we have now exported one variable which is a string. We will later learn how to share an object which has a bunch of different methods on it allowing us to export a whole bunch of things. So in `utils.js` we defined a variable and we exported that variable and other files can now take advantage of the variable.

Whatever we assign to `module.exports` is available as the return value from when we require the file. Therefore, when we require `utils.js` in our `app.js`, that return value is whatever we assigned in the `utils.js`, which was the string *'John Doe'* that is stored on the `name` variable.

Within our `app.js` file we can create a variable and call it whatever we want including *'name'* - variables

within the app.js file is independent from variables from other files because they have different scopes. We assigned this variable value to the require function, which returns the module.export variable value assigned from the utils.js file.

Our application is now back to a working state whereby we can run the app.js script and have the console log the name variable from the utils.js file.

```
:~.../notes-app$ node app.js  
John Doe
```

We can also import functions in the same fashion as we did with variables:

notes-app > utils.js:

1. `const add = function(a, b) { a + b }`
2. `module.exports = add`

notes-app > app.js:

1. `const add = require('./utils.js')`
2. `const sum = add(1, 2)`
2. `console.log(sum)`

The sum function will return 3 as the value because $1 + 2 = 3$.

```
:~.../notes-app$ node app.js
```

```
3
```

CHALLENGE:

Define and use a function in a new file.

1. Create a new file called notes.js
2. Create getNotes function that returns "Your notes..."
3. Export getNotes function
4. From app.js load in and call the function printing the message to the console

SOLUTION:

notes-app > notes.js:

1. `const getNotes = function() { return 'Your notes...' }`
2. `module.exports = getNotes`

notes-app > app.js:

1. `const getNotes = require('./notes.js')`
2. `const msg = getNotes()`
3. `console.log(msg)`

Importing NPM Modules

We can use the Node Module System to load in npm packages which will allow us to take advantage of all of the npm packages inside of our node applications. NPM modules allows us to install code written by other developers so that we do not need to reinvent/recreate the wheel from scratch.

There are always things that every applications out there needs to do such as validating data such as emails and maybe even sending an email. These are core functionality which is not specific to what our application does for our users. So if we use npm modules to solve common problems (which is the standard in the node community) then we can spend our development time focusing on features that makes our app unique.

When we installed node on our machine, we also got the npm program installed on our machine. This gives us access to everything at <https://www.npmjs.com>. We can use npm terminal commands such as the one below which shows the npm version installed on our machine:

```
:~.../notes-app$ npm -v
```

Before we can use npm modules within our scripts, we have to take two very important steps.

1. We have to initialise npm in our project
2. We have to install all of the modules we actually want to use

To initialise npm in our project, we need to run a single command in the project root directory:

```
:~.../notes-app$ npm init
```

This command is going to initialise npm in our project and create a single configuration file that we can use to manage all of the dependencies from the npm website that we want to install. The above command will ask a few question to configure the configuration file such as package name (i.e. the folder/project name), version, description, etc. We can type in our own values to overwrite the default values to the questions. Once we complete all of the questions and execute the command this will the create a package.json file in the root directory of our projects. This is the configuration file which contains all the npm module dependencies and scripts for our project.

We can go onto <https://www.npmjs.com> to look for packages/modules we wish to use in our project. Once we find a package that we want to install we can use the npm terminal command within our project directory. The below example demonstrates installing the validator module package:

```
:~.../notes-app$ npm install validator@11.1.0
```

```
:~.../notes-app$ npm i validator@11.1.0
```

When we run the command, we will notice that two things has occurred. Firstly, we would now have a package-lock.json file and secondly, we have a new node_modules directory (folder) in our project root.

The `node_modules` is a folder that contains all of the code for the dependencies that we have installed. In the above example, this would have a subfolder called `validator` which contains all of the code for that dependency package we installed using `npm`. We should never go into the `node_modules` folder to manually change these files as this is simply a package management. When working with `node_modules`, it is going to get generated and edited when we run the `npm install` command. The `npm` maintains this directory.

The `package-lock.json` file contains extra information for `npm` to make `npm` a bit faster and secure and again is another file that we do not modify. This file lists out the exact versions of all of our dependencies as well as where they were fetched from and the hash masking sure that we are getting the exact code that we got previously, if we were to install a dependency again. This file will be maintained by `npm`.

When we ran the above command, the package is also added as a dependency within our `package.json` file which lists the package name and the version installed.

Now that we have our dependency installed, we can now load in the dependencies within our application's files using `require` and take advantages of the functionality it provides.

`notes-app > app.js:`

```
1.  const validator = require('validator')
```


To load in npm modules we list out the package names similar to node core modules. Require is going to return all of the stuff that the validator package provides us. We can create a variable and assign it to the require function which gets its contents from the package it is returning.

When it comes to figuring out how to use a given npm package, this is when we have to turn to the documentation of that package to understand how it was intended to be used. The below example provides an example of using the validator package and one of its many functions in our app.js file:

```
notes-app > app.js:
```

1. `const validator = require('validator')`
2. `console.log(validator.isEmail('test@email.com'))`

```
:~.../notes-app$ node app.js  
true
```

Is we run our application, the validator isEmail function will validate whether the given string is actually an email. In the above example, this equated to true.

As mentioned above, the node_modules is not a file that we should be manually editing. This is because when we use the npm command again, our edits are going to be overwritten. We can actually recreate this

directory from scratch using npm based off of the contents of package.json and package-lock.json.

The node_modules folder is necessary for our app to run if we are using npm modules as dependencies for our application code. To install the node_module folder with all the dependencies, we can run a simple npm command:

```
:~.../notes-app$ npm install
```

This is going to look at the package.json and package-lock.json to determine which dependencies and versions our application is using in order to recreate the node_module folder from scratch based on the contents of these two .json files (*the .json extension stands for JSON Object Notation*).

This is useful for when sharing your code as the node_modules folder can get really huge in file size and it would be easier to regenerate the folder using the npm install command using the two package files. Our application will be back to its working state as we would have all the necessary files in order to make our application work again.

CHALLENGE:

Install and use the chalk library.

1. Install version 2.4.1 of chalk
2. Load chalk into app.js
3. Use it to print the string "Success!" to the console in green
4. Test your work
5. Bonus: use chalk documentation to play around with other styles e.g. make text bold and inverse.

SOLUTION:

```
:~.../notes-app$ npm install chalk@2.4.1
```

notes-app > app.js:

1. `const chalk = require('chalk')`
2. `console.log(chalk.green('Success!'))`
3. `console.log(chalk.blue.bgYellow.bold.inverse('Hello World!'))`

```
:~.../notes-app$ node app.js
```

Global NPM Modules and nodemon

Installing Global npm module packages will allow us to get new commands that we can execute from the terminal. So far we have learnt how to install npm packages locally, this is where we install the dependencies explicitly into our project directory and these appear in the package.json and package-lock.json file as dependencies.

We can install a npm package called nodemon globally which is a nice utility when working with node. This is going to allow us to run our application and automatically restart the app whenever the app code changes i.e. whenever we save any changes to our app code file. This would mean that we would not have to constantly switch to the terminal and rerun the same command over and over again to test our code.

The documentation for nodemon can be found on <https://www.npmjs.com/package/nodemon> which we can read to understand more about the package and how to use it.

When installing packages globally, the command is exactly the same as if we were installing it locally, however, there is one slight difference - we add the -g flag to indicate installing the package to be global.

```
:~$ npm install nodemon@1.19.1 -g
```

```
:~$ sudo npm install nodemon@1.19.1 -g
```

On linux and MacOS, we can use the sudo in the command to install the package as an administrator to avoid errors in the command.

It is advisable to install packages locally rather than globally as you can tend to run into errors installing packages globally depending on the OS environment. Furthermore, packages are updated regularly and this would mean older and newer projects would have different dependency packages at different versions that are used in development which could also create problems for your development environment and your code running especially if there are syntax differences or deprecation in the dependency packages used. Therefore, it is more advisable to either install packages locally or have multiple virtualised development environments whereby we can install packages globally.

To check that we installed nodemon correctly globally, we can run the following command:

```
:~$ nodemon -v
```

If installed locally we can use the following command to check the version installed:

```
:~.../notes-app$ ./node_modules/nodemon/bin/nodemon.js -v
```

To start using the nodemon to automatically run our application script we would run the following command:

```
:~.../notes-app$ nodemon app.js
```

We would use the following command if we were running it locally:

```
:~.../notes-app$ ./node_modules/nodemon/bin/nodemon.js app.js
```

We will notice that when we run nodemon and our script has run, it is not bringing us back to the command prompt even though our application has finished, this is because the nodemon process is still executing. If we make any changes it will automatically run our code when we save the file i.e. the process is now listening for any saved changes to our file.

The nodemon package is a great utility to improve our overall developer experience when developing node applications.

To terminate the nodemon process, within the terminal running nodemon, we can press control and c on our keyboard and this will end the process from running.

Section 3

FILE SYSTEM & COMMAND LINE INTERFACE ARGUMENTS

Getting Input From Users

Input from the user is essential to create anything meaningful. This allows interaction between the user and the application.

We can get input from the user either from a command line argument or from a client such as a browser. We are going to focus on the fundamentals of getting input from the user with command line arguments.

We can run our app using the node command in the terminal. However, we can also pass in additional information that our application can choose to use to do something dynamic for example print a greeting with the name inside of the message:

```
:~.../notes-app$ node app.js "John Doe"
```

The above command will run our script as normal, but does not do anything with the additional information

we passed in. The question follows, where exactly do we access those command line arguments? The answer is that on the global process object there is a property that allows us to access all of the command line arguments passed into our app.

Process is an object that contains many methods and properties and the property we are interested in is the `argv` (*argument vector*) property. This property is an array that contains all of the arguments provided.

```
notes-app > app.js:
```

```
1. console.log(process.argv)
```

If we now rerun our script with John Doe appended, we should now receive a new array of the command line arguments:

```
:~.../notes-app$ node app.js "John Doe"
```

```
[..., ..., 'John Doe']
```

There are always 2 strings provided by the `argv` property, the first one is the path to the node.js executable on our machine and the second is the path to our `app.js` file. These paths are going to be slightly different depending on where the file lives on the machine. The third value and all values after are the arguments the user provides which we can take advantage of the string in our application to do something meaningful.

We can extract the individual values from the array by using JavaScript bracket notation to retrieve the value from its index. JavaScript uses zero indexing whereby 0 is the first item within the array.

```
notes-app > app.js:
```

```
1. console.log(process.argv[2])
```

```
:~.../notes-app$ node app.js "John Doe"
```

```
John Doe
```

We can use the command line argument to check for the value and then perform a certain code in our application. For example, we can take the first argument as a command and look at the value passed in such as add and then run a particular code to add a note:

```
notes-app > app.js:
```

```
1. const command = process.argv[2]
2. if (command === 'add') { console.log('Adding note!')
3. } else if (command === 'remove') { console.log('Adding note!') }
```

```
:~.../notes-app$ node app.js add
```

```
Adding note!
```

We now have a way to get the users input and perform some form of meaningful action. Now that we can get the command from the user, we need to get a command line options so that the user can provide additional information for example the note title or note body they wish to add/remove.

```
:~.../notes-app$ node app.js add --title="This is my title"
```

If we were to view the `process.argv` we will notice that this command line option will not be parsed. We would see in the console `--title="This is my title"` printed. We would have to figure out how to parse the title and then figure out whether the `--title` option was provided and if so, we have to get that value.

If we wrote our own code for parsing the string, we would have to write tests and maintain this code and this code does not do anything unique to our application. This would be a best case scenario where we would look for a npm package that can take care of parsing for us.

We have now found a way of our users interacting with our application and getting their input via the command line arguments.

Argument Parsing with Yargs

Node does not provide any argument parsing and it is a very bare bones utility allowing us to access the

raw elements. So when we pass in an argument such as `--title="This is my title"` option, it was not parsed in a way that is particularly useful. We would need to write some code to extract the key value pair.

Most node.js application use command line arguments in some way and there are a many of great npm packages that make it easy to setup our commands and options as we want. Yargs is a widely used and tested utility package that will help us to parse command line arguments (<https://www.npmjs.com/package/yargs>). To install Yargs in our project we can run the following command:

```
:~.../notes-app$ npm install yargs@13.3.0
```

Once installed, we can use yargs in our project to parse command line arguments. To use yargs we first need to require the npm package. When using require, it is good practice to use a pattern i.e. require core modules first, then npm packages and finally our own files.

notes-app > app.js:

1. `const yargs = require('yargs')`
2. `console.log(yargs.argv)`

We can compare and contrast the yargs command with nodes `process.argv` command by running the `app.js` script and passing in arguments, in order to see the difference between the two approach when the parsing command line arguments.

```
:~.../notes-app$ node app.js  
{ _: [], '$0': 'app.js' }
```

We would notice by running the first example, yargs almost provides nothing compared to the default. In the yargs output we have an object with two properties: the first is an underscore property which will get populated with various arguments and the second is the \$0 property which has the value of the file name we executed.

```
:~.../notes-app$ node app.js add --title="This is my title"  
{ _: ['add'], title: 'This is my title', '$0': 'app.js' }
```

Running the second example, we can see yargs provides us with a much more useful parsed object. Our commands appears under the underscore property and we then have an actual options property on this object that contains the string value (*i.e. title: 'This is my title'*). Yargs has done the heavy lifting of parsing our options and putting them on the object so that they are easy to access. We can access the title property to do something with it such as creating a new note and save it to our data store.

This is the very bare bones way of using yargs, since we have not configured it to do anything special. By default we get some very useful functionality behaviour. If we run the below command, we can see the version of yargs (by default this is 1.0.0)

```
:~.../notes-app$ node app.js --v  
1.0.0
```

We can change the version by accessing yargs and calling the version method. This takes in a string as its one and only argument as the new version number. This allows us to specify a different version number than the default 1.0.0 version.

```
notes-app > app.js:
```

1. `const yargs = require('yargs')`
2. `yargs.version('1.1.0')`

```
:~.../notes-app$ node app.js --v  
1.1.0
```

Yargs comes with a lot of useful parsing and features of its own but it can be configured to do anything we want, for example we can setup distinct commands, display help options for the commands, create options to pass in data, etc. all of which is supported by yargs.

