



# 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.

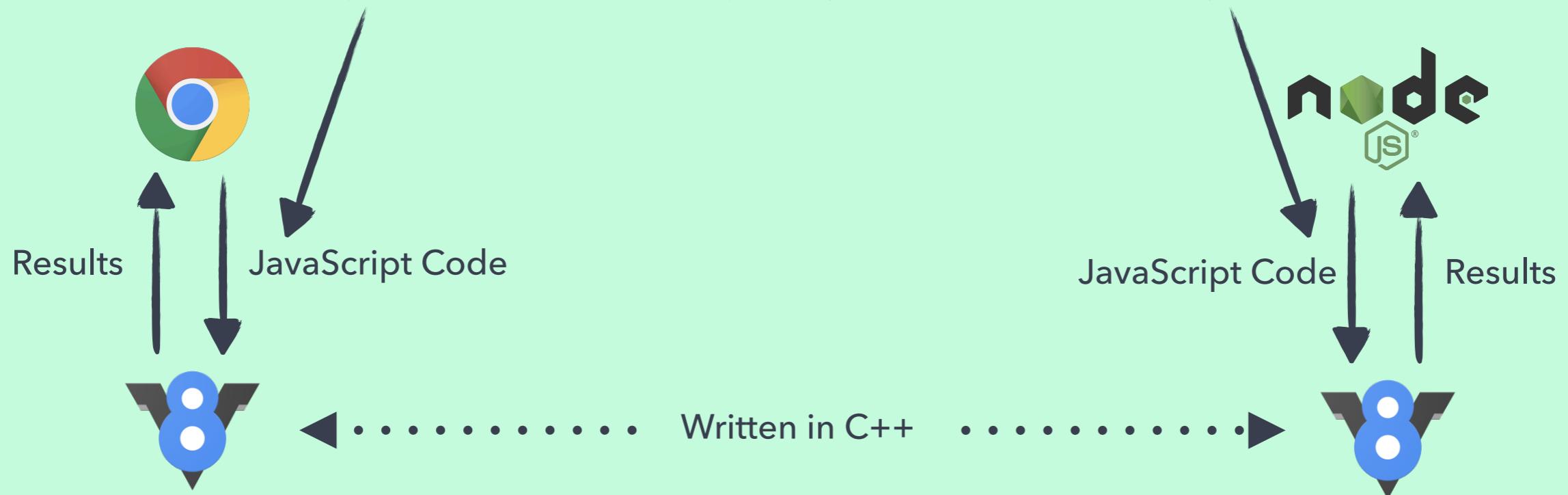
Node.js is not a programming language. The runtime is something that provides custom functionality i.e. various tools and libraries specific to an environment. So in the case of Chrome, the runtime provides V8 with various objects and functions that allow JavaScript developers in the chrome browser to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

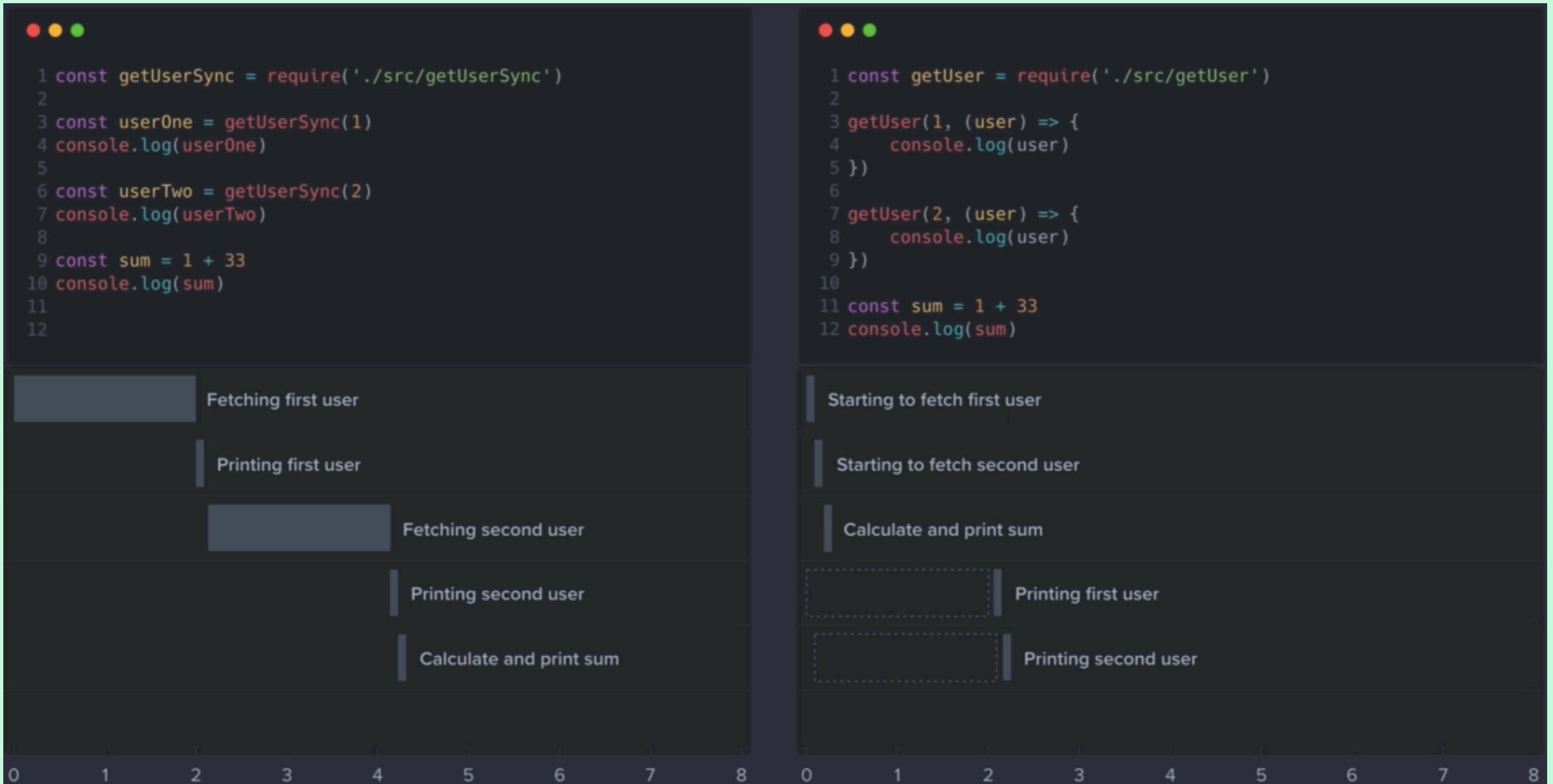
*"Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js package ecosystem, npm, is the largest ecosystem of open source libraries in the world."*

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 questions 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 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 have 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 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 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 applications use command line arguments in some way and there are 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 node's `process.argv` command by running the `app.js` script and passing in arguments, in order to see the difference between the two approaches when 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 stringify() 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 method with working with JSON data. We can integrate this with the file system core module to stare 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('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('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('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('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('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('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 her 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 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 a 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
```

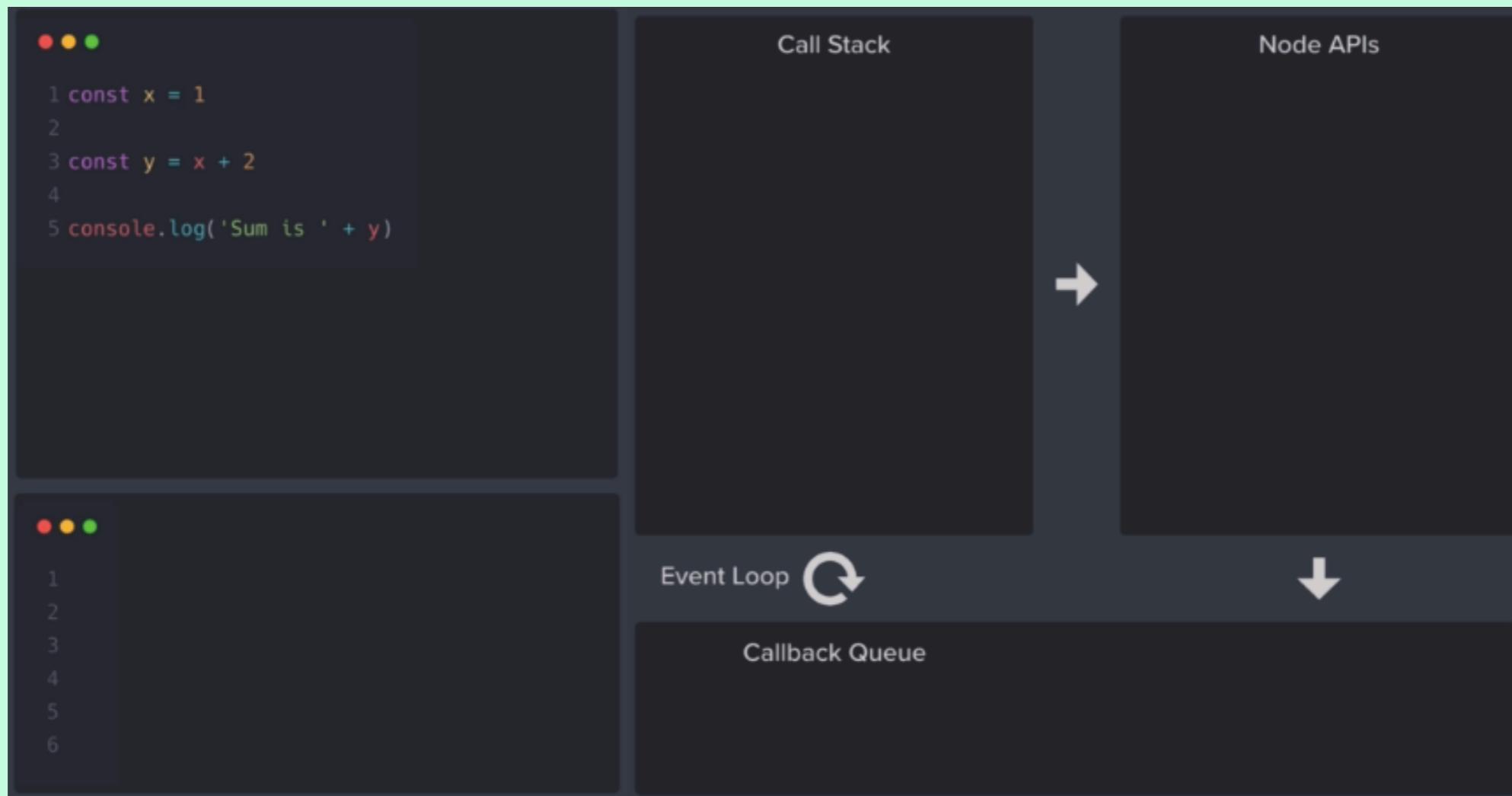
```
2 second timer
```

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.

The screenshot shows a terminal window with a dark background. On the left, a script is displayed with the following code:

