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 a 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.

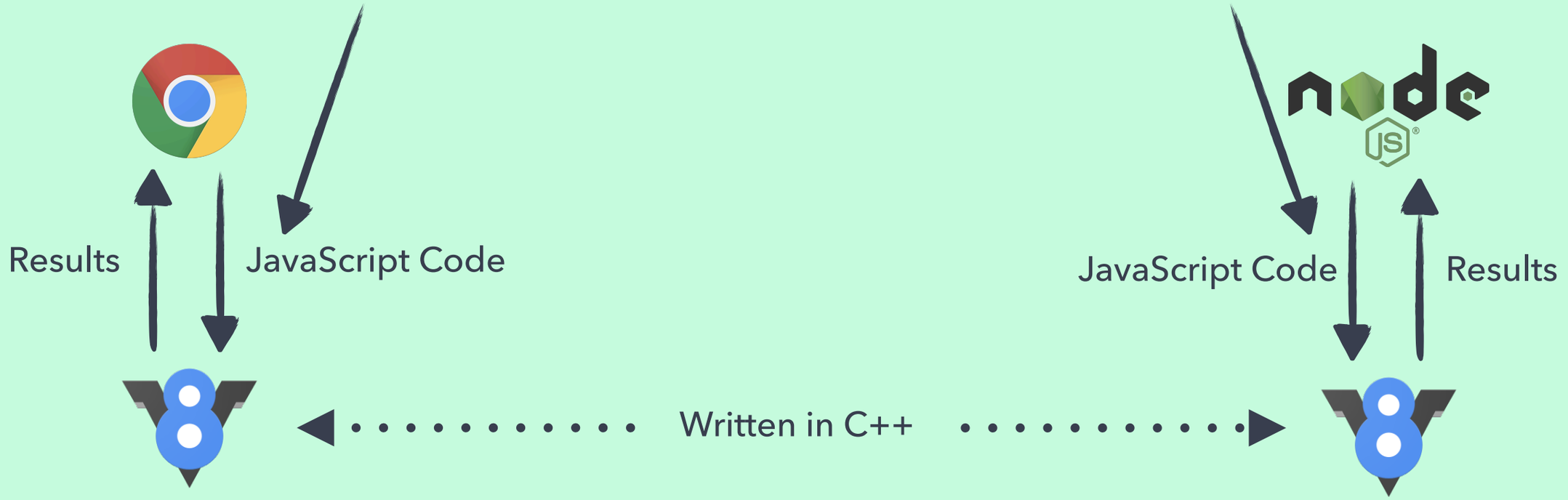
Node.js is not a programming language. The runtime is something that provides custom functionality i.e. various tools and libraries specific to an environment. So in the case of Chrome, the runtime provides V8 with various objects and functions that allow JavaScript developers in the chrome browser to 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 are 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.

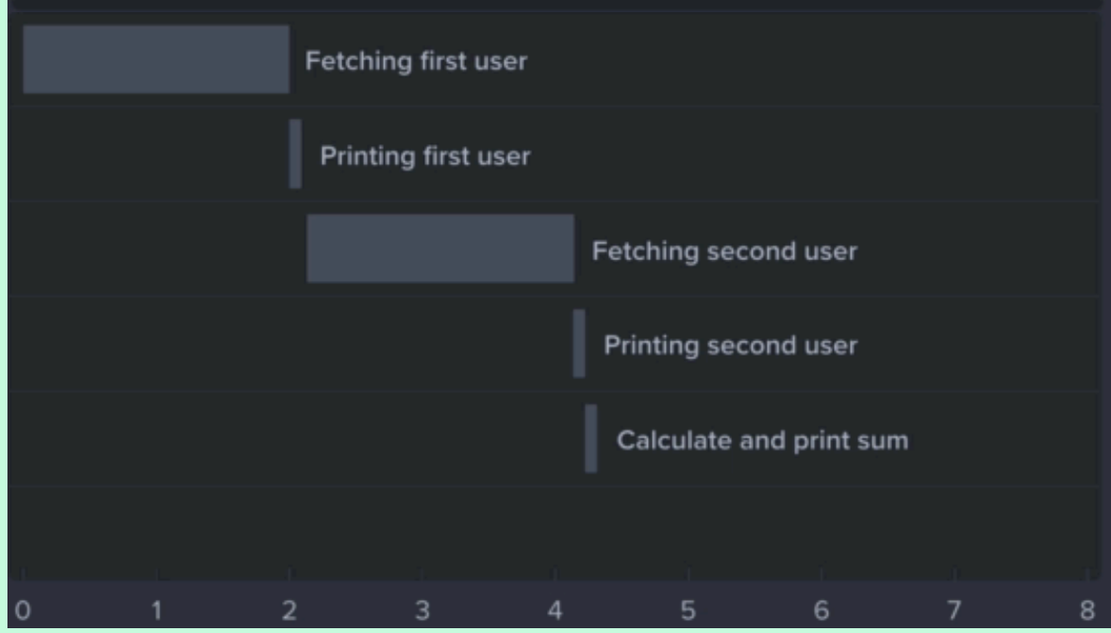
> *"Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js package ecosystem, npm, is the largest ecosystem of open source libraries in the world."*