The `yargs command()` function takes in an object, which is the options object, where we can customise how this command should work. The first command parameter is the name of the command. The second describe parameter allows us to add a description of what the command is suppose to do. We can setup a

third handler property which is the code that is actually going to run when someone uses the command described in the first parameter. Below is a example yargs command code:

notes-app > app.js:

```
1. const yargs = require('yargs')
2. yargs.command({
3.   command: 'add',
4.   describe: 'Add a new note.',
5.   handler: function() {
6.     console.log('Adding a new note!')
7.   }
8. })
9. console.log(yargs.argv)
```

If we use the yargs --help command, this will now add a new Commands line displaying all of the different commands we have setup in yargs along with the help description of the commands.

```
:~.../notes-app$ node app.js --help
```

Commands:

```
  app.js add  Add a new note.
```

Options: ...

We can now also use the add command within our command line argument:

```
:~.../notes-app$ node app.js add  
Adding a new note!
```

Now that we know how to use the yargs utility package to create our command line argument options we now need a way to configure options for those commands. To setup a configuration option, there is another property we can add to our configuration object that we pass to our yargs command function called builder. The builder value is an object where we can define all of the options we want the given command to support. Each option value is also a object where we can customise how the option works:

notes-app > app.js:

```
1.  const yargs = require('yargs')  
2.  yargs.command({  
3.    command: 'add',  
4.    describe: 'Add a new note.',  
5.    builder: {  
6.      title: { describe: 'Note title.', demandOption:true, type: 'string' }  
7.    },  
8.    handler: function(argv) {  
9.      console.log('Adding a new note!', argv)  
10.    }  
11.  })
```

12. yargs.parse()

The option property value takes in a describe property which describes the option. With our option defined we can now access that in the command handler and we get access via the first argument provided in our function. If we now run the following command in terminal:

```
:~.../notes-app$ node app.js add --title="This is my title"  
Adding a new note! { _: ['add'], title: 'This is my title', '$0': 'app.js' }
```

We will notice that we get the console logging the text along with yarg's version of argv inside of the handler. We also have access to the options and its values for example argv.title which we can now use to actually define a note.

One thing to note is that the options are not required and we can run our terminal command without using any of the options and things will continue to work. It is up to us as the developer to decide whether or not a given option should be required. If we want a option to be required we can set this up as well using the demandOption property. By default this property is set to false. If we set this to true, this would mean that we must provide the option in order for the command to function correctly.

If we now try to run the command without the required option this will print an error message of missing required argument: ... listing out the missing command options. It will also display all the available options

and which ones are required. If we run the command with the required option but without a value:

```
:~.../notes-app$ node app.js add --title  
Adding a new note! { _: ['add'], title: true, '$0': 'app.js' }
```

We will notice that the value for title will default to a boolean value. To set the argv --title option property to be a string value, we can enforce that by setting the type property on our option object configuration. We can set this to one of the supported types as a value for example boolean, string, number, array, etc.

If we now run the following command without a value, at least the value for title will now be a string, even if this is an empty string value.

```
:~.../notes-app$ node app.js add --title  
Adding a new note! { _: ['add'], title: '', '$0': 'app.js' }
```

In order to view the output of yarg we need to `console.log(yargs.argv)` at the end of our code else we would not see any output. If we do not need to access the `yargs.argv`, which is less than ideal, we can parse it using the `yargs.parse()` function.

We can have access to our options property on the argv for example within the handler we can write:

```
9.      console.log('Title: ' + argv.title)
```

This will print whatever value the user passes in for the title option that was setup.

We now know how to setup yargs to create our own commands and options for the user to interact with our node application via the command line arguments and we can use this utility to parse the arguments in order for our application to do something meaningful.

CHALLENGE:

Add new command using Yargs.

1. Setup command to support "read" command (print placeholder message for now)
2. Test your work by running command and ensure correct output in the terminal

SOLUTION:

notes-app > app.js:

```
1. const yargs = require('yargs')
2. yargs.command({
3.   command: 'read',
4.   describe: 'Read a note.',
5.   handler: function() {
6.     console.log('Reading a note!')
7.   }
8. })
```

```
9. console.log(yargs.argv)
```

```
:~.../notes-app$ node app.js read  
Reading a note!
```

CHALLENGE:

Add an option to Yargs.

1. Setup a body option for the add command
2. Configure the option with a description, make it required and set the type to string
3. Log the body value in the handler function
4. Test your work!

SOLUTION:

notes-app > app.js:

1. `const yargs = require('yargs')`
2. `yargs.command({`
3. `command: 'add',`
4. `describe: 'Add a note.',`
5. `builder: {`

```
        body: {
            describe: 'Note body',
            demandOption: true,
            type: 'string'
        },
5.     handler: function(argv) {
6.         console.log('Body: ' + argv.body)
7.     }
8. })
9. yargs.parse( )
```

```
:~.../notes-app$ node app.js add --body='This is my body.'
Body: This is my body.
```

Storing Data with JSON

We can use the fs core module to save data to the file system. We can save the data in the JSON data format which structures the data as JavaScript Objects. JSON has native support in JavaScript and is very easy to learn since it is essentially arrays and objects with various properties. We can model our data in JSON and store the data in a .json file type which we can access through our application. If we access to these arrays and objects, then we have an array of items like we would in any database.

The fs core module only knows how to work with string data. In order to take a JavaScript object and store it as a string representation, we would use the `JSON.stringify()` method:

```
playground > json.js
```

```
1.  const book = {  
2.      title: 'Ego is the enemy',  
3.      author: 'Ryan Holiday'  
4.  }  
5.  const bookJSON = JSON.stringify(book)  
6.  console.log(bookJSON)
```

The `stringfy()` method takes in an object, array or any value and it returns the JSON string representation. In the above example, we can pass in the book variable object and convert it to a JSON string:

```
:~.../playground$ node json.js  
{ "title": "Ego is the enemy", "author": "Ryan Holiday" }
```

We will notice that the JSON string data looks exactly like a JavaScript object with the only difference of all properties name and values wrapped in double quotes. The JSON data is a string and not an object and therefore we cannot access its properties like we would with a JavaScript object i.e. `bookJSON.title` does not exist unlike the `book.title` object.

To convert a JSON data back into a JavaScript object we would use the `JSON.parse()` method:

```
playground > json.js
```

```
...
```

```
7.  parsedData = JSON.parse(bookJSON)
```

```
8.  console.log(parsedData.author)
```

This method takes in the JSON string and converts it into a JavaScript object which we can then have access to the object properties:

```
:~.../playground$ node json.js
```

```
Ryan Holiday
```

These are the two core methods with working with JSON data. We can integrate this with the file system core module to store and retrieve data from our application using JSON.

```
playground > json.js
```

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

```
2.  const book = { title: 'Ego is the enemy', author: 'Ryan Holiday' }
```

```
3.  const bookJSON = JSON.stringify(book)
```

```
4.  fs.writeFileSync('book-json.json', bookJSON)
```

We can use the `fs.writeFileSync` function to create a new file in our file system called `book-json` which will have the file extension of `.json` as this will be a JSON file containing JSON data. The second argument we pass is the data to write to the file which is going to be the `bookJSON` data. If we now run the above code

this should create a new file in the same directory as the script we ran called book-json.json and this file would contain our JSON string data:

```
:~.../playground$ node json.js
```

We now have a way to store data with JSON and the file store core module using our node application. This is now a data we can load in at a later time and use within our node application.

To load in a data with the file system core module, we would use the fs.readFileSync method and pass in a single argument, which is a string of the file name we are trying to read:

```
playground > json.js
```

```
1.  const fs = require('fs')
2.  const dataBuffer = fs.readFileSync('book-json.json')
3.  const dataJSON = dataBuffer.toString( )
4.  const data = JSON.parse(dataJSON)
```

We can grab the content of the returned value from the fs.readFileSync which is going to contain the contents of the file. We can create a variable that will store the content that came back from the method call. What comes back from the method is not a string, but actually a buffer which is a way for node.js to represent binary data. We can use the toString() method to convert the buffer into a string as we would expect to see the returned contents. We can finally parse the string into a JavaScript Object which we can have access to its properties and values.

CHALLENGE:

Work with JSON and the File System.

1. Load and parse the JSON data

data.json:

```
1. {"name":"Andrew","planet":"Earth","age":28}
```

2. Change the name and age property using your info

3. Stringify the changed object and overwrite the original data

4. Test your work by viewing data in the JSON file

SOLUTION:

playground > jsonChallenge.js:

```
1. const fs = require('fs')
2. const dataBuffer = fs.readFileSync('data.json')
3. const dataJSON = dataBuffer.toString( )
4. const user = JSON.parse(dataJSON)
5. user.name = "Beth"
6. user.age = 21
7. const userJSON = JSON.stringify(user)
8. fs.writeFileSync('data.json', userJSON)
```

```
:~.../playground$ node jsonChallenge.js
```

ES6 Arrow Functions

To understand the ES6 arrow functions, we first need to compare it with a simple ES5 function. Below is a regular function that takes a number and returns the squares of the given number using the ES5 syntax:

```
playground > arrow-function.js:
```

1. `const square = function(x) { return x * x }`
2. `console.log(square(3))`

```
:~.../playground$ node arrow-function.js  
9
```

The square of 3 will return 9 ($3 \times 3 = 9$) and therefore is printed to the terminal when we run the script. Below is the equivalent function but using the new ES6 arrow function syntax:

```
playground > arrow-function.js:
```

1. `const square = (x) => { return x * x }`
2. `console.log(square(3))`

```
:~.../playground$ node arrow-function.js  
9
```

This too will return 9 in the terminal, and the function continues to operate using the ES6 syntax. The first difference we can see is that we do not use the function keyword, we actually start with the argument list.

The next step is to put the arrow function by using the `=>` sign/symbols to create the arrow. Finally, we can setup the curly brackets and define the code block for what the function is suppose to do. Now that we know how to setup a basic arrow function, we can now explore its hidden features that make it a tool worth using:

The first feature is its shorthand syntax. With a lot of the functions we end up writing in JavaScript and node.js they end up being pretty simple. We take in some arguments and then immediately return some result. If our function was just going to have a single statement which returned something, we can put that something right after the arrow function:

```
playground > arrow-function.js:
```

1. `const square = (x) => x * x`
2. `console.log(square(3))`

```
:~.../playground$ node arrow-function.js
```

```
9
```

There is no need for both the curly brackets or the return keywords. Whatever we put after the arrow function is going to be implicitly returned. Now, in reality we would not be able to use this shorthand syntax for every arrow function that we create, but we can use it for simpler function which immediately returns some sort of value. If we have longer code blocks such as using if statements, we would use the long form version where we setup curly brackets and add as many lines of code as needed.

We can now take this up a gear and look at arrow functions work in the context of methods, so arrow functions as properties on an object. Below is a example of an event object with two properties. The second property calls on a function using the ES5 function syntax:

```
playground > arrow-function.js:
```

```
1.  const event = {  
2.      name: 'Birthday Party',  
3.      printGuestList: function( ) { console.log('Guest list for ' + this.name) }  
4.  }  
5.  event.printGuestList( )
```

```
:~.../playground$ node arrow-function.js
```

```
Guest list for Birthday Party
```

The function `printGuestList` will start with a static text of *"Guest list for"* followed by the actual event name i.e. *"Birthday Party"* - we can read the `name` property value. We know that with our methods, our functions as object properties, we have access to the original object via the `this` binding. Therefore, this is a reference to our object, which means we can access properties on the `this` keyword such as `this.name`. We can use the `printGuestList` property method on the event object to print the text to the console.

If we now view the above code but written using the ES6 arrow function syntax, we can view the difference between the two syntax and the second hidden feature provided by ES6 arrow functions:

```
playground > arrow-function.js:
```

```
1.  const event = {  
2.      name: 'Birthday Party',  
3.      printGuestList: ( ) => { console.log('Guest list for ' + this.name) }  
4.  }  
5.  event.printGuestList( )
```

```
:~.../playground$ node arrow-function.js
```

```
Guest list for Birthday Party
```

The above is the valid syntax but we will notice that if we run the code, we do not get the results that we might be expecting. The arrow function is unable to find the name property and this is because arrow functions do not bind their own *"this"* value, which means we do not have access to the `this` keyword as a reference to the object because of the fact we are using the arrow function.

Arrow functions are not well suited for method properties, that are functions, when we want to access *"this"*. In this case it would be best to use a standard ES5 function. This does not mean we are stuck with the ES5 syntax. There is actually a ES6 method shorthand that allows us to use a shorter more concise syntax while still having access to standard function features like the `this` binding:

```
playground > arrow-function.js:
```

```
1.  const event = {  
2.      name: 'Birthday Party',  
3.      printGuestList( ) { console.log('Guest list for ' + this.name) }  
4.  }  
5.  event.printGuestList( )
```

```
:~.../playground$ node arrow-function.js
```

```
Guest list for undefined
```

We remove the colon and function keyword keyword so that it goes from the method property name right into the arguments list followed by the function body contained in the curly brackets. This is now a standard function which does have the this binding and it is not an arrow function. This is using an alternative syntax available to us when we are setting up methods on objects. If we run the script, we would now have a working code block using this concise syntax.