```
1 const x = 1
2
3 const y = x + 2
4
5 console.log('Sum is ' + y)
```

A blue arrow points to the first line of the script. On the right, a sidebar titled "Call Stack" shows a single purple bar labeled "main()".

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.

The screenshot shows a code editor with a dark theme. On the left, there is a terminal window displaying the output of a script. The script content is:

```
1 const x = 1
2
3 const y = x + 2
4
5 console.log('Sum is ' + y)
```

The terminal output shows:

```
1 Sum is 3
2
3
4
5
6
```

On the right, there is a "Call Stack" section. It contains two purple rectangular boxes stacked vertically. The top box contains the text "log('Sum is 3')". The bottom box contains the text "main()". A blue arrow points from the bottom of the "log" box towards the bottom of the "main" box, indicating the flow of execution.

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)
```

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.

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)
```

main()

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)
```

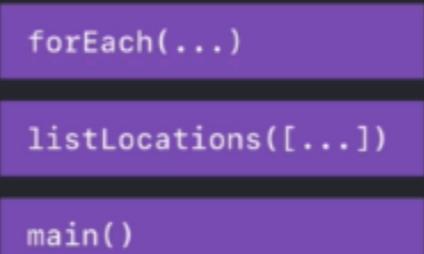
listLocations([...])

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)
```

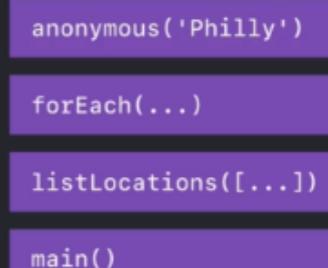
Call Stack



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.

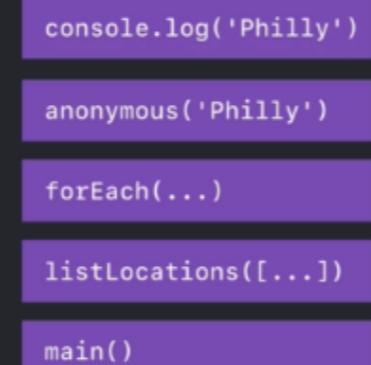
This function is going to run one time for each location, it gets access to the location and it will print it to the console/terminal. The forEach function is going to be responsible for calling this function multiple times. The first time it gets called, it gets added onto the stack. We would see the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

Call Stack



```
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



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.



Call Stack

forEach(...)

listLocations([...])

main()

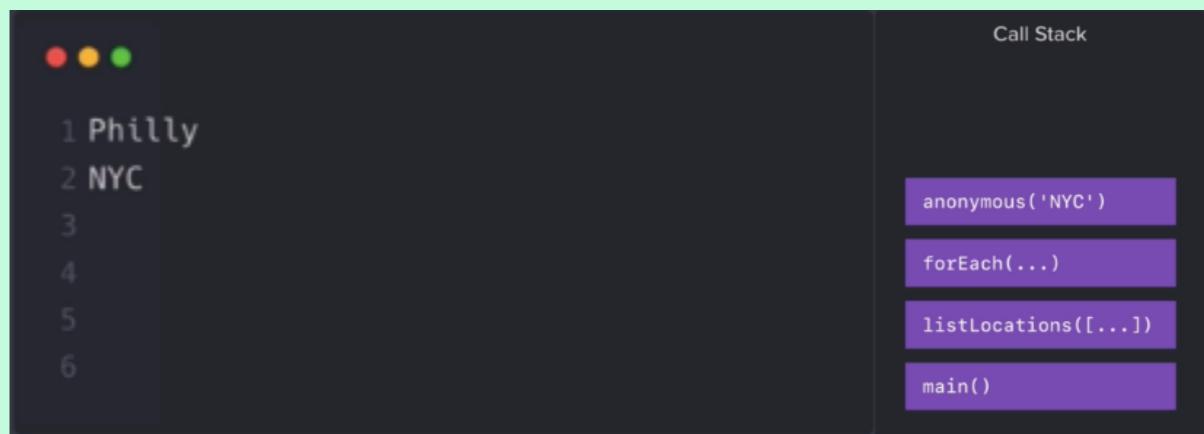
When this log finishes, this is also the end of our anonymous function. This anonymous function will also now pop off the Call Stack as well.

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 screenshot of a terminal window showing the output of a script. The window has a dark background with three small colored dots (red, yellow, green) in the top-left corner. The output is displayed in white text on the left side of the window.

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

To the right of the terminal window, there is a vertical line and the word "Call Stack" centered above a dark grey area, indicating that the call stack has been cleared.

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.

```
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

Call Stack

1 Starting up  
2  
3  
4  
5  
6

console.log('Sta...')  
main()

main()

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.

```
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

Call Stack

1 Starting up  
2  
3  
4  
5  
6

setTimeout(..., 2000)  
main()

main()

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

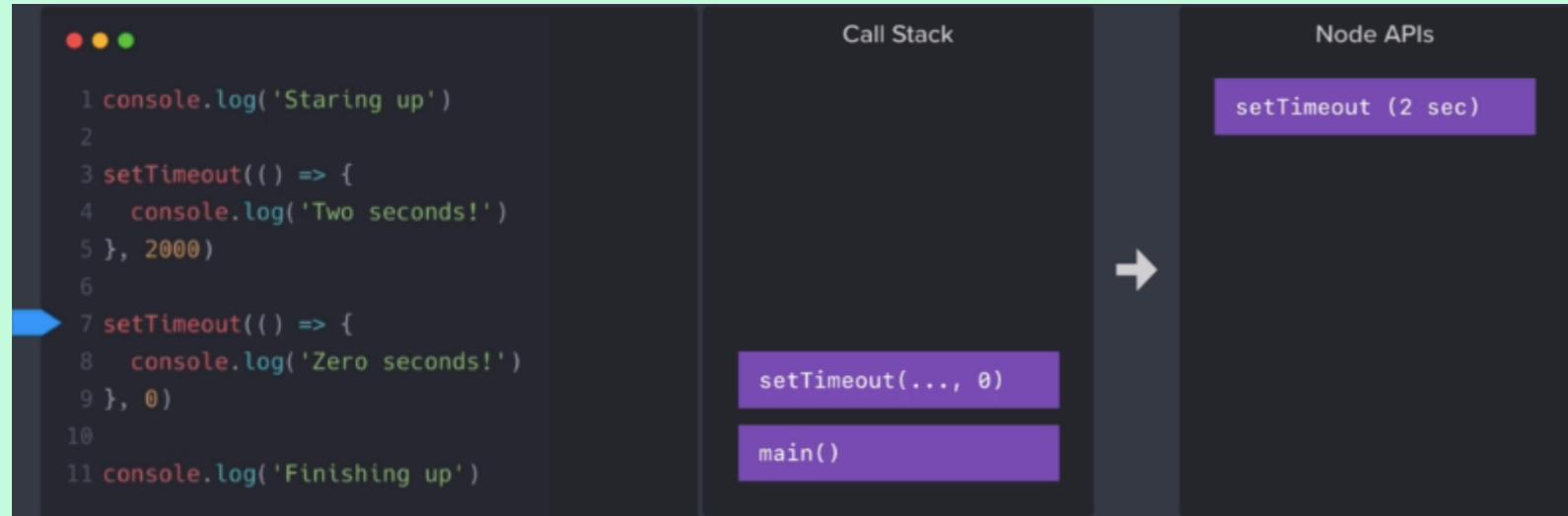
event callback pair, where the event in this case is to wait two seconds and the callback is the function to run. Another example of a event callback pair might be to wait for a database request to complete and then run the callback that does something with the data.

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.

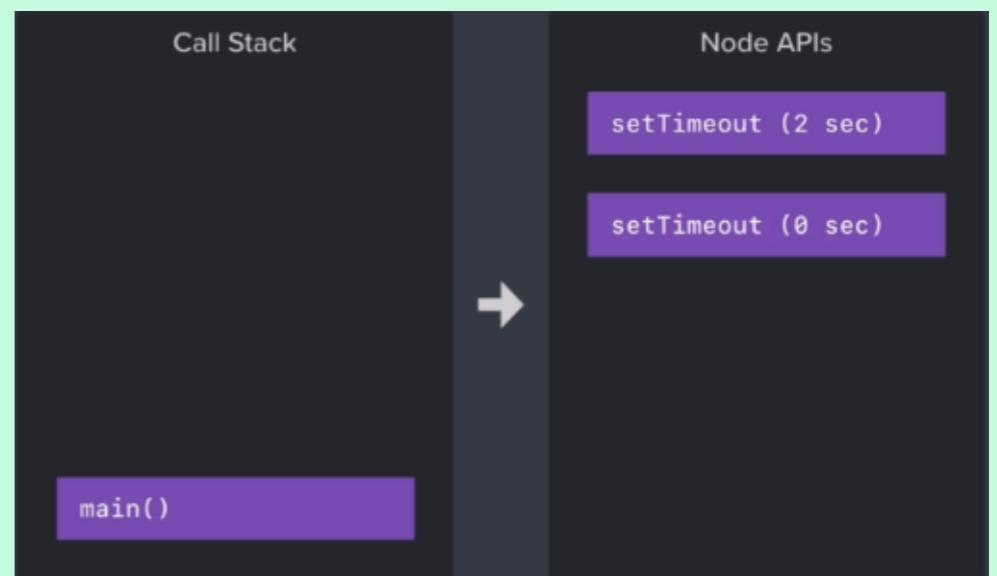
Node APIs

`setTimeout (2 sec)`



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.