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.

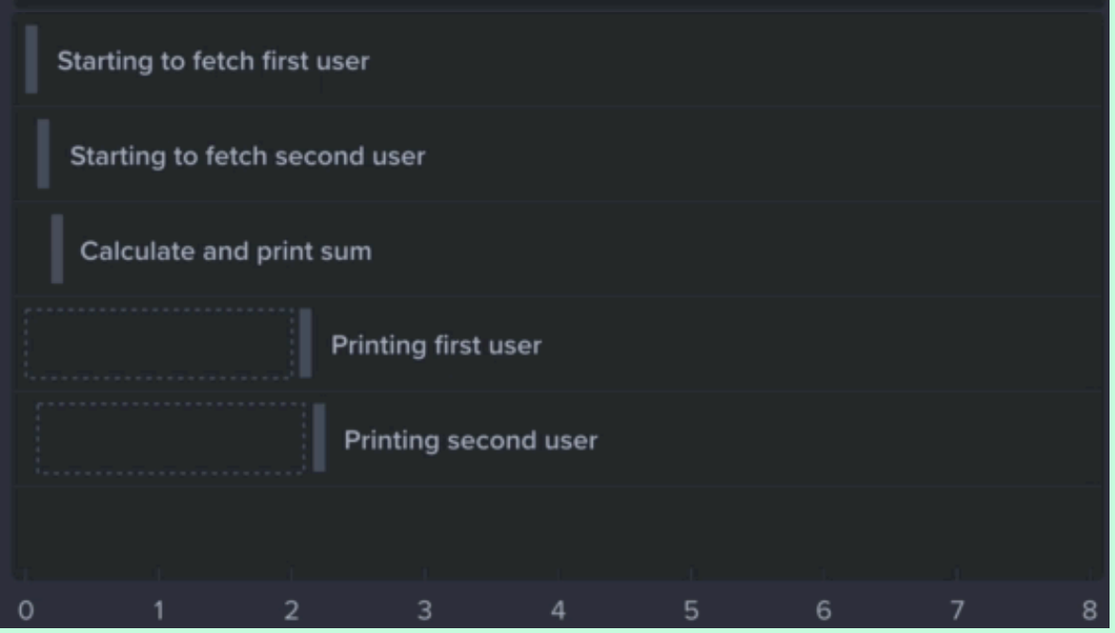
Blocking I/O

```
1 const getUserSync = require('./src/getUserSync')
2
3 const userOne = getUserSync(1)
4 console.log(userOne)
5
6 const userTwo = getUserSync(2)
7 console.log(userTwo)
8
9 const sum = 1 + 33
10 console.log(sum)
11
12
```



Non-Blocking I/O

```
1 const getUser = require('./src/getUser')
2
3 getUser(1, (user) => {
4     console.log(user)
5 })
6
7 getUser(2, (user) => {
8     console.log(user)
9 })
10
11 const sum = 1 + 33
12 console.log(sum)
```



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 allows 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 gobally, 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 ass --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.
```