Why do arrow functions avoid their own this binding? Below is an example of the same function but we now have a guestList property on our event object:

```
playground > arrow-function.js:
```

```
1.  const event = {  
2.      name: 'Birthday Party',  
3.      guestList: ['Alan', 'Beth', 'Carl'],
```

```
4.     printGuestList( ) {  
5.         console.log('Guest list for ' + this.name),  
6.         this.guestList.forEach(function( guest) { guest + ' is attending ' + this.name } )  
7.     }  
8. }  
9. event.printGuestList( )
```

```
:~.../playground$ node arrow-function.js
```

Guest list for Birthday Party

Alan is attending undefined

Beth is attending undefined

Carl is attending undefined

If we now try to print that guestList for each individual guest, we can use the forEach method on the array using the this binding. The forEach method takes in a function which gets called one time for every array item within the list i.e. one for each guest. We get access to the guest via the first argument, which we could name as anything we want. We can use console.log function to print a message for each guest and we can access the name using the guest argument variable. We can see that things are not working as expected as the event name is undefined. This has to do with the this binding. Standard functions are going to have their own binding (i.e. the forEach function has its own binding), which is a problem because we do not want the forEach function to have its own this binding. We want to be able to access the this binding from the parent function which is printGuestList(). In the past there were all sorts of work around for this for

example we would have created a const variable of *"that"* and setting it equal to *"this"* within the parent function, so essentially creating a reference that we can access later, and in the sub function we would use the *"that"* variable, e.g. *that.name*, to access the object property from the parent which would work:

playground > arrow-function.js:

```
1.  const event = {  
2.      name: 'Birthday Party',  
3.      guestList: ['Alan', 'Beth', 'Carl'],  
4.      printGuestList( ) {  
5.          that = this,  
6.          console.log('Guest list for ' + this.name),  
7.          this.guestList.forEach(function( guest) { guest + ' is attending ' + this.name } )  
8.      }  
9.  }  
10. event.printGuestList( )
```

```
:~.../playground$ node arrow-function.js
```

```
Guest list for Birthday Party
```

```
Alan is attending Birthday Party
```

```
Beth is attending Birthday Party
```

```
Carl is attending Birthday Party
```

Although the code will work using the workaround; however, this is not ideal and this is where ES6 arrow functions solves this problem as seen below:

playground > arrow-function.js:

```
1.  const event = {
2.      name: 'Birthday Party',
3.      guestList: ['Alan', 'Beth', 'Carl'],
4.      printGuestList( ) {
5.          console.log('Guest list for ' + this.name),
6.          this.guestList.forEach(( guest) => { guest + ' is attending ' + this.name } )
7.      }
8.  }
9.  event.printGuestList( )
```

```
:~.../playground$ node arrow-function.js
```

Guest list for Birthday Party

Alan is attending Birthday Party

Beth is attending Birthday Party

Carl is attending Birthday Party

The solution is turning the ForEach regular function into an arrow function. If we now run the script, we will see the code block running as expected. Arrow functions do not bind their own *"this"* value. They access the *"this"* value in the context which they are created, which in the above case is inside of the printGuestList

function. This means that we have access to the `this.name` which is pointing to the property up above within the object.

To conclude, there are three key aspects we have learnt:

1. There is an alternative syntax using ES6 arrow function
2. With arrow functions we have a shorthand syntax for returning immediate values
3. Arrow functions do not bind their own "this" value

Therefore, arrow functions are poor candidates for object methods (i.e. functions on an object) but they are great candidates for everything else. Moving forward, we are never going to use a function where we have the `function` keyword. We are either going to use arrow functions or when necessary we will use the ES6 method definition syntax. This understanding will make us better JavaScript developers.

Section 4

DEBUGGING NODE.JS

Debugging Node.js

When we run our program, things do not always work as expected. We may introduce a typo into our code causing the program to crash or we may misuse a function causing strange and unexpected behaviour.

The goal in this section is to have the tools needed to actually fix these problems. Therefore, debugging and inspecting the application as it is running is an important skill so that we can figure out exactly where the program is going wrong and why. This is going to give the information needed to fix the mistake in the code and get back to a working application.

As we make mistakes it is only going to get easier to fix them over time. What may have taken 10 minutes to debug will only take 10 or 20 seconds to fix. You will notice and start to see a lot of the same issues and patterns over time. The real goal is to recover from the mistake quickly to get back on track and be productive.

We will go through debugging strategies as well as exploring debugging tools provided by node to make it much easier to fix our broken application. We will explore the node debugger which integrates directly with the chrome developer tools, providing us with a nice GUI for debugging our backend applications.

In general, there are two types of errors we can run into when working with node.js and these are:

1. Explicit error message e.g. typos in the class
2. Logical errors i.e. no syntax error or error message

With logical errors there will be no error message, and so we will need to first figure out where it is and then secondly adjust to make the application code actually work the way we envision it.

We can use the same debugging strategies for both type of errors. Firstly, we can explore the basic tools that node provides:

1. The console.log command

This allows us to view the value of variables when we run a script which allows us to figure out what is happening with our application. This is the most basic tool available to us for debugging our application.

2. The node debugger

This is the node's built in debugging tool which integrates with V8 and the Chrome browser. The debugger just like console.log needs to be added at a specific point in our application code. Using the keyword debugger is going to stop the application at a point in time whereby we can use the developer tools to

look at any values we want. This gives us more flexibility, because unlike `console.log` where we have to log every variable we want to see, with debugger we can just add the debugger statement one time and once we get to the dev tools, we can view everything of our application from where it stops at the debugger.

When we have debuggers in our application, they are not going to pause the program by default, we have to run node with a special option to get that done. This special option is called `inspect`:

```
:~.../notes-app$ node inspect app.js add --title="t" --body="b"  
< Debugger listening on ws://127.0.0.1:9229/XXXXXXXXXXXXXX
```

We would now get a much different output. Node is letting us know that a debugger is now up and running and which url it is listening on. We can go into chrome following the url to actually inspect our application. We can go into chrome and navigate to <chrome://inspect> which will bring up the Chrome debugger which is using the built in V8 debugger tools. Chrome is the only browser we can inspect our node applications because we require the V8 engine's debugger tools.

Under Remote target we should see two targets i.e. one at the port and another at the shorthand localhost (both are the exact same process). If you see nothing under Remote Target, it is likely you have a misconfiguration. If we click on the configure button, we should see two values:

localhost:9229 and 127.0.0.1:9229

If we do not have these values, we can add them here into the configuration and should be able to see our

two targets. From here we can actually inspect our application and we can pause at that point in time that we put the debugger statement and view all of our application variables and values. In the Target, we would click on the inspect link which will open a new instance/window of the debugger tools for debugging our Node.js application.

The contents and sources tab are the two most important tabs that we would be using to debug our application. Within the sources tab we have in the middle our file code, exactly how we have it inside of our text editor with one very important difference; we have a function that wraps our whole node.js script.

In other programming languages we might have a main function that we define. When we run the program, that is the function that starts up our application. With Node.js we do not have that as it runs our entire script from top to bottom. Therefore, to make node run our program through V8, our code is wrapped in this function which can be considered as our main function. This function provides to our application a few different values we can access such as exports, require, module, __filename and __dirname variables.

On the left hand side we can add our project folder to the developer tools by clicking on the +Add folder to workspace button under the Filesystem. This will change the middle screen slightly as we no longer will have the wrapper function.

If we press the escape key we can toggle the console at the bottom of the debugger to have both source

and console on the same window. We can use the console to run various commands. It is important to note that at this point in time, not a single line of code in our script has executed. Line one of our code is highlighted in blue which indicates the line that the debugger is paused at. When it is paused at a line, it has not executed that line.

On the right hand side we have a lot of information about where our program currently is at. Information such as: the current call stack, the scope we currently have access to and above tools to work through our application code. We have to manually tell the node debugger tool to continue running our application. The play button will run our application until it is told otherwise. One of the things that is going to pause the application is the debugger statement which we have in our application code. This will open the file that has the debugger statement, if we placed it in another file. At this point in time we can really start to debug our application using the right hand side debugger tools.

The call stack will show us where in the program we are at i.e. the line number of the specific node.js file along with the function called. Within the scope section, we can actually see access to the variables we have in scope. We can dive deeper into those values by running statements from the console e.g. grabbing one of the notes value from the array. We can dump the values in the console and inspect the individual properties to check that the value is as expected it to be.

The debugger statement allows us to dive deeper into the current state of our application and we can

figure out what is not working correctly and what needs to change. If we continue to run our application and there are no more debugger statements, the debugger will run to the end of the application.

Note: the Chalk npm package cannot style the console in the debugger, but this should be OK as it is a little utility to make our application look a little nice when displaying outputs, therefore, not essential to our application code.

Once we run the code in the debugger to the end of our application and close down the developer tool, we should see that on the <chrome://inspect> page, there are no longer any Remote Target available. If we wanted to run through the program again and debug things, we can run a single command in the terminal called restart:

```
< Waiting for the debugger to disconnect...  
debug> restart
```

This is going to restart the program again and we should see a new remote target in the <chrome://inspect> page again that we can click to inspect to run and debug the application again. To shut down the command we can press ctrl + c twice on our keyboard within the terminal to exit the process to bring us back to the command line to do whatever we want.

We can fix our code after debugging and remove the debugger statement once we are done fixing our code. These are the couple of tools we can use when debugging our Node.js applications.

Errors Messages

We can explore some of the common error messages we may see when our application runs:

```
ReferenceError: variableName is not defined
```

After a new line maybe from 3 to 5 lines down in the terminal when we run our code, this provides the actual error message for why the application failed. This provides the explanation from V8 engine as to why our application code could not run.

A ReferenceError is a whole category of errors, but should follow by a explicit message to what the error relates to. This lets us know why things failed but it is not really letting us know why and so it is up to us as developers to figure out what is going wrong. So in the above, is variableName not defined because we spelt it wrong or is it not defined because the function/method did not give us anything back what we thought it would have and therefore when we referenced it later, the code broke. It is still up to us to really work through the problem that caused the error, but the error could get us to the correct part of the code.

Below the error message, we have a stack trace which contains a trace of all of the functions that are running to get to the point where the code breaks. This will point to the file that contains the function as well as the line and the column number. Therefore the first line after the error message contains the most useful information in the stack trace. As we go down the stack trace we typically get less actionable information.

Further down the list we get towards we can view the where the function was called by another function/ method in our application code. After our code, we then get into the internals for example code running from packages such as yargs and the further we get down the stack trace list, the closer we get to the node internals codes functions.

To conclude, when we are looking at the stack trace, we want to start from the top to bottom because the top lines in the terminal contains the most explicit pertinent information related to the error.

We now should know a little about how to work with error messages and how to use the node debugger tools to debug our node.js applications using the Chrome developer tools. These are both techniques we can use to fix our issues faster. Over time as we get more comfortable with Node, we as developers should make fewer issues for ourselves to solve and the ones we do create, we should be able to be solved faster. This like everything else, it is a skillset that needs to be built up over time.

Section 5

ASYNCHRONOUS US NODE.JS

Introduction

If we read a few articles of what Node is, we are likely to come across the same four terms recurring over and over again. These four terms are: Asynchronous, Non-Blocking, Single Threaded and Event Driven.

These terms are accurate ways of describing Node.js but the question is what do these terms mean and how are they going to impact the node applications we are building. We will explore these four terms within section 5 of this document.

Asynchronous Basics

Non-blocking means the application can continue to run other tasks while it is waiting for some long running I/O process to complete. This is what makes node.js fast and efficient.

Below is a basic example of asynchronous code to understand how node.js operates asynchronous tasks.

In a synchronous programming model, one line runs after the next sequentially regardless of how long each line takes to execute. For example:

```
weather-app > app.js:
```

1. `console.log('starting')`
2. `console.log('stopping')`

In the above the first line will execute and print to the console and once completed the next line will execute and print to the console. We can now take this example and write some asynchronous code in-between the two `console.log` to see the behaviour of the asynchronous model.

```
weather-app > app.js:
```

1. `console.log('starting')`
2. `setTimeout(() => { console.log('2 second timer') }, 2000)`
3. `console.log('stopping')`

The `setTimeout` is the most basic asynchronous code that node.js provides us. The `setTimeout` is a function which allows us to run some code after a specific amount of time has passed. This function takes in two arguments and both are required. The first is a function and the second is a number which represents the number of milliseconds we want to wait before the callback function gets executed. Therefore, 2000 millisecond is equal to 2 seconds.

In the above example, the synchronous model will execute the first line and print to the console, it will then

move onto the second line and execute the code which will wait two seconds before printing to the console. It will then finally move onto the third line and execute the final code and print to the console. So the output would look something like:

```
:~.../weather-app$ node app.js  
starting  
2 second timer  
stopping
```

However, asynchronous model will output the logs in a different order. This is because the application will continue to execute the next code without having to wait for the previous code to complete its execution. We will notice the order of the console.logs will be different:

```
:~.../weather-app$ node app.js  
starting  
stopping  
2 second timer
```

To conclude, in a synchronous model we needed to wait two seconds before the programme would continue to the next line of code, while in an asynchronous non-blocking model; node.js can continue to do other things i.e execute the next line of code below while it waits for the two seconds to pass. The above `setTimeout` helps demonstrate the basics of asynchronous programming; however, if we imagine a database request to fetch data for one user, it would be nice to do other things while we are waiting for the

time it takes to get that information back from the database. With a non-blocking model, we are able to do a lot of different tasks at the same time for example waiting for 60 database requests to come back while still issuing more requests.