A screenshot of a debugger interface showing the Call Stack. On the left, there is 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')
```

An orange arrow points to line 11. On the right, under the heading 'Call Stack', there are two purple boxes: one for 'console.log('Fin...')' and one for 'main()'. The 'main()' box is larger and has a darker shade of purple.

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 out 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.

```
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

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

console.log('Zero...')

Callback (0 sec)

Call Stack

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.

## Making HTTP Requests

HTTP request allows our application to communicate with the outside world. If we want to get real time weather data into our application, we are going to have to make a HTTP request. If we want to send an email to someone from our application, this is also going to be another HTTP request, the same is true with sending a text message using something like the Twilio API which also requires a HTTP request.

HTTP request is at the core of building real world applications that actually communicate with the outside world. In the above three examples, it is our Node.js application making HTTP request to another companies server to get some task done. This would mean somewhere in our code we are going to specify the URL we want to make a request to, which is provided in the APIs documentation, and we are going to fire that request off sending some data (possibly) and getting a response back. For example, to get a weather information, we would send the location we want for the weather information for and we would get back that weather information which we can use in any way we would like in our application.

To practice making HTTP Requests with some weather data, we can use the website: <https://darksky.net/dev> which is the developer portal for the darksky weather application, allowing us to integrate their data into our application. We are required to signup; however, this API is free to use with 1,000 free requests every day. This is more than enough which will help us understand how to make HTTP requests in our Node.js applications and how we can use the returned data.

Once we signup to the API and login we should see a screen with our API secret key and a sample API Call example as seen in the below example:



# Dark Sky API

DOCS FAQ CONSOLE ACCOUNT SETTINGS LOG OUT

Your secret key has been reset.

## Your Account

### Your Secret Key

Your secret key grants you access to the Dark Sky API. You must supply the secret in all API requests.

774f2be3dbdbff72e8bab9cf673a2e70

RESET SECRET KEY

The Sample API Call url link consists of our secret key followed by the longitude and latitude of the location we want to get the data from. If we click on this link to make the HTTP request to the darksky API, we would see the data we would get back from the sample call request.

```
{"latitude":37.8267,"longitude":-122.4233,"timezone":"America/Los_Angeles","currently":{"time":1567348341,"summary":"Clear","icon":"clear-day","nearestStormDistance":239,"nearestStormBearing":163,"precipIntensity":0,"precipProbability":0,"temperature":65.73,"apparentTemperature":65.73,"dewPoint":54.44,"humidity":0.67,"pressure":1013.4,"windSpeed":3.14,"windGust":4.14,"windBearing":252,"cloudCover":0.02,"uvIndex":0,"visibility":9.297,"ozone":294},"minutely":[{"summary":"Clear for the hour.","icon":"clear-day","data":[{"time":1567348320,"precipIntensity":0,"precipProbability":0}, {"time":1567348380,"precipIntensity":0,"precipProbability":0}, {"time":1567348440,"precipIntensity":0,"precipProbability":0}, {"time":1567348500,"precipIntensity":0,"precipProbability":0}, {"time":1567348560,"precipIntensity":0,"precipProbability":0}, {"time":1567348620,"precipIntensity":0,"precipProbability":0}, {"time":1567348680,"precipIntensity":0,"precipProbability":0}, {"time":1567348740,"precipIntensity":0,"precipProbability":0}, {"time":1567348800,"precipIntensity":0,"precipProbability":0}, {"time":1567348860,"precipIntensity":0,"precipProbability":0}, {"time":1567348920,"precipIntensity":0,"precipProbability":0}]}]
```

This is JSON data coming back and this contains all of the weather information for the provided location. When it comes to exchanging data between services, JSON is the standard format for getting this done.

This JSON data is easier for a machine to use and we can use all of the values inside of the data request to provide our users with a detailed forecast of their weather.

We can make the same request as provided in the Sample API Call within our Node.js application. To make a HTTP request, there are a few different things we could do:

1. We can use the core node modules, but this is a very low level (barebones way) which requires us to write a lot of unnecessary code to get everything working together.
2. There are a bunch of npm modules that are essentially wrappers around the core module making it much easier and streamlined to make HTTP requests.

We will use the second option of using the npm module called Request (<https://www.npmjs.com/package/request>). We will later learn how to create HTTP request using the barebones core module approach.

In the terminal we would need to initialise our project as a npm project by running the command in the terminal within the root project directory. The -y flat answers yes to all the setup questions using the default values:

```
:~/.../weather-app$ npm init -y
```

We can then run the following command to install the request npm package:

Once we have our package.json file generated with all of the default values in place, we can now run the following command to install the request npm package. The i flag is short for install:

```
:~/weather-app$ npm i request@2.88.0
```

Once the package has been installed locally in our project, we can go into our app.js file to load and start using the request module to make a HTTP request to our darksky API via our application.

weather-app > app.js:

```
1. const request = require('request')
2. const url = 'https://api.darksky.net/forecast/774f2be3dbdbff72e8bab9cf673a2e70/37.8267,-122.4233'
3. request( { url: url }, (error, response) => {
4.     console.log(response)
5. })
```

This is going to now work because we have the package installed in our package.json file as well as all the necessary modules within our node\_modules folder to get the request package to work. Therefore, we have request itself as well as all of its dependencies showing up inside of the node\_module folder.

To use request module, we need to work with the same URL we have in the Sample API Call example in the browser where we got the JSON data back. So instead of visiting the URL and viewing the data in the

browser, we are going to visit via the request module and we will get that data back as a variable that we could access in our code.

The request function takes in two arguments, the first is an options object which outlines what we would like to do i.e. this is where we would provide the url and other information and the second argument is a function to run once we actually have that response i.e. once the data is available for us to use in our application.

In the first argument we use the url property and pass in the url which we stored in a variable called url. The second argument is the function to run when we are actually getting the response back which we used an anonymous callback function. This function will run when either the response returns the weather data or something went wrong (e.g. the machine does not have a network connection) and we were not able to get the weather data.

The anonymous function is called with two arguments of error and the response. If there is an error this will appear in the error argument else the error argument will return undefined and the response argument will allow us to actually access the response data. The response data is going to contain everything about the response which is way more information than just the JSON data.

If we run the application in the terminal we should see the successful response logged to the terminal:

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

We are going to see hundreds of lines of information printed to the terminal and response has many properties we could use and those properties have other properties and so on. The string of data in green is the string JSON data that we would want to parse and access which is contained in the body property.

We already know how to parse JSON data, below is an example:

weather-app > app.js:

```
1. const request = require('request')
2. const url = 'https://api.darksky.net/forecast/774f2be3dbdbff72e8bab9cf673a2e70/37.8267,-122.4233'
3. request( { url: url }, (error, response) => {
4.   const data = JSON.parse(response.body)
5.   console.log(data)
6.   console.log(data.currently)
7. })
```

We have used `JSON.parse` to parse the data that lives on `response.body` property and store it in a variable called `data`. We can now work with that `data` variable and access any of its properties (e.g. `currently` which contains the current forecast information) to log to the console.

To conclude, we have now accessed our first HTTP API from Node.js allowing us to pull in outside information into our applications.

## Customising HTTP Requests

We can customise our HTTP requests using the request module so that request can automatically parse the JSON response for us. We can also explore some options in the darksky API allowing us to do things like change the language and change the units. This allows us to truly customise our requests to suit our applications needs so that we can actually print a forecast.

To make the request module automatically parse the JSON response for us, we can do this by customising the options object that we pass as the first argument in our request function. The list of all available options are available in the request documentation. We can add a second property of json and set this value to true.

### weather-app > app.js:

```
1. const request = require('request')
2. const url = 'https://api.darksky.net/forecast/774f2be3dbdbff72e8bab9cf673a2e70/37.8267,-122.4233'
3. request( { url: url, json: true }, (error, response) => {
4.     console.log(response.body.currently)
5. })
```

This would mean the `response.body` is already going to be an object and there is no need for us to create a variable to parse the JSON data. We can now log the `response.body.currently` property to the console as we did before in a few lines of code.

To print a forecast to the user, we need to figure out what data we wish to include and which to exclude. This may seem difficult viewing the data in the terminal and it may be easier to view it in the browser to explore what data we want to include from a JSON response. There are extensions for browsers that allow us to view nicely parsed/formatted JSON data. A useful extension for the Chrome browser is JSON Formatter (<https://chrome.google.com/webstore/detail/json-formatter/bcjindcccaagfpapjjmafapmmgkkhgoa?hl=en>) which can be installed in our browser. Once the extension has been installed, if we were to view the Sample API Call in the browser and refresh the page, we should now be able to see the JSON data nicely parsed/formatted by the extension. We can switch between the Raw and Parsed JSON data to view the returned request.

The screenshot shows a JSON response from a weather API. At the top right, there are two buttons: "Raw" (grayed out) and "Parsed". Below the buttons, the JSON data is displayed in two sections. The first section, labeled "Raw", contains the full JSON string. The second section, labeled "Parsed", shows the same data as an expandable tree structure. The tree starts with a single brace {}, which is expanded to show the "latitude", "longitude", "timezone", and "currently" fields. The "currently" field is further expanded to show its "time", "summary", and "icon" properties. The JSON data includes various weather parameters like temperature, humidity, pressure, wind speed, and visibility.

```
{"latitude":37.8267,"longitude":-122.4233,"timezone":"America/Los_Angeles","currently":{"time":1567354309,"summary":"Clear","icon":"clear-day","nearestStormDistance":255,"nearestStormBearing":167,"precipIntensity":0,"precipProbability":0,"temperature":67.36,"apparentTemperature":67.36,"dewPoint":56.08,"humidity":0.67,"pressure":1014.13,"windSpeed":3.23,"windGust":4.32,"windBearing":242,"cloudCover":0.07,"uvIndex":2,"visibility":9.942,"ozone":294.4},"minutely":[{"summary":"Clear for the hour.","icon":"clear-day","data":[{"time":1567354260,"precipIntensity":0,"precipProbability":0}, {"time":1567354320,"precipIntensity":0,"precipProbability":0}, {"time":1567354380,"precipIntensity":0,"precipProbability":0}], "raw": "minutely": [{"summary": "Clear for the hour.", "icon": "clear-day", "data": [{"time": 1567354260, "precipIntensity": 0, "precipProbability": 0}, {"time": 1567354320, "precipIntensity": 0, "precipProbability": 0}, {"time": 1567354380, "precipIntensity": 0, "precipProbability": 0}]}]
```

```
{ "latitude": 37.8267, "longitude": -122.4233, "timezone": "America/Los_Angeles", "currently": { "time": 1567354309, "summary": "Clear", "icon": "clear-day", "nearestStormDistance": 255, "nearestStormBearing": 167, "precipIntensity": 0, "precipProbability": 0, "temperature": 67.36, "apparentTemperature": 67.36, "dewPoint": 56.08, "humidity": 0.67, "pressure": 1014.13, "windSpeed": 3.23, "windGust": 4.32, "windBearing": 242, "cloudCover": 0.07, "uvIndex": 2, "visibility": 9.942, "ozone": 294.4}, "minutely": [{"summary": "Clear for the hour.", "icon": "clear-day", "data": [{"time": 1567354260, "precipIntensity": 0, "precipProbability": 0}, {"time": 1567354320, "precipIntensity": 0, "precipProbability": 0}, {"time": 1567354380, "precipIntensity": 0, "precipProbability": 0}]}]}
```

All of the properties within this data object is accessible to us from the response.body property. From here we can grab any of its properties such as the currently property which is why the response.body.currently works in our code. We can also use response.body.timezone to get the timezone for the current location. So we should now have a better idea of as to how we are accessing the information.

We can click on the DOCS link in the darksky API page to view the documentation of the API to learn how to customise the options available to us. The Forecast Request link will scroll us to the section which provides us the example request. We can see the [key]/[latitude],[longitude] all showing up in the list with the example response below.

The screenshot shows the 'Forecast Request' section of the DarkSky API documentation. On the left, there's a sidebar with links: Overview, Forecast Request (which is highlighted), Time Machine Request, Response Format, FAQ, Data Sources, API Libraries, Privacy Policy, and Terms of Service. The main content area has a title 'Forecast Request' and a URL 'https://api.darksky.net/forecast/[key]/[latitude],[longitude]'. Below the URL, it says: 'A Forecast Request returns the current weather conditions, a minute-by-minute forecast for the next hour (where available), an hour-by-hour forecast for the next 48 hours, and a day-by-day forecast for the next week.' Underneath that is a section titled 'Example Request' with a code block:

```
GET https://api.darksky.net/forecast/0123456789abcdef9876543210fedcba/42.3601,-71.0589
```

```
{
  "latitude": 42.3601,
  "longitude": -71.0589,
  "timezone": "America/New_York".
```

If we scroll past that example request, we get to the Request Parameters where we can start to customise

exactly what is happening. The key, latitude and longitude parameters are provided directly into the url and are required by the API. All of the optional parameters such as units, lang and excludes are all setup as query strings. This would mean at the end of our url, we are going to setup a query string which starts with a question mark followed by key=value pairs. We can add multiple key=value pairs by sing the & sign. We would use this format to setup the other options we want to customise our HTTP request. It is important that the options comes at the end of the url i.e. after the longitude value.

Below is an example of setting up the unit and language options.

weather-app > app.js:

```
1. const request = require('request')
2. const url = 'https://api.darksky.net/forecast/774f2be3dbdbff72e8bab9cf673a2e70/37.8267,-122.4233?units=si&lang=en'
3. request( { url: url, json: true }, (error, response) => {
4.   console.log(response.body.currently)
5. })
```

We now know how to customise the response we are getting back from the API by adding the query strings to our URL. It is very important to read the API documentation as the setup and customisation of APIs will vary from API to API and therefore a good documentation on how to use the API is crucial.

## CHALLENGE:

### Print a small forecast to the User

1. Print: "It is currently 58.55 degrees out. There is a 0% chance of rain."
2. Use the temperature and precipProbability properties from the response data to dynamically replace the forecast values for the location in our printed message.
3. Test your work!

## SOLUTION:

weather-app > app.js:

```
1. const request = require('request')
2. const url = 'https://api.darksky.net/forecast/774f2be3dbdbff72e8bab9cf673a2e70/37.8267,-122.4233'
3. request( { url: url, json: true }, (error, response) => {
4.     console.log('It is currently ' + response.body.currently.temperature + ' degrees out. There is a ' +
       response.body.currently.precipProbability + ' chance of rain.')
7. })
```

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

It is currently 68.29 degrees out. There is a 0% chance of rain.

## Handling Errors

We can always hope things will go right but that is not always the case. There are plenty of things that could go wrong with our http requests. We are now going to focus on those errors to give the user a better experience when things don't go as expected.

The obvious way of things going wrong when making a http request is if there is no network connection. We can test this by temporarily turning off our computer's wifi connection or unplug the ethernet cable and run a http request script. We would get back a really long error message in the terminal but we would notice a `TypeError: Cannot read property 'body' of undefined`.

In this case when things go wrong and there are no network request the error argument contains a value and response does not. We need to add a little conditional logic to check that there are no error before we try to interact with the response data. The response is undefined and what is causing the error message of body undefined problem.

We cannot fix the lack of network connectivity, the purpose of the error argument is to print out a regular human readable way that the user does not currently have access to the whichever API they are trying to access.

weather-app > app.js:

1. ...
2. `request( { url: url, json: true }, (error, response) => {`

```
3.     if (error) { console.log('Unable to connect to weather service') }  
4.     else { console.log(response.body.currently) }  
5.   })
```

We would add a if statement within our http request function to test whether we have a error returned back. If we console.log(error) this will return the error in a format that is not so bad as to what we had before; however, it still does not contain information that would be useful to someone unless they really knew a good deal about programming. We are going to check for the existence of the error object using the if statement.

If there is an error message we would print a error statement using the console.log so that the user would actually understand such as unable to connect to the API. The else part of the if statement will print the forecast as this would be the block of code that runs if there was no error message meaning we had a successful connection to the API. We now have a little bit of error handling within our application making sure that when things don't go as expected, we at least give the user information that is useful about the error.

In summary, we have these two arguments of error and response. Only one of the argument is ever going to be resolved/populated while the other is undefined. Therefore, if we have a value for error, there is not going to be a value for response and vice versa. This is not the only error we would want to guard against as the other error relates to the users input.

Using the darksky weather API, if we were to take the URL and remove the longitude coordinate to simulate a user related error and paste this URL in a browser. In this case we would get a response with two properties back from the API. We have the code property and the error property on the response body.

In this situation we would need to add a little defensive programming to make sure we are printing a useful error message to the user in this error situation.

It is important to note, where we have an invalid input, the server might respond with data that might be considered an error. This will not populate the error argument object. The error argument is used for lower level OS related errors such as a complete lack of a network connection. If the server we are contacting cannot find the data we need, it is typically come back on the response body argument object. We would need to add an else if clause to our if statement to look at the response and see if there is an error code setup. If there is a error code setup, we can print a different error message, else we would continue to print the forecast data.

#### weather-app > app.js:

```
1. ...
2. request( { url: url, json: true }, (error, response) => {
3.   if (error) { console.log('Unable to connect to weather service') }
4.   else if (response.body.error) { console.log('Unable to find location') }
5.   else { console.log(response.body.currently) }
6. })
```

The else if statement is checking to see if the response.body has an error property and if it does then the condition is true and will run the code block for the true statement i.e. the message to print the error to the user, else there is no error property and the forecast will be printed instead.

To conclude, there are two types of errors we need to handle and these are low level OS errors and invalid input errors. Using the if statements and the error argument will help us to handle the different edge cases and provide useful error messages to our users.

When working with API, it is important to test the different edge cases for input errors as this could return different results depending on the API setup and the application logic will need to be setup correctly to handle the errors.

## The Callback Function

We are going to dive deep into the callback pattern to see how it works and how we can take advantage of it by creating our own. This is important to understand because it is at the core of asynchronous Node.js development. It is worth investing time to learn callback functions so that we are more comfortable with it as we continue to use them.

playground > app.js:

1. `setTimeout( () => { console.log('Two seconds are up') }, 2000)`

In this simple setTimeout example, there is a single callback function and that is the anonymous function we defined and passed in as the first argument to the setTimeout function.

A callback function is nothing more than a function we provide as an argument to another function with the intention of having it called later on. Therefore, in the above example, we are providing a function as an argument to setTimeout with the intention that setTimeout is going to call the function at some point in the future. That's all a callback function is.

In the above we are using the callback function in an asynchronous way. The setTimeout is a node provided API that is indeed asynchronous. This does not mean every time we use the callback pattern, it is actually asynchronous. This is a point of confusion for a lot of people.

We have already proven this; we know that setTimeout uses the callback pattern and it is asynchronous and we know that our array methods such as forEach and filter use the callback pattern but they are synchronous.

playground > app.js:

1. const names = ['Andy', 'Beth', 'Cathy']
2. const shortNames = names.filter((name) => { return name.length <=4 })

In this example the filter method is going to be run each time for each array object; however, we are

returning the names that are 4 or less characters long in the new shortNames array. In this case we are indeed using the callback pattern as well but there is nothing asynchronous about filter. It is not interacting with a native Node API as it is a standard JavaScript code that is completely synchronous.

So far we have only worked with callback functions that we have never defined such as setTimeout and filter.