Below is another example of asynchronous programming, but one which provides a weird result:

```
weather-app > app.js:
```

1. `console.log('starting')`
2. `setTimeout(() => { console.log('2 second timer') }, 2000)`
3. `setTimeout(() => { console.log('0 second timer') }, 0)`
4. `console.log('stopping')`

```
:~.../weather-app$ node app.js
```

```
starting
```

```
stopping
```

```
0 second timer
```

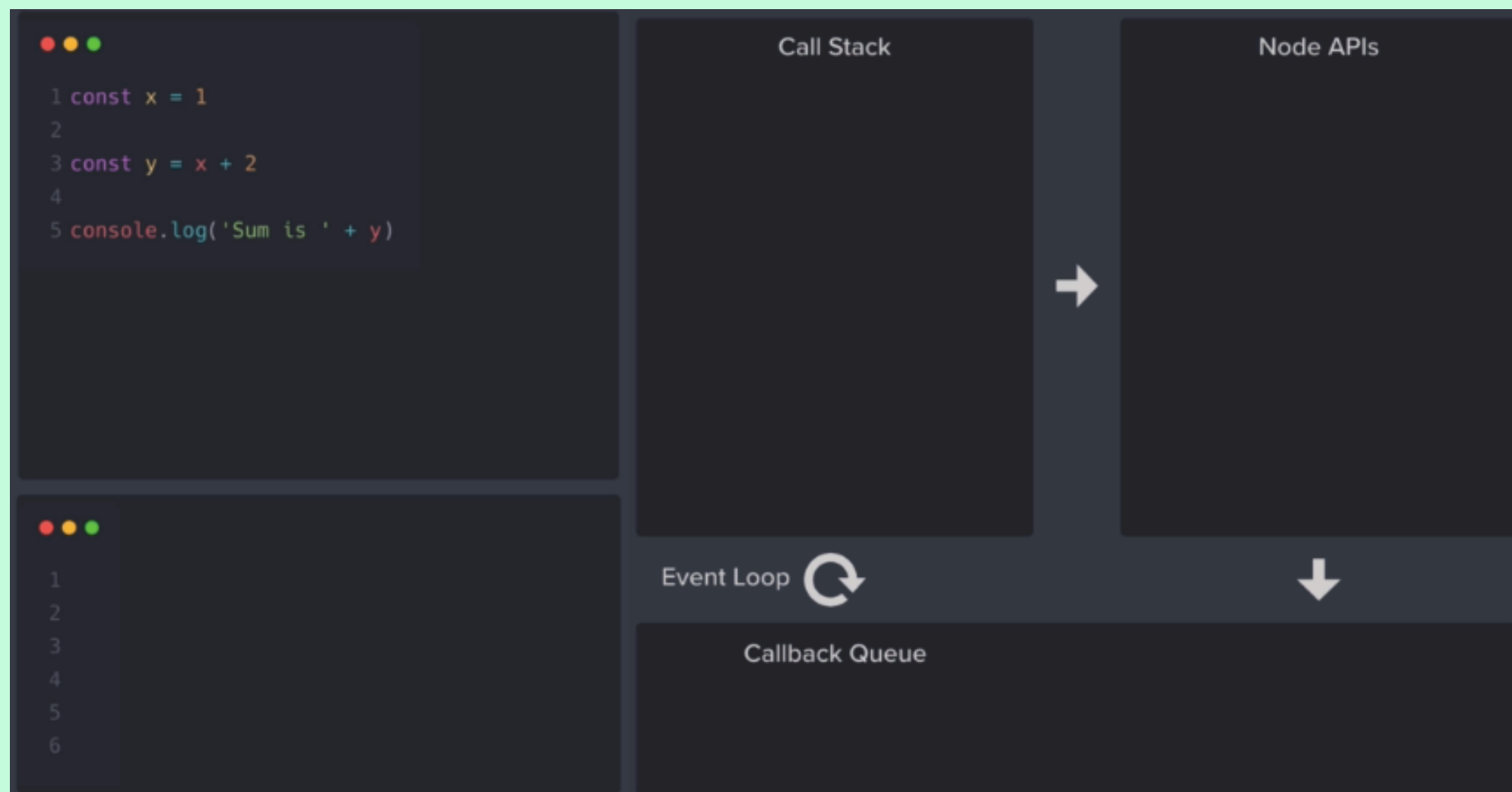
```
2 second timer
```

We would have expected '0 second timer' to print to the console before 'stopping' because we are not waiting for any timer for the function to execute, however, this is clearly not the case. We will explore the internals of Node.js that are responsible for actually asynchronous programming to work, so that we can understand how Node.js is running the code behind the scenes and understand our code executions.

Call Stack, Call Queue and Event Loop

We will explore the internals of Node.js to understand how asynchronous code are executed so that we can answer questions such as why does a `setTimeout` delay not prevent the rest of the programme from running or why does a `setTimeout` of zero milliseconds cause the function to run and execute after code below it as seen in the previous examples. This should hopefully provide us a better visual understanding of what is occurring behind the scenes and exposure to various forms of asynchronous programming.

In the first of three examples, we will see the behind the scenes for a simple synchronous code running in Node:



Even in a simple synchronous code, there is still a lot going on in the background. To break down the above image, we have:

1. In the top left corner we have the node file that is executing. This very simple script that creates a couple of variables and then we print a message to display to the screen.
2. In the bottom left corner we have the terminal output for the script. So when the script prints the message, it will appear in this window.
3. On the right hand side we have all of the internals running behind the scenes in both node and V8 engine. These are the Call stack, Node APIs and the Event Loop, all of which work in tandem to get our asynchronous programmes running.

For a simple synchronous script, the only behind the scenes we are concerned with is the Call Stack.

The Call Stack is a simple data structure provided by the V8 JavaScript engine. The job of the Call Stack is to track the execution of our programme and it does that by keeping track of all of the functions that are currently running.

The data structure for the Call Stack is very simple: we can add something on to the top of the list and we can remove the top item and that is it. It is best to think of the Call Stack as a can of tennis balls. We can add a tennis balls to the top by dropping it into the can and if we want to remove a tennis ball from the can, we would have to remove the top one's first. We cannot remove a tennis ball in the middle without first removing the ones above it and we cannot add one to the bottom if there are already balls inside the can.

We can explore how the Call Stack helps us to run our above script on the left hand side. The first thing that happens with our script, it is wrapped in the `main()` function that is provided by node.js - this is not a function created by us, but a anonymous function created and provided by node.js. This anonymous function is often referred to as the main function for the programme.

This first thing that happens is that the `main()` function gets pushed onto the Call Stack dropping all the way to the bottom, since there is nothing else inside the Call Stack. When something is added to the call stack, it means that it will get executed.



This main function will kick things off with line 1 and move its way down the script lines. Therefore, this will create a const of `x` with a value of 1, then a const of `y` with a value of 3 ($1 + 2 = 3$) and then it will move onto the last line to log 'Sum is 3' within the console/terminal. The `log()` is a function; whenever we call on a function, the function gets added onto the Call Stack. Therefore, the `log()` function is now added to the Call Stack to be executed and we now have two functions on the Call Stack, which are `main()` and `log()` functions.

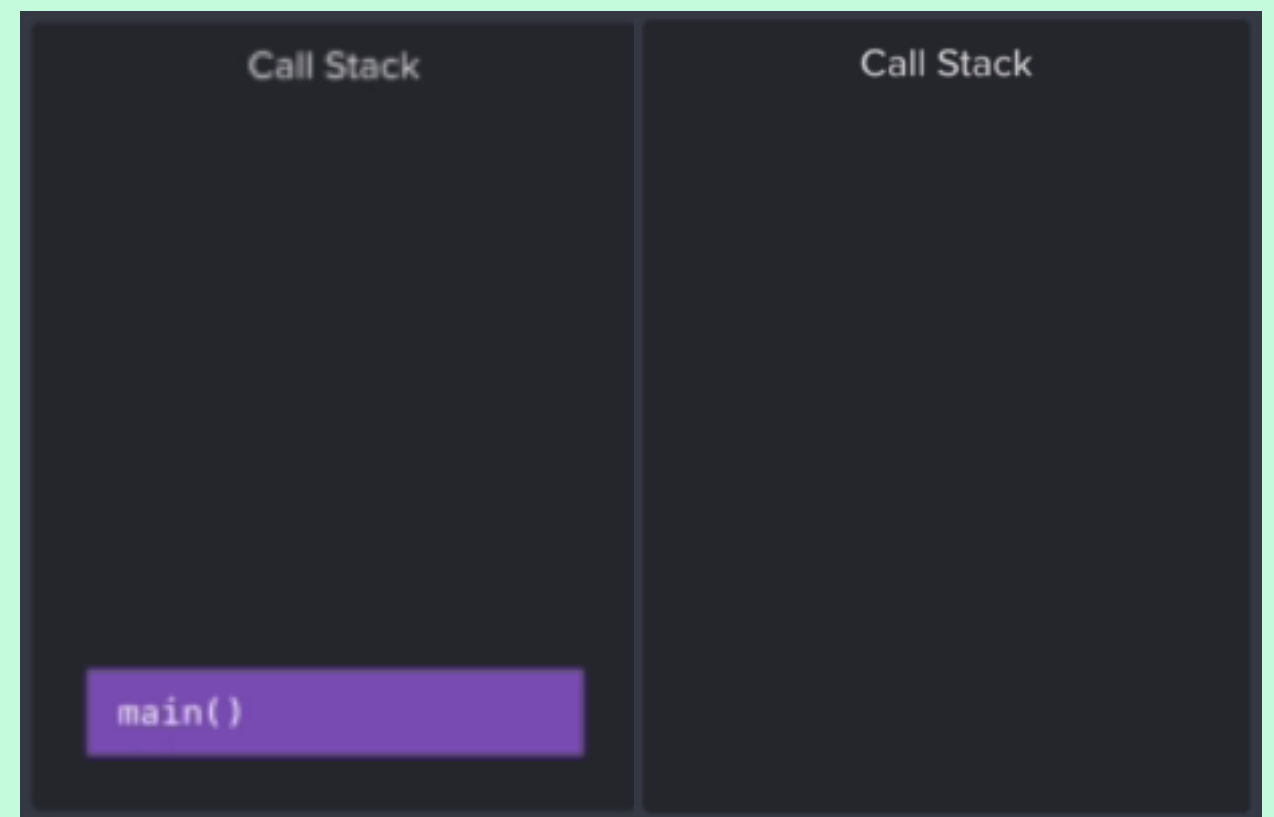


Once the log function executes, this will print the 'Sum is 3' in the terminal which is what we would see when we run the programme.

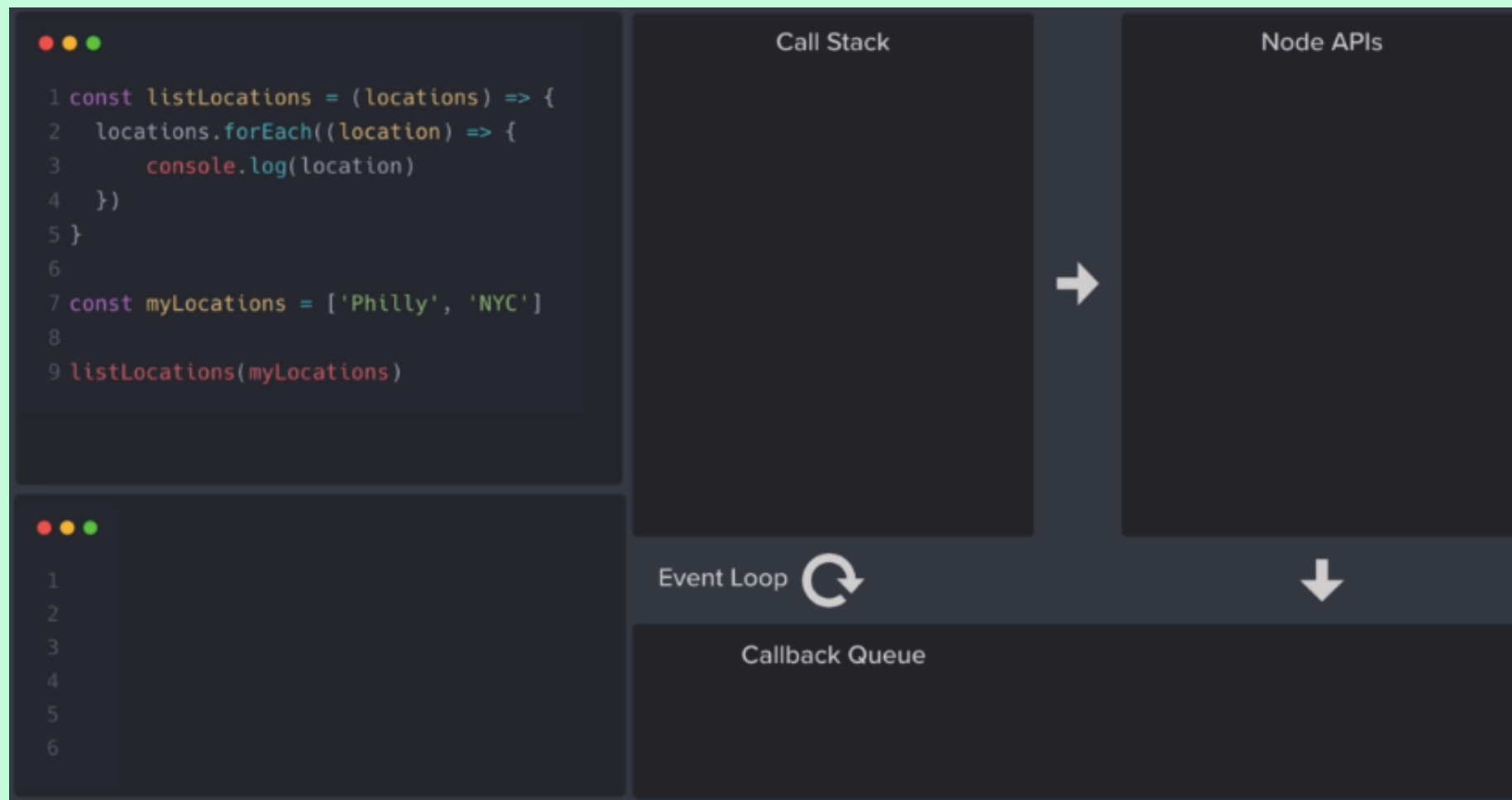
When a function finishes by either running to the end or returning a value, the function gets removed from the Call Stack since it is no longer executing.

Therefore, at this point the log function has done its job by printing something to the console. It is now going to get finished by being removed from the Call Stack.