Imagine in our programme there are four different places where we want to be able to take a location and get the coordinates back using the geocode API. If we have four more location we need to grab the location, we would have to write the code and place it in four or more different places which is not ideal. What we would do instead is call a function called geocode or something alike and all of the code will live in that named function we created. To get this done, we would need to be familiar with the callback pattern, below is how the code would look like using the callback pattern:

playground > app.js:

```
1. const geocode = (address, callback) => {  
2.     const data = { latitude: 0, longitude: 0 }  
3. }  
4.  
5. geocode('London')
```

The above is a function we defined. We can setup this function to take in a callback function as an argument just like in setTimeout and filter, and we can set as many arguments as we would like.

If we pretend that in the future we would get the data back from request, what would we do with it? The goal is to give it back to the caller of geocode. So if we call geocode and pass in an address, we want to get access to the coordinates. There are two ways in which we could get this done:

The first could be a return value from the geocode function as we have seen previously in every function so far. This would mean that we do not need the callback argument:

playground > app.js:

```
1. const geocode = (address) => {  
2.     const data = { latitude: 0, longitude: 0 }  
3.     return data  
4. }  
5. const data = geocode('London')  
6. console.log(data)
```

```
:~/.../playground$ node app.js  
{ latitude: 0, longitude: 0 }
```

The problem with the above is that there is nothing asynchronous happening with our code. Later on we would use request which we know is asynchronous.

We will notice with the below code we would have a unexpected result.

playground > app.js:

```
1. const geocode = (address) => {
2.     setTimeout( () => {
3.         const data = { latitude: 0, longitude: 0 }
4.         return data
5.     }, 2000)
6. }
7. const data = geocode('London')
8. console.log(data)
```

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

```
Undefined
```

This will return undefined in the terminal because geocode is not returning anything. In this function there is a single call to setTimeout and there is no return statement directly inside geocode function. Therefore, geocode finishes almost immediately and if you do not return something from a function, we know that JavaScript will implicitly return undefined and so that is what we end up seeing. There is a return in geocode; however, it is nested in another function. That return statement is returning a value from the setTimeout function and not from the geocode function, which is why we are not getting a value back. The return pattern is no longer going to work for us when we start to do asynchronous things inside of our functions. This is where the callback pattern is going to come into play.

The alternative is to provide a callback and get the data from there to get the function back to a working state. We would need to pass in callback as the second argument.

playground > app.js:

```
1. const geocode = (address, callback) => {
2.     setTimeout( () => {
3.         const data = { latitude: 0, logitude: 0 }
4.         callback(data)
5.     }, 2000)
6. }
7. geocode('London', (data) => { console.log(data) })
```

```
:~/playground$ node 4-callbacks.js
{ latitude: 0, logitude: 0 }
```

We know with asynchronous behind the scenes, none of our callback functions i.e. the ones that go to the Node APIs and then come down to the Callback Queue, they do not get executed until the call stack is empty. That means the main() function has to finish. We have to adjust the call to the function call as geocode is not going to have a direct return value. What we do instead is add the callback function as the second argument. The callback function is going to be called at some point in time, the question is what

things are we going to have access to which is unto us to choose. Instead of returning the data in our setTimeout function, we are going to call the callback function passing in the data as the first argument. Therefore, when we call on the geocode function and pass the second argument of the callback function, we are also calling on data as the first argument (the name of the argument in the geocode callback function does not need to match the argument name in the setTimeout callback function as it is only the position that matters). We can now console.log the data.

We are now using the callback pattern in our fictitious asynchronous defined function example.

Note: we could have called the callback argument within const add anything we wanted to as we are not limited to calling it callback.

To summarise, if our function is completely synchronous we are able to use return to get a value out of the function and back to the part of the code that called that function. When our function starts to do something asynchronous, the return is no longer an option. Instead of returning a value, we take a callback in as an argument to our function and we call the callback with the value we want to send back when we have it. This achieves the same goal as the return statement does as it still gets the value back to the code that needs it.

## CHALLENGE:

### Mess around with the callback pattern

1. Define an add function that accepts the correct arguments
2. Use setTimeout to simulate 2 second delay
3. After the 2 seconds are up, call the callback function with the sum
4. Test your work!

```
add(1, 4, (sum) => { console.log(sum) })
```

## SOLUTION:

### playground > app.js:

1. const add = (a, b, callback) => {
2. setTimeout( () => {
3. callback(a + b)
4. }, 2000)
5. }
6. add(1, 4, (sum) => { console.log(sum) })

```
:~/playground$ node 4-callbacks.js
```

```
5
```

## Callback Abstraction

Building on the previous callback pattern we are going to learn how to create a http request inside of a function. This will allow us to do two very important things:

1. It is going to make it easy to create code that is reusable and easy to maintain i.e. we can call on the function as many times as we need without copying it over and over again.
2. It is going to make doing one thing before or after something else e.g. we can geocode the address and output latitude and longitude to use that to fetch the weather data.

weather-app > utils > geocode.js:

```
1. const request = require('request')
2. const geocode = (address, callback) => {
3.   const url = 'https://api.mapbox.com/geocoding/v5/mapbox.places/' +
    encodeURIComponent(address) + '.json?access_token=pk.eyJ1IjoieXR1YmUtdHV0b3JpYWxzliwi
    YSI6ImNrMDFhaHNiNTBvYnczbXF0bm15aXVxNnQifQ.F9o3SEHg8NI0-qQS5Gt8ng&limit=1'
4.
5.   request({ url: url, json: true }, (error, response) => {
        if (error) { callback('Unable to connect to the services', undefined) }
        else if (response.body.features.length === 0) { callback('Unable to find location...', undefined) }
        else { undefined, { latitude: response.body.features[0].center[1], ... } }
      })
7. }
8. module.exports = geocode
```

We would create a function and define the parameters/arguments required as input for the function to operate. We would require a callback function as an argument because http request is an asynchronous task and we would call on this callback function once we have the latitude and longitude data from the request. When we call on the geocode function we would pass in the address input but also define a callback function where we have access to the results after the geocode process is complete.

When working with callback functions and the callback pattern, it is typical to see two arguments passed to callbacks which is error if things went wrong and data if things went well. This is not enforced but is a common convention that we would see across many different tools, utilities and libraries (as we have already seen with the request npm package). Only one of the two arguments are ever going to have a value while the other is set to undefined, allowing us to conditionally check what is going on.

We have the http request url contained within a variable. This url is going to be static except with the one piece where we would need to add the address we wish to return the latitude and longitude. We can concatenate the string beginning and end and place in between the address input to make the url string dynamic. The encodeURIComponent is a function which is used to concatenate a string but this helps in scenarios where a user searches for a location that contains special characters that actually mean something in a URL structure e.g. ? which will be converted into to encoded version i.e. %3F. This will allow to handle the http request correctly without crashing.

The goal is to create a function that is highly reusable, so when someone calls geocode, they may want to do something different with the error message that log it to the console (e.g. save it to a log file on the file system or email it to a system admin). Instead of logging out the error we can pass it back to the callback and the callback function can choose what to do with the error message. Therefore, we can call geocode() five different times and do five different things with the error message.

In the if statement error condition we used the callback function to pass in the message for the error value and undefined for the data value. We do not need to explicitly specify undefined as JavaScript will define this implicitly if we do not have any values for an argument. In the above example we explicitly defined data to be undefined in the callback function which will also work. These values will be returned to the callback function if the http request was to fail and we can use the callback function to do whatever we want with the returned values. This is callback abstraction where we can abstract the values from the callback function to make our functions more flexible.

#### weather-app > app.js:

```
1. const geocode = require('./utils/geocode')  
2.  
3. geocode('London', (error, data) => {  
4.     console.log('Error': error)  
5.     console.log('Data': data)  
6. }
```

## Callback Chaining

The callback chaining pattern allows us to one function and then then next which is particularly useful when chaining multiple different asynchronous I/O operations/functions for example taking the coordinates returned from one function and using it as the input for the next function.

weather-app > app.js:

```
1. const geocode = require('./utils/geocode')
2.
3. const location = process.argv[2]
4.
5. if (!location) {
6.     console.log('Please provide a location.')
7. } else {
8.     geocode(location, (error, data) => {
9.         if (error) { return console.log('Error': error) }
10.        forecast(data.latitude, data.longitude, (error, forecastData) => {
11.            if (error) { return console.log('Error': error) }
12.            console.log('Data': forecastData)
13.        })
14.    })
```

We have the second function nested in the first function. We start the asynchronous call to geocode and when it is done, the event loop is going to make sure that the callback gets called. From there we are going to start another asynchronous I/O operation called forecast and then wait for that callback to finish and inside of this callback we have access to the final data. This is an example of callback chaining whereby we are chaining together multiple callbacks to do multiple things in a specific order.

We are taking the success data returned from the geocode callback and populating the input argument for the forecast function's latitude and longitude values i.e. the input for forecast comes from the output of geocode. When the forecast resolves it will either print the error or success in the console/terminal from its return error or data argument values.

We use the if statement to check for errors before running the second nested function (in the else clause) to prevent it from running should there be an error in the first function. Alternatively, as demonstrated in the above example, we can use the return statement instead of else to return the error message which will stop the second function from executing after printing the error message to the console/terminal, which is another very common pattern.

The output argument name for data in both geocode and forecast are exactly the same which is a problem because the forecast data argument will overshadow the geocode data argument. To address this issue, all we need to do, is come up with a unique name for one or both of the arguments.

## ES6 Object Property Shorthand and Destructuring

We are going to explore some ES6 features that make it easier to work with objects i.e. creating objects and accessing properties on an object.

The ES6 object property shorthand allows us to add values onto an object with a shorthand syntax under certain conditions. Below is the regular ES5 syntax for creating an object and printing it to the terminal:

playground > app.js:

1. const name = 'Alex'
2. const userAge = 20
3. const user = { name: name, age: age, location: 'London' }
4. console.log(user)

The object property syntax can be used when defining an object as seen above. It comes into play when we are setting up a property of which its value comes from a variable of the same name as seen above.