The blue arrow will then go to the next line of the programme which is actually the end of the script, which means the main function is going to finish as well and be removed from the Call Stack leaving us with an empty Call Stack. At this point the programme is now completed.



In the second example, we still have a synchronous script but there is a little more complexity with multiple function calls.

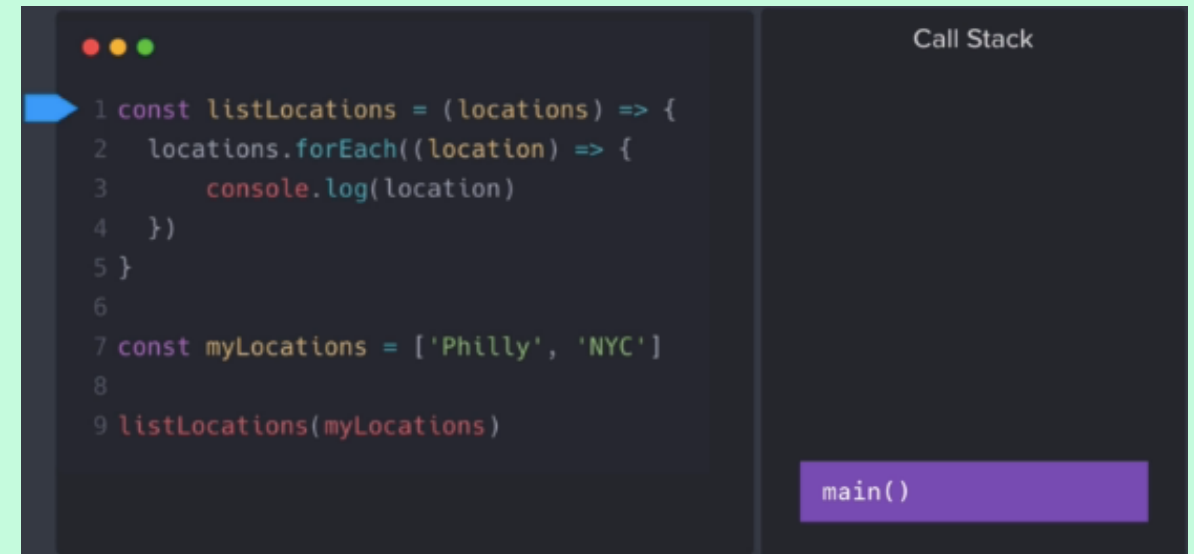


In the script we define a `listLocations` function which takes in an array and for each location, it loops through the item, printing the location to the console. Below we define the location array stored in a variable and pass it into the function.

We can now analyse how node.js will run this programme in the background should we run this script.

The first thing that will execute is the main() function which will be added to the Call Stack, this will allow the script to start executing line by line.

This will define the listLocations variable creating the function which is the variable's value. At this point we are not calling the function so it is not going to appear on the call stack.



```
1 const listLocations = (locations) => {
2   locations.forEach((location) => {
3     console.log(location)
4   })
5 }
6
7 const myLocations = ['Philly', 'NYC']
8
9 listLocations(myLocations)
```

Call Stack

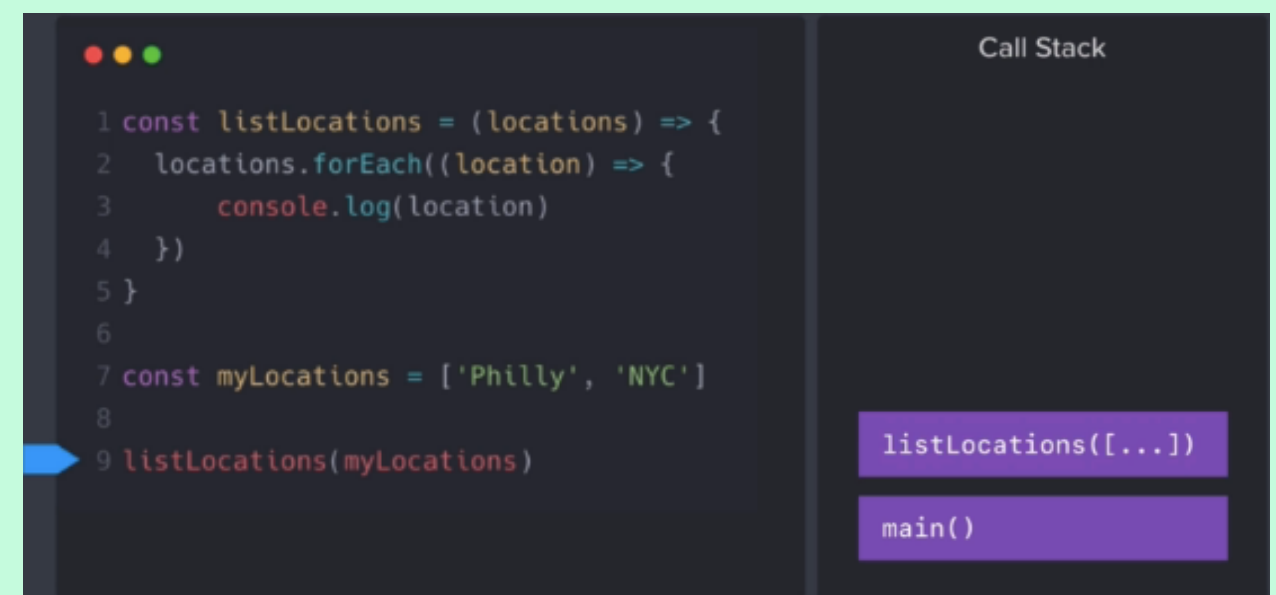
main()



```
1 const listLocations = (locations) => {
2   locations.forEach((location) => {
3     console.log(location)
4   })
5 }
6
7 const myLocations = ['Philly', 'NYC']
8
9 listLocations(myLocations)
```

The next thing that happens is the main function will move onto line 7 of the programme where we define our myLocations array adding a couple of locations to the array object.

From there we move onto line 9, which is where we call the listLocations function passing in the myLocations array object. This is indeed a function call and so we are going to see listLocations added onto the Call Stack.



```
1 const listLocations = (locations) => {
2   locations.forEach((location) => {
3     console.log(location)
4   })
5 }
6
7 const myLocations = ['Philly', 'NYC']
8
9 listLocations(myLocations)
```

Call Stack

listLocations([...])

main()

```
1 const listLocations = (locations) => {  
2   locations.forEach((location) => {  
3     console.log(location)  
4   })  
5 }  
6  
7 const myLocations = ['Philly', 'NYC']  
8  
9 listLocations(myLocations)
```

Call Stack

- forEach(...)
- listLocations([...])
- main()

The listLocations function will now start to execute which is defined between line 1 to 5. The only thing inside of this function is a call to the forEach method looping over each location provided. The forEach is a function call, which is also going to be added onto the Call Stack.

This function is going to run one time for each location, it gets access to the location and it will print it to the console/terminal. The forEach function is going to be responsible for calling this function multiple times. The first time it gets called, it gets added onto the stack. We would see the anonymous function is called with the argument 'Philly'.

Call Stack

- anonymous('Philly')
- forEach(...)
- listLocations([...])
- main()

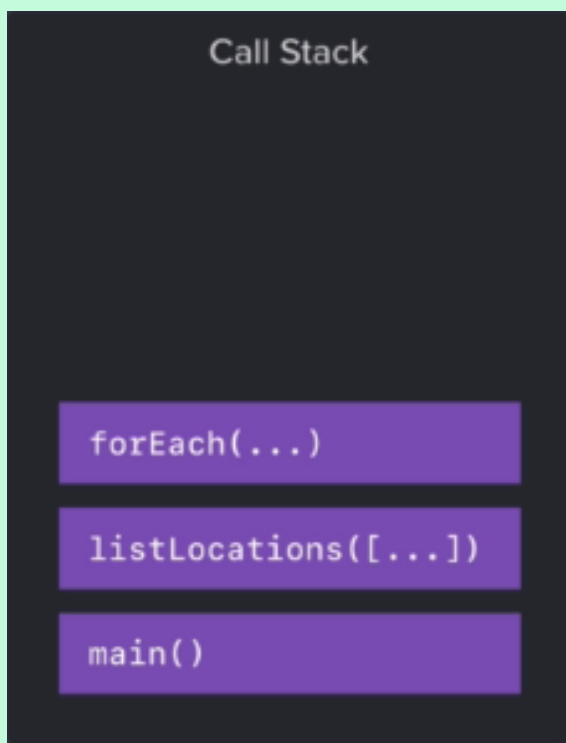
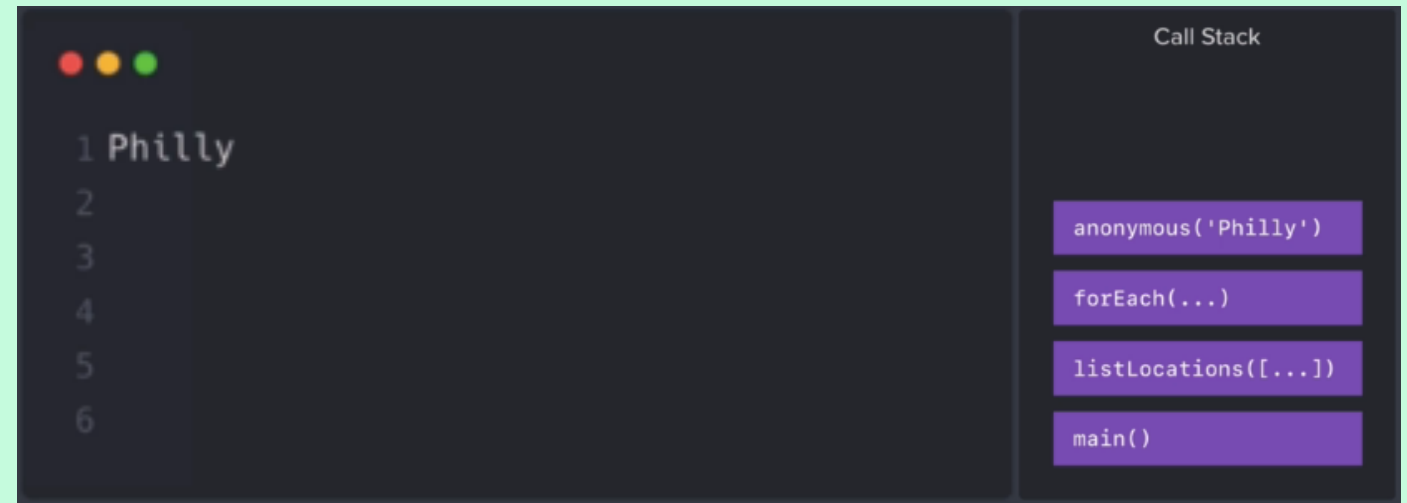
```
1 const listLocations = (locations) => {  
2   locations.forEach((location) => {  
3     console.log(location)  
4   })  
5 }  
6  
7 const myLocations = ['Philly', 'NYC']  
8  
9 listLocations(myLocations)
```

Call Stack

- console.log('Philly')
- anonymous('Philly')
- forEach(...)
- listLocations([...])
- main()

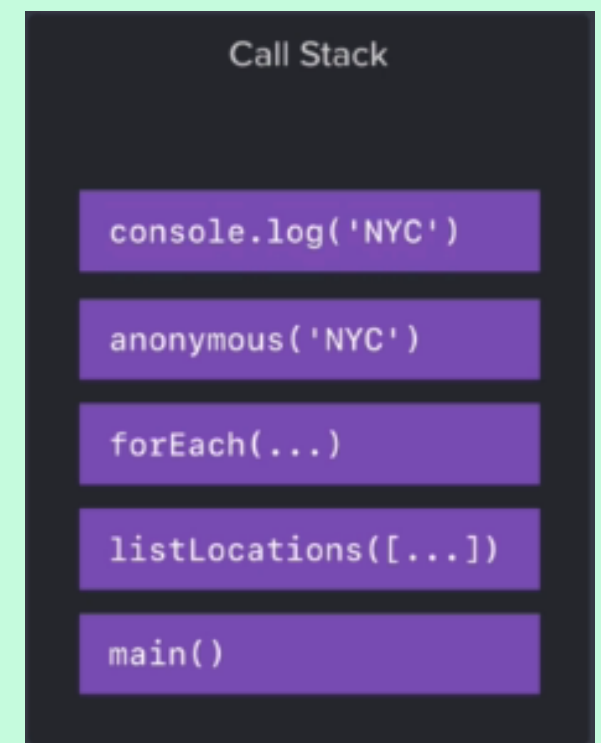
From there we call on the log function to print to the console/terminal which is also then added to the Call Stack.

We will now see Philly printing to the terminal and once this has finished, the `console.log('Philly')` function will now be removed from the Call Stack.

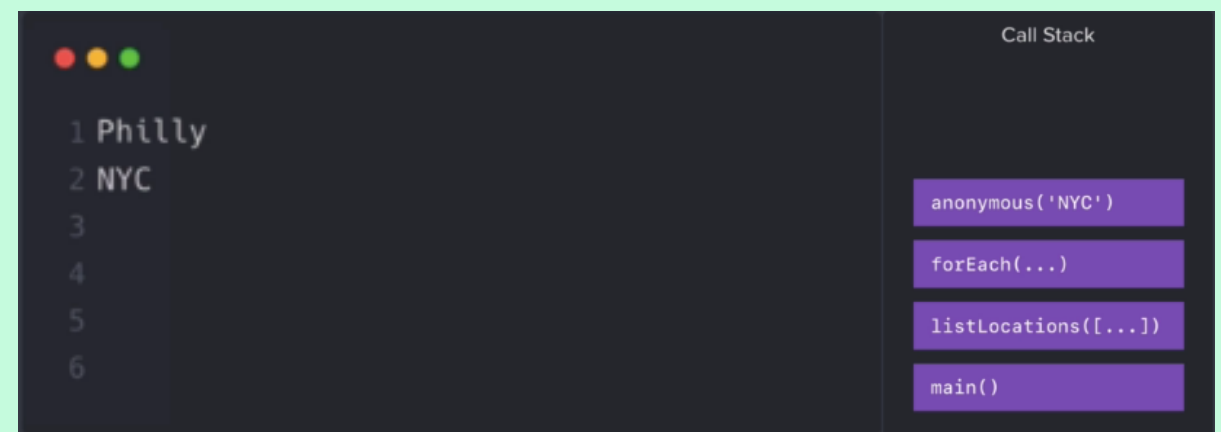


When this log finishes, this is also the end of our anonymous function. This anonymous function will also now pop off the Call Stack as well.