Therefore we can use the shorthand syntax. We remove the colon and the variable leaving what looks like only the property name. This is actually a object property shorthand syntax.

playground > app.js:

1. const name = 'Alex'
2. const userAge = 20
3. const user = { name, age: userAge, location: 'London' }
4. console.log(user)

The property name and variable name must match in order to take advantage of the ES6 object property shorthand syntax else we would end up with a `ReferenceError` message in the console if variable is not defined.

ES6 Destructuring is useful for when we have an object and we are trying to access properties from it. The goal of destructuring is to extract object properties and their values into individual variables.

playground > app.js:

```
1. const product = { label: 'Red notebook', price: 3, stock: 201, salePrice: 'undefined' }
```

In the above example we can destructure the `product` object so that instead of a `product.price` property, we can have a `price` variable with the value of 3, etc. This is useful for when working with complex objects that have a lot of properties we are constantly referencing.

It is nice to have standalone variables that we could easily use, such as `product` and `stock` as seen below. We can access the `label` and `stock` variables throughout the rest of the programme without having to grab it off the object each and every time we want to use it.

playground > app.js:

```
1. const product = { label: 'Red notebook', price: 3, stock: 201, salePrice: 'undefined' }
2. const label = product.label
3. const stock = product.stock
```

The problem with the above example, is that we end up writing a lot of code for each property we want to extract from the object. Therefore, we are going to have multiple lines to pull off multiple values. With ES6 destructuring, we get an improved syntax as seen below:

playground > app.js:

1. const product = { label: 'Red notebook', price: 3, stock: 201, salePrice: 'undefined' }
2. const { label: productLabel, stock, rating = 5 } = product
3. console.log(productLabel, stock, rating)

This syntax seems strange at first, but the more we use it the more easier it becomes to understand. To setup destructuring we create a variable and use the curly brackets followed by the equal sign followed by the name of the object we are destructuring. To create a label and stock variable without accessing it through via `product.label` and `product.stock`, inside of the curly brackets we list out all of the properties we are trying to extract. This creates individual label and stock variables which we can use.

The destructuring syntax makes it easy to extract properties off of an object, creating individual variables that stores the property values. When destructuring we can grab as many properties as we want by listing them out in a comma separated list within the curly brackets and this includes properties that do not exist on the object at all such as rating which value will be undefined by default.

Another nice feature of destructuring is that we can rename the variable we end up creating by following

the property we are extracting with a colon and the new property name. So the new productLabel variable gets its value from the object (which we defined after the equal sign) property (which we defined on the left hand side of the colon).

The last feature we have with destructuring is the ability to setup a default value for the variable should the object not have that property. We do this by adding the equal sign followed by the value after the property as seen in the above for rating. This will override the default undefined for rating as we have now assigned a new default value if the property does not exist on an object. If the object does have the property it will use the value from the object overriding the default we setup should the property not exist on the object.

Finally, we can use destructuring when working with function arguments.

In the below example, we can destructure from our function argument in order to create a local variable that we can use:

playground > app.js:

1. const product = { label: 'Red notebook', price: 3, stock: 201, salePrice: 'undefined' }
2. const transaction = (type, myProduct) => {
3. const { label } = myProduct
4. console.log(label)
5. }
6. transaction('order', product)

We can take this a step further and destructure the argument within the argument list. Therefore, instead of giving a name for the argument, we simply setup the curly brackets and we can destructure whatever we want off of that variable without ever having access to the whole thing. We have access only to the values we choose to destructure.

playground > app.js:

1. const product = { label: 'Red notebook', price: 3, stock: 201, salePrice: 'undefined' }
2. const transaction = (type, { label, stock } ) => {
3.     console.log(type, label, stock)
4. }
5. transaction('order', product)

When we are using destructuring, we can destructure a standard object outside of the arguments for a function and if we know an argument is an object, we can destructure it right inline as both syntax are valid.

We should now be more familiar with the ES6 object property shorthand and destructuring features which we can now utilise in our applications and to become better developers.

## HTTP Requests Without a Library

NPM libraries typically make it easier to do things in node as we have seen with both the request and yargs packages. We do not need those packages to work with command line arguments or to make HTTP requests, but it does make the process a whole lot easier. In this section we are going to explore how to make HTTP request using the modules that node provides.

There are two core modules called HTTP and HTTPS. The documentations for the modules can be found on the node.js website: <https://nodejs.org/dist/latest-v12.x/docs/api/>

There are separate modules depending on the protocol we are using to make our requests. With the npm request package the distinction is not necessary as it abstracts off of that behind the scenes allowing us to easily switch out the protocol we're using without needing to load in a completely separate library.

We can use both these module libraries to create a new server as well as make requests to an existing server. The method we are interested in is the `request` method which appears in both modules and there are two variants (i.e. `.request(options[, callback])` and `.request(url[, options][, callback])`). We use the HTTP module for standard request while we use HTTPS if we are making a request to a secure server.

We will look at the `https` module as an example, since our API requests use the `https` protocol.

## playground > app.js:

```
1. const https = require('https')
2. const url = 'https://api.darksky.net/forecast/774f2be3dbdbff72e8bab9cf673a2e70/40,-75?'
3. const request = http.request(url, (response) => {
4.     let data =
5.     response.on('data', (chunk) => {
6.         //console.log(chunk)
7.         data = data + chunk.toString()
8.     })
9.     response.on('end', () => {
10.         //console.log(data)
11.         const body = JSON.parse(data)
12.         console.log(body)
13.     })
14. })
15. request.on('error', (error) => {
16.     console.log('An error: ', error)
17. })
18. request.end()
```

Since this is a core module, we do not need to install the library and we can simply require the module to load it in. We can use the request function to fire off a request. The first thing we need is a URL to make a

request to. We can then make a request using that URL within the `https.request( )` function passing in two arguments, first the UR and second a callback function which is called with the response as its argument

The callback we use here is very different from the callback we are use to with the npm request package. Core node modules typically operate at a lower level while npm modules abstract away a lot of that complexity. Since we are using the core node module, we are going to have to setup things that we might not think are necessary. For example, in this callback we do not have access to the complete response body and instead we can go ahead and grab the individual chunks that come through because http data could be streamed in multiple parts. What does this mean for us? It means that we have to listen to each individual chunks to come in and we also have to listen for when all chunks have arrived in the request is done.

We start the process by using `response.on( )` which is a function and it allows us to register a handler. There are a few different events we can register callback functions for and one of them is data. We provide the event name as the first argument represented as a string. The second argument we can provide the callback which will fire when new data comes in and we get access to that data via the first and only argument commonly called chunk. This is a chunk of the response which might be the entire thing or it might not depending on how exactly the server has been setup. The other thing we need to setup is to figure out when we are done which we can do via `response.on( )` function but this time on the 'end' event handler. When things are over, this callback function is going to run and it does not get back any

arguments because instead by running we just know that we are done. We now need to put both these functions together to figure out how to get the entire response body, parse it from JSON to a javascript object and then actually use it.

The data event handler callback is going to fire when data comes in which could happen once or multiple times; either way we need to take these chunks and add it somewhere we can store it until we have all of the chunks in order to parse it as JSON. We can do this by creating a variable which we called data which we set to an empty string. We use let instead of const as we are going to reassign the value over time changing the string.

If we try to run the code, things are going to hang i.e. it will not do anything and will not return us to the terminal to be able to do any new commands. It is stuck because we do not have a complete request. There is another method we need to use to say we are actually ready to send this off. To do this we need to make a slight change to the programme. What we get back from the https.request method is what we could refer to as the request itself - we store this in the const request variable. This is the variable that comes back from the return value from request. On here there is a method we can use to kick things off which is request.end(). This is another example of how the low level API might not give us the exact tools we were hoping for and it could get a little confusing event with a simple example as there is a lot going on with very little progress.

If we were to `console.log` the chunks we would see either one or a few `console.log` calls printing various Buffers to the terminal. Therefore, the chunk data that comes back is indeed a buffer and we want a string. We would want to convert the buffer to the string using the `toString( )` method and add the data on the data variable. We take the existing data value and add the new chunk that is converted from a buffer to a string value. We would end up with access to the entire body response within the data variable.

On the end event handler we are going to have access to the body response data. The data event handler is going to run multiple times, one for each chunk, while end event handler is going to run a single time once things are done. The end data is going to be our long JSON data. From here we could go ahead and parse the data using `JSON.parse` to get an object that we can actually pull properties off of.

This is what it takes to make a request with the core HTTP and HTTPS module and we are not really done yet because we do not have any error handling setup. To setup a error handler, just before `request` and `request.end( )`, we would need to setup another `request.on( )` listener to listen for the error event handler. Our callback function is going to allow us to do something with the error when this event is triggered. The error is the first and only argument to the callback.

We now have a very basic HTTPS request with the core node module. As we can see this provides everything we need but at a much lower level than we have probably expected. This is why in the real world people are not making requests with the core module and instead they are using libraries like

request, axis or others to make the request process easier. We may be thinking why doesn't Node just change how those core modules work to make it easier to use and look a little more like a library such as a request. The core Node module are supposed to provide the low level implementation and Node comes bundled with NPM because we are suppose to be using NPM modules when building out our applications.

To conclude we took a dive into how we can use the core Node modules to make requests and we have learned a little bit more about why NPM libraries such as requests are so valuable to the Node ecosystem.

## Section 6

# WEB SERVERS

### Introduction

To this point, all of the application code we have seen examples of are for applications accessible via the command line. This is a great way to get started learning Node; however, it is not realistic for users.

We are now going to explore ways for users to interact with our applications via web browsers. We will start to learn one of the most popular NPM libraries called Express which is a web server framework. Express makes it really easy to create web servers with Node.js. Now that we have an understand of the fundamentals of Node.js, this is going to make it easier for us to differentiate between what is Node.js and what is Express.

### Hello Express!

We will learn how to build our own Node.js server which is going to allow users an entirely new way to interact with our applications. Instead of needing to run commands from the terminal to interact with our app,

users will be able to visit a URL in the browser and interact with our Node.js application. With the Node.js server, we can serve whatever our application needs such as HTML, CSS and client side JavaScript files, etc. Alternatively, instead of serving up a website we could serve up a HTTP JSON based API similar to the darksky API where we are exchanging JSON data back and forth with the server.