The `forEach` does not come off the Call Stack because it is not completed its execution. It still has to call the anonymous function again to make sure NYC prints to the console. We are going to see a new anonymous function added onto the Call Stack and from there we are going to see another log function added as well.



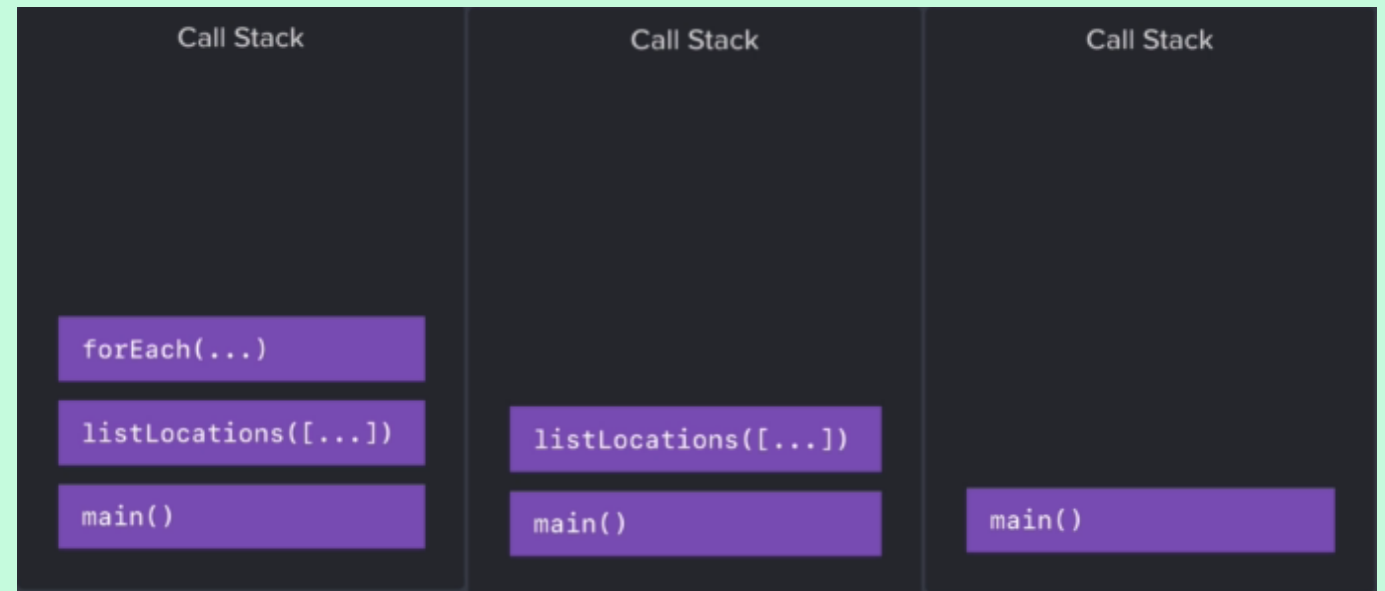
Once the log executes and prints NYC to the console, the log function will be removed from the Call Stack. This is also also the end of the anonymous function which will finish and also removed from the Call Stack.



At this point, the `forEach` function has gone through the process two times because there are only two items in the array. The `forEach` is now completed and will also be removed from the Call Stack as well.

Now that the `forEach` is completed, the `listLocations` function is also completed because it is the only thing the `listLocations` function does i.e. it calls the `forEach` method. The `listLocations` function will also now be removed from the Call Stack.

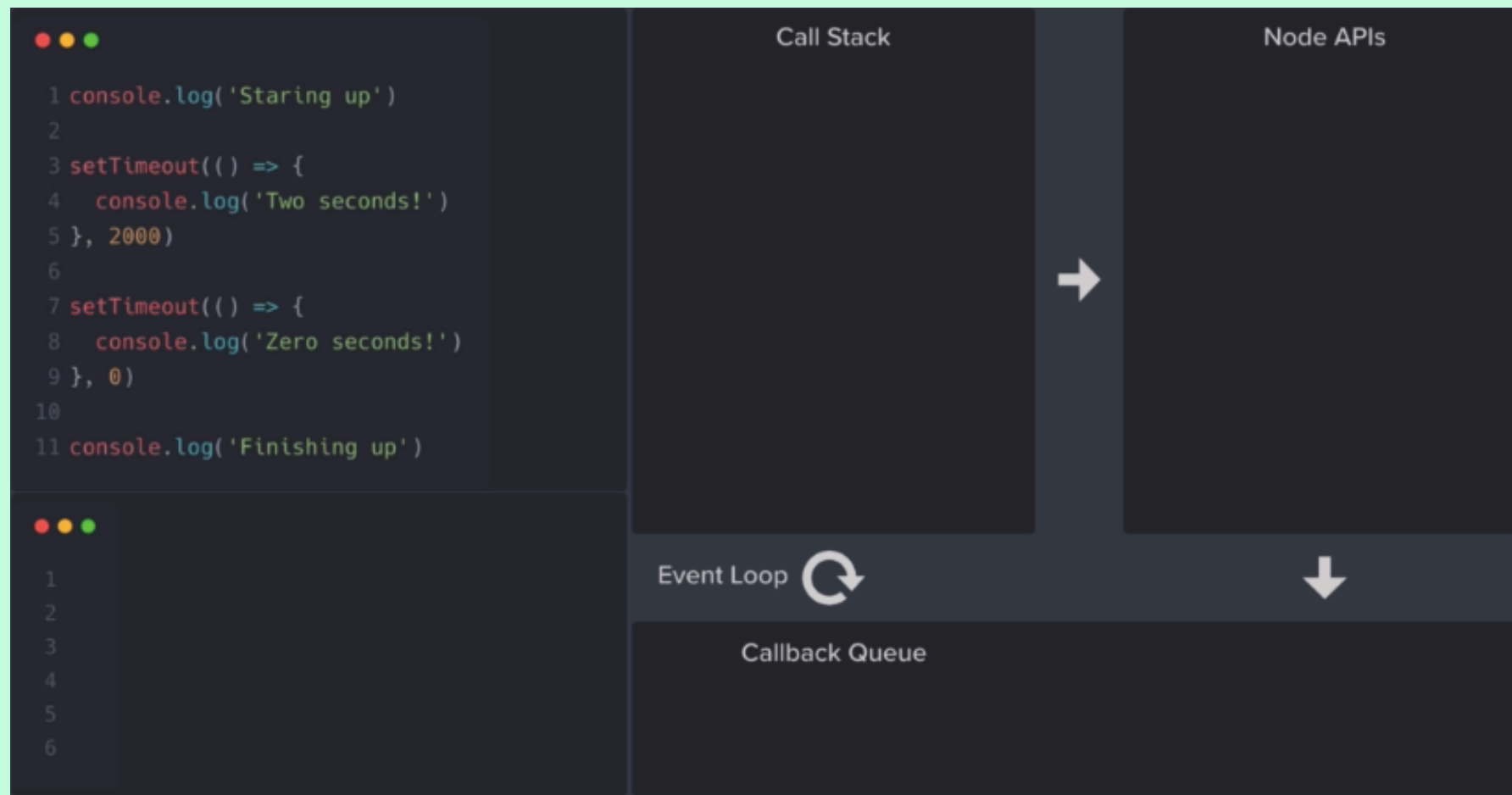
This now brings us back to line 9 of our script which is the last line of our programme which also means that the `main()` function has now finished and will be removed from the Call Stack. Our Script has completed its execution and the final result is Philly followed by NYC printing to the console/terminal.



A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The terminal displays the output of the script:

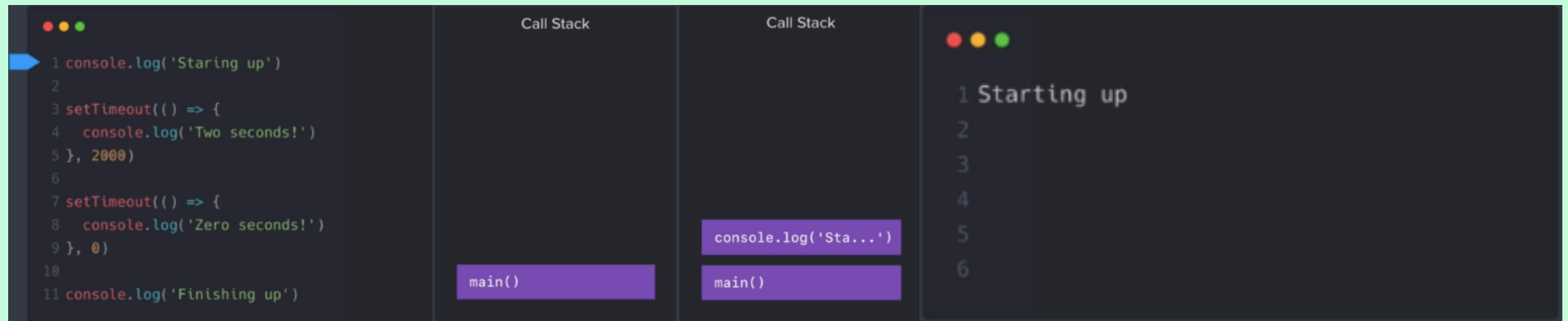
```
1 Philly
2 NYC
3
4
5
6
```


In the third and final example, we have our asynchronous script we explored at the beginning of this chapter. We are now going to explore how it runs behind the scenes to understand why we were seeing things printed to the console in the order that we were seeing them.

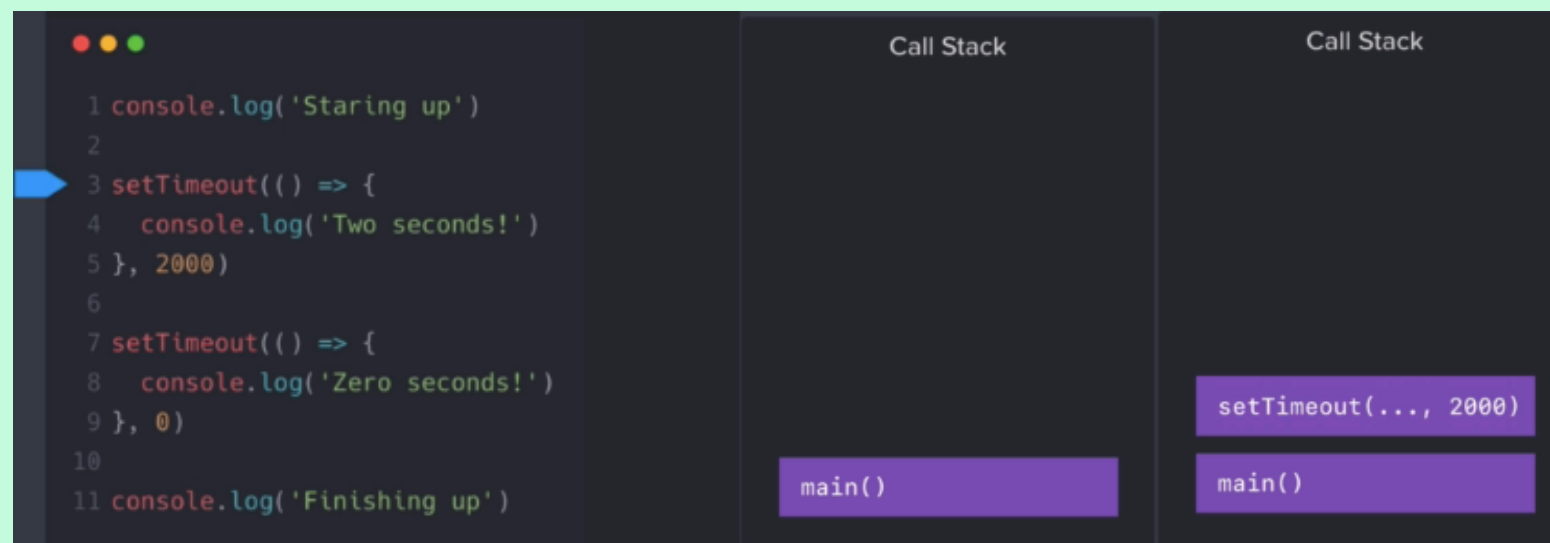


Since the code in this example is asynchronous, we will be using the Call Stack along with the Node APIs, the Callback Queue and the Event Loop all working together to get our application running.

Like with our synchronous examples, the first thing added to the V8 engine's Call Stack is the `main()` function which starts the execution of our programme. On line 1, it calls on the `log` function which also gets added to the Call Stack.



Our message will show up in the console/terminal and the log function is removed off of the Call Stack once it has completed its execution. The next step will move onto line 3 of our script which calls on the `setTimeout` function which gets pushed onto the Call Stack.



The `setTimeout` function is not part of the JavaScript programming language and we are not going to find its definition anywhere in the JavaScript specification and V8 has no implementation for it.

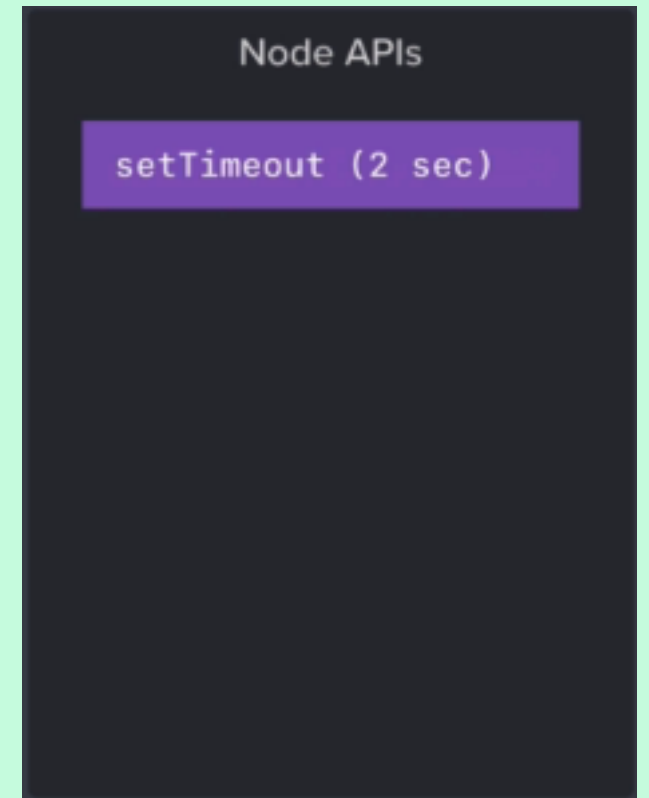
Instead, it is Node.js which creates an implementation of `setTimeout` using C++ and provides it to our Node.js scripts to use.

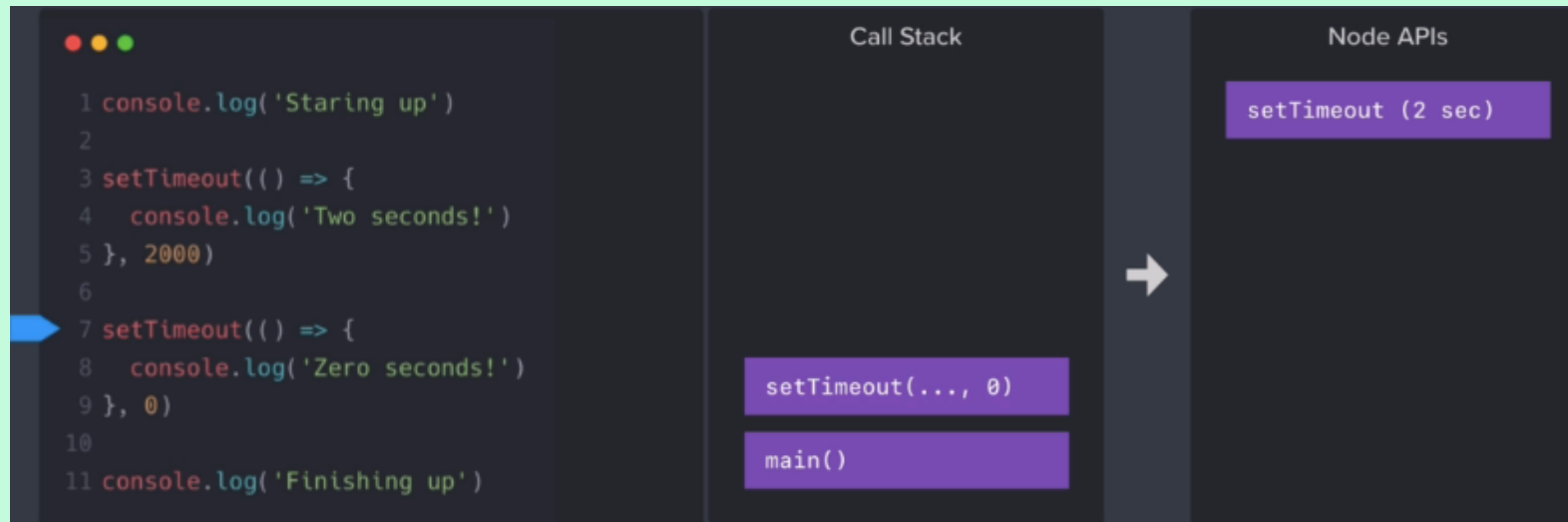
The `setTimeout` function is a asynchronous way to wait a specific amount of time and then have a function run. Therefore, when we call `setTimeout`, it is really registering an event with the Node.js APIs. This is an

event callback pair, where the event in this case is to wait two seconds and the callback is the function to run. Another example of a event callback pair might be to wait for a database request to complete and then run the callback that does something with the data.

Therefore, when we call the `setTimeout` function, a new event gets registered in the Node APIs which is the `setTimeout` callback waiting for two seconds. At this point in the process, the two seconds timer starts ticking down. While we are waiting for those two seconds to complete, we can continue to perform other tasks inside of the Call Stack.

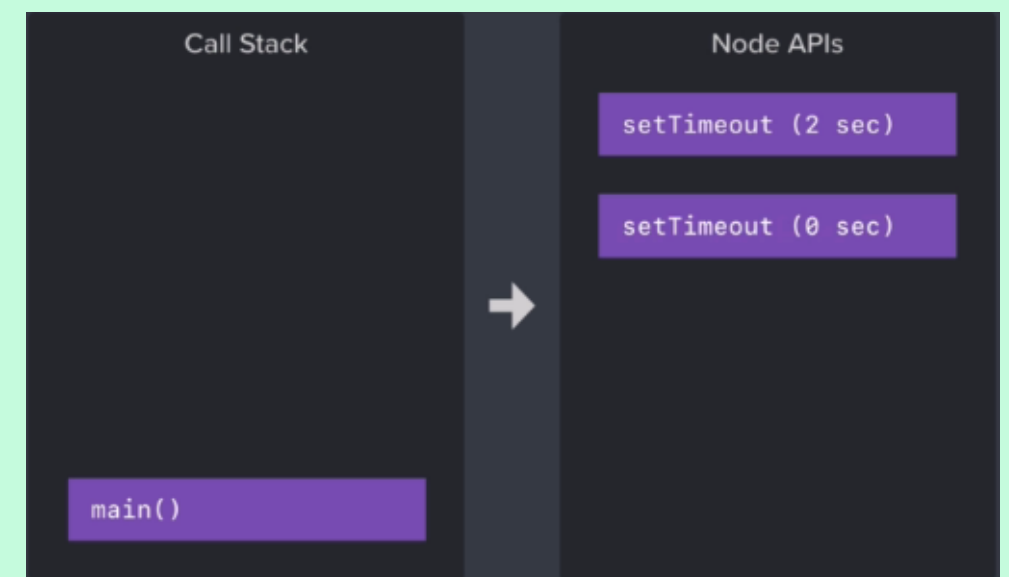
It is important to note that JavaScript itself is a single threaded programming language. You can do one thing at a time and the Call Stack enforces that. We can only have one function on the top of the Call Stack which is the task we are executing. There is no way to execute two tasks at the same time. This does not mean Node.js is completely single threaded. The code we run is indeed still single threaded but Node.js uses other threads in C++ behind the scenes to manage our events, which allows us to continue running our application while we are waiting for the event to complete. This is the non-blocking nature of Node. The `setTimeout` is not blocking the rest of our application from running.





The main() function moves onto line 7 which is another setTimeout function call which gets added to the Call Stack again.

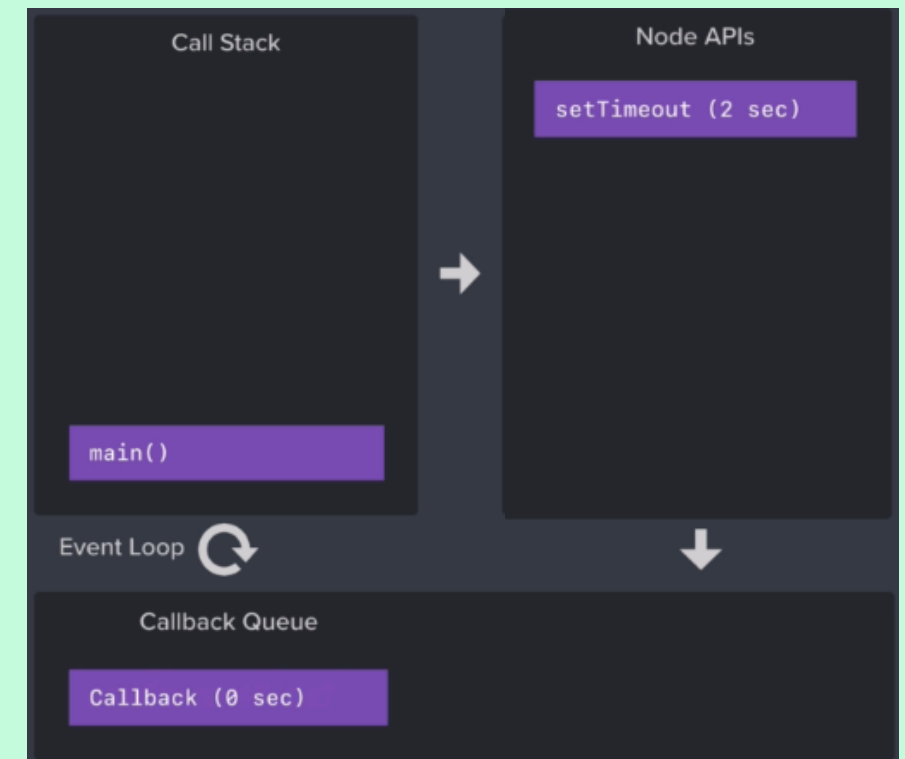
This is going to then register yet another event callback in the Node API area where the callback is waiting zero seconds. We now have two Node APIs waiting in the background. We can continue to do other things while both of the callbacks are waiting for the event to complete (the first case is two seconds and the second case is zero seconds).



The zero seconds are up, therefore the callback needs to get executed. This is where the Callback Queue and the Event Loop come into play. The job of the Callback Queue is simple; its job is to maintain a list of all of the callback functions that are ready to get executed. So when a given event is done, i.e. the zero second timer is completed, that callback function is going to be added onto the Callback Queue. This is a simple line, whereby the function gets added at the end of the line and the front line gets executed first working its way down the list.

Since there are no items in the Callback Queue list, the callback for the zero timer event gets added to the front of the Queue. We now have this callback and it is ready to get executed, but before it can be executed, it needs to be added onto the Call Stack. The Call Stack is where functions go to get run/ executed. This is where the Event Loop comes into play.

The Event Loop looks at two things, the Call Stack and the Callback Queue. If the Call Stack is empty it is going to run items from the Callback Queue. So at this point the Event Loop says that it knows that a callback function got added to the Callback Queue, however, the Call Stack is not empty so it cannot execute the Callback function and therefore the reason why the function does not get executed right away.

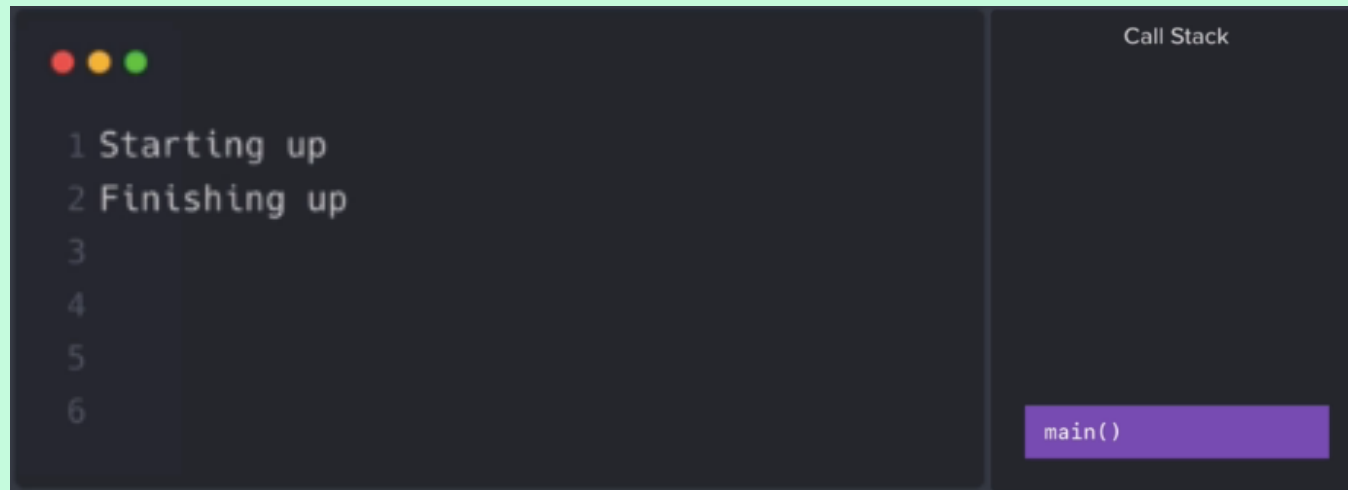


```
1 console.log('Starting up')
2
3 setTimeout(() => {
4   console.log('Two seconds!')
5 }, 2000)
6
7 setTimeout(() => {
8   console.log('Zero seconds!')
9 }, 0)
10
11 console.log('Finishing up')
```