Express is a very popular tool (Node.js server library) used to create Node.js servers. We can find the Express documentation on their website: <https://expressjs.com/>

Express is one of the original npm packages and helped put node.js on the map because it made it so easy to create web servers whether static website or a complex HTTP JSON based API to serve as a backend to serve up a mobile or web application.

In this section we are going to explore how to setup a basic web server whereby we can navigate to a URL in the browser and view an asset that the server has served up.

Inside of the terminal within our project directory, we can run the following code to install the Express npm package within our project directory:

```
:~/.../web-server$ npm i express@4.17.1
```

Once we have successfully installed the Express npm package, we now have a library to create a Node.js script that will configure and start up the server. We can create a sub-directory in our project file called src (short for source) and this will be the directory where we would end up putting all of our Node.js scripts. This will help keep things more organised rather than keeping all our files in the root directory. This will help us create a much more complex directory structure which will allow us to scale a little better as the application starts to grow. As things become more complex, it is important to stay more organised.

Within the src folder we will create a file called app.js which will act as the starting point to our node application. We will use this file to load in express, configure it to serve something up and then start the server.

web-server > src > app.js:

```
1. const express = require('express')
2. const app = express()
3. app.get('', (req, res) => {
4.   res.send('Hello Express!')
5. })
6. app.listen(3000, () => {
7.   console.log('Server is up on port 3000')
8. })
```

We first load in the express package by setting a const variable and requiring the express library exports. The express library exposes a single function. Express is actually a function as opposed to something like an object and we call it to create a new express application.

We create a variable, for example app, to store our express application. To generate the application we simply call on the express function. The express function does not take in any arguments, instead we configure our server by configuring various methods provided on the application itself. This mean we are all done setting up a express application and we can start to tell our express application what exactly it should do.

If we imagine that we own the following domain called app.com for our application. When someone visits this URL we want to show the user something such as a homepage for the company website. We will also have other pages such as app.com/help or app.com/about etc. Therefore, we have a single domain and all of it is going to run on a single express server. What we have setup are multiple routes such as the root, / help and /about routes and we can add many other routes to our application.

We would configure our server to send a response when someone tries to get something from a specific route. We set this by setting up a method on the express application object which we called app. This method is called .get() which allows us to configure what the server should do when someone tries to get the resource at a specific URL. The .get() method takes in two arguments, the first is the route and the second is the function. The route is represented by a empty string. The function is where we describe what

we want the server to do when someone visits the particular route. The anonymous function gets called with two very important arguments, the first is an object containing information about the incoming request to the server which is common called req which is short for request. The second argument is the response common called res which is short for response. The response contains a bunch of methods allowing us to customise what we are going to send back to the requestor.

Within the code block we can go ahead and look at the request to figure out what we want to do such as read data from the database, create some html, etc. We can use various methods on response to actually send a response back. The above example provides a very basic text response displaying some text in the browser using the .send() method.

The .send() method allows us to send something back to the requestor. Therefore, if the requestor is making a request from code using something like the npm request library, they will get the response text back and if they are making a request from the browser, this is the text response that is going to display in the browser window. In the above we now have something that will show up at the root URL i.e. the text Hello express! displayed in the browser window.

The last thing we need to do is actually start the server up. To do this we need to use a method on app which we will ever use a single time in our application. This method is called .listen() which will start up the server and have it listen on a specific port. A common development port is port 3000.

It is important to note that port 3000 is obviously not the default port. When we visit a website, we do not provide the port because there are default ports for example, for a HTTP based website the default port is 80. As we explore production deployment in later sections, we will learn how to use those ports. In local development environment where we are viewing things on our local machine, port 3000 is going to work really well.

The second argument we pass to the `.listen()` method is a callback function which just runs when the server is up and running. The process of starting up a server is a asynchronous process, although it will happen almost instantly. We can print a message to the console to let the person who is running the application know that the server did start correctly. This message will not display in the browser.

Now that we have our application server loaded and configured, we can start up the server by running the following code in the terminal:

```
:~/.../web-server$ node src/app.js  
Server is up on port 3000
```

When we run the script and the server is up and running (confirmed by the terminal message printed) we will notice that we are not brought back to the command prompt to run something else. This is because the node process is still up and running. With a web server this process is never going to stop unless we stop it, this is because its job is to stay up and running, listening and processing new incoming requests.

We can always shut the server down using control + c (or command + c) on our keyboard to shut down the web server process and bring us back to the command line prompt and we could always start it again if we need to.

When the server is up and running, we can visit it in the browser by following the URL link <http://localhost:3000/> and we will see the message Hello express! printed on the browser screen. When we visited that url in the browser, it went to our server and the Express server found the matching route for the root and it processed the request using our handler. The handler used res.send() to send back a text response and is exactly what we are seeing inside of the browser.

To add a second route, we would add a second call to app.get() but this time providing a different route string for the first argument. We can continue doing this to create even more routes to our web server.

Whenever we make changes to the server script file, we must restart the server for the changes to take effect because the server will not know that we have made changes to the file. This is simply shutting the server process using control + c on the keyboard and running the node command again. We can use something like nodemon package globally which will automatically restart our server whenever we make changes to our files.

```
:~/.../web-server$ nodemon src/app.js
```

If we installed nodemon locally to our project directory and not globally to our machine, we would have to run the following command to run the nodemon.js script locally passing in our file as an argument:

```
:~/web-server$ ./node_modules/nodemon/bin/nodemon.js src/app.js
```

We could setup a script within our package.json file so that we can run the npm run command followed by the script name to run the above within a shorter syntax.

If we were to navigate to a route that has not been setup within our express web server application, we would see a error message within our browser in the format of Cannot GET [routename] for example:

Cannot GET /about

We can setup a 404 page for routes that we do not have support setup for which we will learn in a later section as it is a more complex topic. We now know how to setup a simple Express web server and serve simple text to certain URL routes.

## **CHALLENGE:**

### **Setup Two New Routes**

1. Setup an about route and render a page title
2. Setup a weather route and render a page title

### 3. Test your work by visiting both in the browser

#### SOLUTION:

web-server > src > app.js:

```
1. const express = require('express')
2. const app = express( )
3. app.get('/about', (req, res) => {
4.     res.send('About Page')
5. })
6. app.get('/weather', (req, res) => {
7.     res.send('Weather Page')
8. })
9. app.listen(3000, () => {
10.    console.log('Server is up on port 3000')
11. })
```

```
:~/.../web-server$ node src/app.js
```

```
Server is up on port 3000
```

Visit <http://localhost:3000/about> and <http://localhost:3000/weather> to test your work.

## Serving up HTML and JSON

We have explored how to send back a text string to the browser for a request; however, in reality we are never going to just send back a text string. We are either going to send back a HTML designed to be rendered in the browser or we are going to send back JSON data designed to be consumed and used by code.

To serve up HTML and JSON, we are going to once again use the `res.send()` method but we would change what we would pass inside as the argument.

### HTML

We can add the html directly within the argument string as seen below using `<h1>` element tags:

```
1. app.get('/html', (req, res) => {  
2.   res.send('<h1>About Page</h1>')  
3. })
```

### JSON

We add an either an object or an array as the value to the `.send()` method:

```
1. app.get('/object', (req, res) => {  
2.   res.send( { name: 'Elle', age: 20 } )  
3. })
```

```
1. app.get('/array', (req, res) => {  
2.     res.send( [ { name: 'Lisa' }, { name: 'Jason' } ] )  
3. })
```

Express is going to detect that we have provided either a object or an array as the send value and it will automatically go to stringify the JSON for us and it is going to get sent to the requestor in the JSON format correctly.

To conclude, we can use the .send() method to send back regular text, html and JSON data to the requestor using our web server. It is important to note in reality we will serve up the contents of an entire directory and therefore if we have a very large webpage with a lot of HTML, we would not want to write all of that code inside of a string inside our node server .send() method. We would rather have a separate HTML file which we will serve up instead.

## Serving Up Static Assets

Typing HTML in a string in a JavaScript file is going to get out of hand really quickly as we add more to the given web page. We can write our HTML in separate .html files and have our express server to serve them. We can configure express to serve up an entire directory of assets which could contain HTML, CSS, JavaScript, videos, images, and more file types.

We can create a directory called public to store our application assets. Any file that goes in this directory will be served up as part of express server. We can create a index.html which is a special file and the name indicates that it is going to get served up by default and this is the page that is going to end up showing at the root of our website/web application.

### web-server > public > index.html:

We need to teach express how to serve up the contents of a directory and one important information we need is the path to the public folder. This cannot be a relative path, it must be an absolute path from the root of our machine. To get this done, Node provides us with two handy variables called \_\_dirname and \_\_filename.

The \_\_dirname is short for directory name returns the string path for the directory the current script lives in. This will provide an absolute path from the root of our hard drive all the way to the folder containing the script file. The \_\_filename on the other hand returns a similar string to the \_\_dirname; however, it provides path to the file itself (i.e. the directory and the file name). Both of these variables are provided by the wrapper main function.

We can use the \_\_dirname variable to get the correct path to the public folder directory using path manipulation to get the absolute file path for the desired folder. There are two methods we can take, the first is using string manipulation and the second is to use the core Node module called Path that provides

us a tonne of great utilities for working with paths. The path module is cross platform compatible and is a useful tool to use compared to the string manipulation technique.

web-server > src > app.js:

1. const path = require('path')
2. const express = require('express')
3. const app = express( )
4. app.use( express.static( path.join( \_\_dirname, '../public' ) ) )

The path.join() is a function which allows us to return the final path. We pass to it the individual pieces to the path and it does the job of manipulating the string for us. We can use the \_\_dirname to get a starting path and then add a second argument to manipulate that string. The double dots allows ('..') us to move up a directory from the \_\_dirname path. We can move up multiple directories using the forward slash and double dots ('..../') like so which would move up by two directory and so on. To move into a folder we use the forward slash followed by the name of the folder we wish to move into. We can use the path module to easily string manipulate the directory of the \_\_dirname from src directory to the public directory. We can use this to configure express to serve the public directory up.