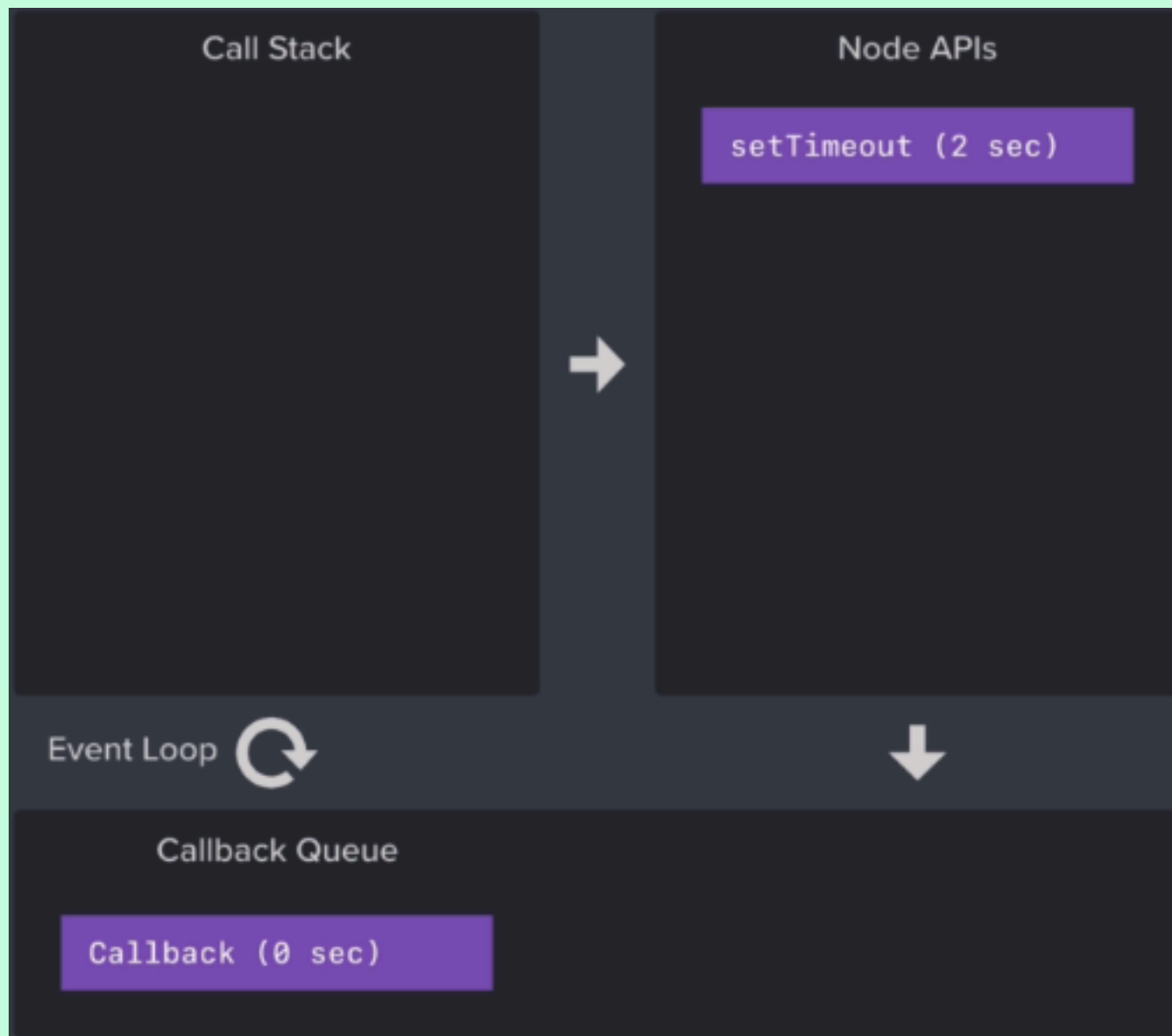
Call Stack

- `console.log('Fin...')`
- `main()`

At this point the `main()` function continues to run. The next thing we see is that line 11 of our programme is going to run, which is a call to log so the function gets added onto the Call Stack and our message gets printed to the console/ terminal.



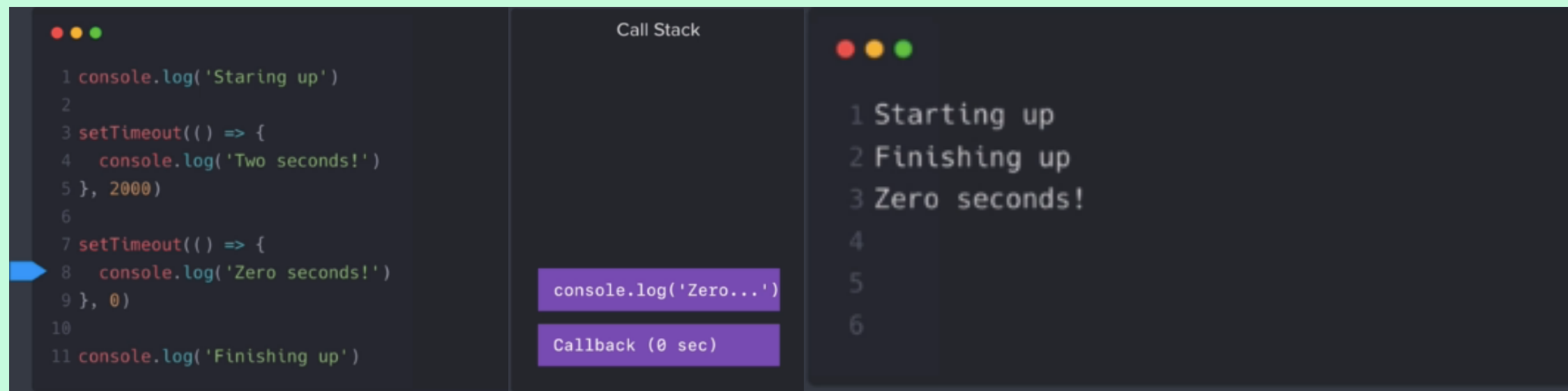
The log function gets removed from the Call Stack and at this point the main() function has completed and will also get removed from the Call Stack.



With our regular synchronous scripts completed, this is when the programme has actually finished. The end of main() signifies the end of the application. This is not the case with our asynchronous programmes.

The Event Loop can now start to perform its job. It can see that the Call Stack is empty and there are callback functions within the Callback Queue. It takes that item and moves it up into the Stack so that the callback can run.

At this point it can run the function on line 8 which is a single call to log which gets added to the Call Stack. The message gets printed onto the console/terminal.



The screenshot shows a code editor with the following JavaScript code:

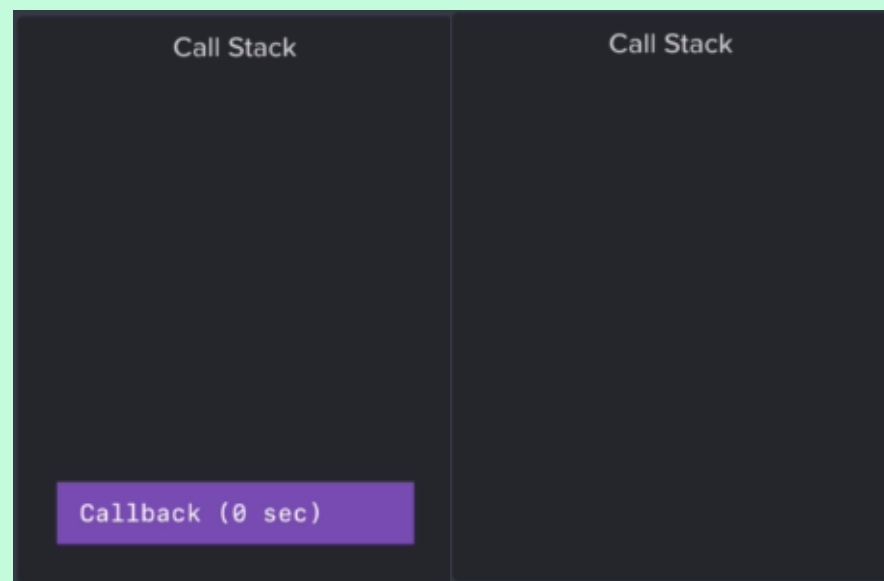
```
1 console.log('Starting up')
2
3 setTimeout(() => {
4   console.log('Two seconds!')
5 }, 2000)
6
7 setTimeout(() => {
8   console.log('Zero seconds!')
9 }, 0)
10
11 console.log('Finishing up')
```

The Call Stack on the right shows the following frames:

- console.log('Zero...')
- Callback (0 sec)

The terminal on the right shows the output:

```
1 Starting up
2 Finishing up
3 Zero seconds!
4
5
6
```



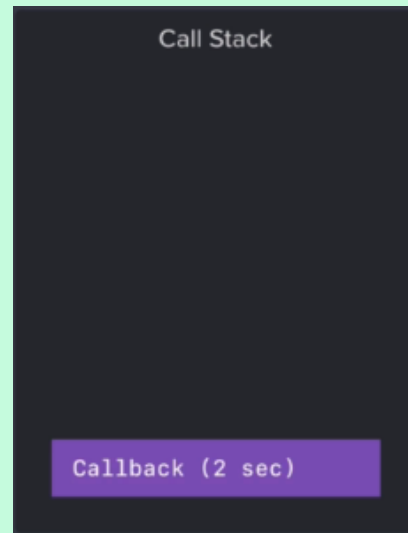
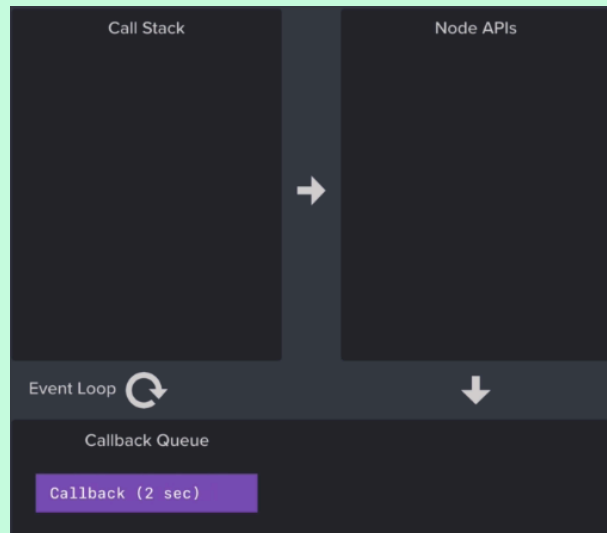
The screenshot shows the Call Stack with the following frame:

- Callback (0 sec)

Once the log completes, the function is removed from the stack and this is also the end of the Callback function which would also get removed from the Call Stack.

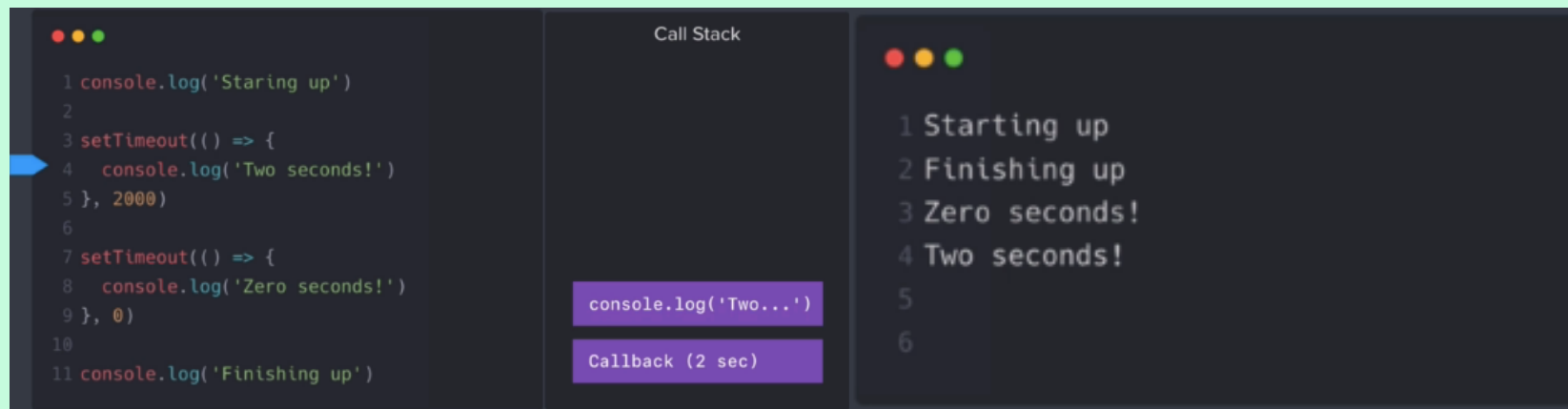
This is the reason for why we were seeing Zero seconds! after Finishing up in the terminal. None of our asynchronous callbacks is ever going to run before the main() function is done.

At this point the programme is not done as it has to wait for the two seconds event for the callback to be added to the Callback Queue .

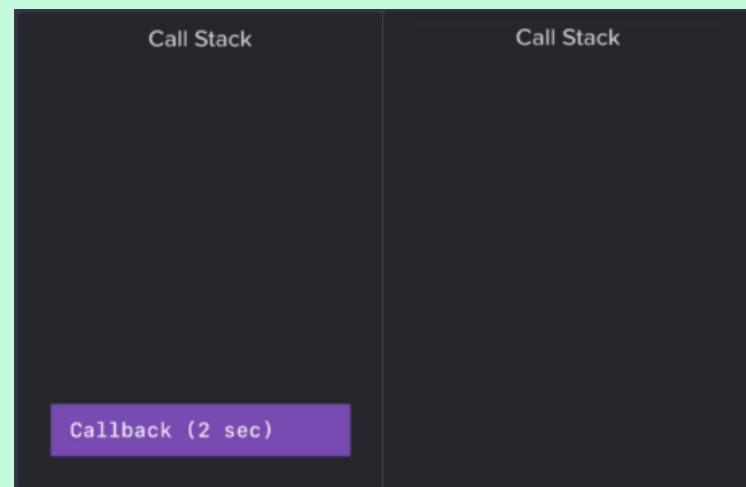


The Event Loop will notice that the Call Stack is empty which means it is ready to run.

The Event Loop takes the callback item adding it to the Call Stack to be executed.



The callback is defined on line 4 which is a call to the log function, which is added to the Call Stack which then gets executed and printed to the console/terminal.



The log function finishes when it prints to the console/terminal and therefore is removed from the call stack which also finishes the callback function and therefore the callback is also removed from the Call Stack. At this point the programme is complete. The Call Stack is empty, the Callback Queue is also empty and there are no other Events registered with Node APIs.

We now know why we got the messages printing in the order we saw them and how Node.js runs our code in the background for both the synchronous and asynchronous model.