We can use the .use() method to serve up the directory containing our static assets. The app.use() is a method to allow us to customise our server. We will explore the .use() method in more detail later. Within this method we need to call express.static() which is a function provided by express. Static takes in the path of the folder we want to serve up. This will setup our express server to serve the public directory.

If we now visit the root of our serve i.e. <http://localhost:3000/>, this will serve up our static file from our public directory i.e. the content from our index.html file which we setup in the public folder. We can also visit this html page by providing the explicit path i.e. <http://localhost:3000/index.html> which will provide the same result. It is important to note that index.html has a special meaning when related to web servers in which we can leave this path off of the URL if we are navigating to the root of the website. This ensures that something loads if the user does not explicitly provide a path.

This configuration also means that we are never going to see the route handler if we set one up for the root route i.e. app.get("", ...). This is because our express server is going to work through our application until it finds a match for that route. In the case of express.static() call, it is indeed going to find a match because it is going to find index.html within the public directory which is going to match the root URL because the file has a special name. This would mean the app.get() call for the root, will never actually run and therefore can be removed from the server script as it no longer serves much of a purpose.

If we have more HTML files within the public directory such as a about.html and help.html files, the express.static() function allows us to serve these files by visiting the route for these files without creating them with the .get() method. For example, we can visit the <http://localhost:3000/about.html> to serve the about.html static file and <http://localhost:3000/help.html> for the help.html file.

We have now explored the configuration setup of our Express server to serve static assets to our routes.

## Serving Up CSS, JavaScript, Images and More

We have explored how to setup express to serve up contents from the public directory. In this section we are going to focus on how we can build out the public folder and adding other assets into the mix and using them in our HTML files.

The CSS file needs to be loaded by the browser which means it needs to go into the public folder as this is the only directory setup to be exposed by the web server. We can add our CSS files in the public folder, and we can stay organised by creating a CSS sub-directory and saving all of our CSS style sheets. This does not mean that the CSS will be loaded in the browser. We would need to load the CSS files inside of our HTML files first using the `<link rel="stylesheet" href="/css/style.css">` element within the `<head>` section. This will load in our CSS style sheet and style our HTML elements.

We can use either a relative (`href=".//css/style.css"`) or an absolute path (`href="/css/style.css"`) to link the css file. The absolute path starts with a forward slash / followed by the folder. The forward slash on our local machine would normally translate to our hard drive; however, the forward slash in this case is relative to the web server root which we setup as the public folder.

The same would apply for JavaScript, Images and more files. We can add these to the public directory within their own folders but must load them within our HTML files to view/serve them in the browser.

## Dynamic Pages with Templating

We can use something called a template engine to render dynamic web pages using express. A template engine we will explore is called handlebars. Handlebars is going to allow us to two very important things. Firstly, it is going to allow us to render dynamic documents as opposed to static documents. Secondly, it is going to allow us to easily create code that we can reuse across pages such as a header and footer elements.

We will need to install two npm packages called handlebars (<https://www.npmjs.com/package/handlebars>) and hbs (<https://www.npmjs.com/package/hbs>).

Handlebars is a very low level library that implements handlebars in JavaScript. It can be used wide variety of settings such as the browser, server, desktop application via election or anywhere else that JavaScript can be used. To use handlebars in our express server, the module is not going to allow us to get this done; instead we need to install another handlebar library called hbs which is like a plugin for express which implements handlebars into express. Hbs uses handlebars behind the scenes and so we will only need to install hbs as it will install handlebars as its dependency.

```
:~/.../web-server$ npm i hbs@4.0.4
```

The process of having this setup is very easy. All we need to do is tell express which templating engine we installed and we do this by using a new method on our app express application variable.

### web-server > src > app.js:

```
1. const path = require('path')
2. const express = require('express')
3. const app = express()
4. app.set('view engine', 'hbs')
```

The .set() method allows us to set a value for a given express setting and there are a few. We have a key which is the setting and the value which is the value we want to set for the setting. To setup the view we pass in the view engine as the first argument and the value we use as the second argument is the name of the module for our templating engine. This single line is all we need to get handlebars setup and we can actually use it to create dynamic templates.

When working with express, it expects all of our views, in this case the handlebars templates, to live in a specific folder. This is a specific folder at the root of the project called views.

### web-server > views:

We would create all of our handlebars views within the views folder and we use the extension of .hbs to indicate a handlebars view file. For example we can create a index.hbs view file as seen below. A handlebar file is nothing more than HTML with a couple of nice to have little features for injecting dynamic values. This means we can start to use regular HTML syntax and create a regular HTML document to serve up.

### web-server > views > index.hbs:

```
1. <!DOCTYPE html>
2. <html lang="en">
3.   <head>
4.     <script src="/js/app.js"></script>
5.     <title>Express Server - Serving Up Dynamic Templates</title>
6.   </head>
7.   <body>
8.     <h1>Weather</h1>
9.   </body>
10. </html>
```

So far the above code is static HTML code and is not taking advantage of handlebar's features; however, we can see how express can serve up the views template to our routes. Note that we can remove the index.html file from the public folder; however, it is not to say that we cannot serve static files. If we wish to serve static files, we would place them in the public folder, whereas if we need something more complex with dynamic content then handlebars (views folder) is our best option.

To serve up the hbs template, we need to setup a route, as seen below:

### web-server > src > app.js:

```
1. ...
2. app.get('',(req, resp) => { res.render('index') } )
```

Inside of the anonymous callback function, we have so far explored sending the response back to the requestor via the `res.send()` method which would send back a string, HTML string, object or an array converted as a JSON data. To send a dynamic template to the requestor we use the `res.render()` method. The `.render()` method allows us to render one of our views which we configured express to use the hbs to use handlebars template engine to render our views. The first argument we need to pass to the `.render()` method is the name of the particular view we want to use and we do not need to add the file extension.

By calling `res.render()`, express goes off and gets the file which matches the name passed in as the argument to this method and it converts it into HTML and makes sure that the HTML gets back to the user in the browser. At this point we have a static hbs/HTML document and there is nothing dynamic about the file even though we are using handlebars. We are now going to explore handlebars features to see how we can make the file more dynamic.

The second argument to the `.render()` method is an object which contains all of the values we want that view to be able to access which we can use to make the template dynamic.

```
app.get('', (req, resp) => {
  res.render('index', {
    title: 'Weather App', name: 'John Doe'
  })
})
```

Within the view file we can use the handlebars syntax to make our templates dynamic and access the values accessible from the `.render()` method:

```
1. <!DOCTYPE html>
2. <html lang="en">
3.   <head>
4.     <script src="/js/app.js"></script>
5.     <title>Express Server - Serving Up Dynamic Template</title>
6.   </head>
7.   <body>
8.     <h1>{{ title }}</h1>
9.     <p>Created by {{ name }}</p>
10.    </body>
11. </html>
```

To inject values into a `.hbs` file, we open and close two curly brackets and inside of this we reference the value we are trying to access using the key name. In the above we are accessing the `title` key's value which is `Weather App`. `Weather App` which is the value will be injected into the `<h1>` element and displayed in the Browser.

We now know how to use handlebar templates to render some dynamic values that end up getting provided by Node.js.

## Customising the Views Directory

We are going to explore how to customise how handlebars is setup so that we can use more advanced features of handlebars called partials to make our templates more dynamic. Currently, we have learned how to setup handlebars with our express server; however, currently the views folder must be in a specific location i.e. the root of the project and it must have the folder name of views.

We can change the folder name from views to something like templates; however, this will break our applications as express is not configured to look for a templates directory. The views folder and file path is the default location express expects the views directory to live in but we can customise this by telling express where to look. This will require us to create a brand new absolute path:

web-server > src > app.js:

1. ...
2. const viewsPath = path.join(\_\_dirname, '../templates')
3. app.set('views', viewsPath)
4. app.set('view engine', 'hbs')

We would store the new absolute path in a variable and then tell express to use this path by using another app.set() method call. The setting we are configuring is called views and changing the default to our new viewsPath location. This will now enable us to use the templates folder as the new configured path. We now know how to customise the location and the name of the folder and we can change the location to anywhere we want and in a nested file structure as long as we update the viewsPath to point to it.

Everything that we have learned thus far about Express and its methods can be found on the Express website under the API reference page (<https://expressjs.com/en/4x/api.html>). This is a very useful API reference document which has 5 distinct categories. If we click on express() category we can see the methods we have available on express(). Under Applications we have all the methods we can use on the application object (which in all the above examples we have named app). This contains documentation on methods such as .get(), .listen(), .path(), .render(), .set(), etc. If we click on .set() we would see the parameters we can use with the .set() method and we should see views and view engine settings that we can set.

This is a very helpful document and we should always refer to this documentation to understand what parameters we can use for each methods we would want to use. Now that we understand how to use the documentation for Express, we should have a feel for how exactly the documentation works and how we can use it to discover new features and customise our own express application further.

## Advanced Templating

Partials as the name suggests allows us to create a little template which is part of a bigger webpage. This is where partials come from. We can think of part of a webpage that we are going to reuse across multiple pages in our website. This will be things such as headers or footers where we want the exact same thing showing on every page to give our website a nice and unified feel. Partials is going to allow us to create, maintain and reuse markup throughout our website very easily without having to copy the markup between all the pages in our website.

To work with partials, we need to load in the hbs module and configure it:

web-server > src > app.js:

```
1. ...
2. const hbs = require('hbs')
3. const viewsPath = path.join(__dirname, '../templates/views')
4. const partialsPath = path.join(__dirname, '../templates/views')
5. app.set('views', viewsPath)
6. app.set('view engine', 'hbs')
7. hbs.registerPartials(partialsPath)
```

Once we load in the hbs module, we need to tell handlebars where we are going to put our partials. Partials are also files with .hbs extensions similar to the template views we have seen previously. We want to keep them separate so that we have one directory for our views and another for our partials. The structure we can adopt is a templates directory which has two sub-directories called views and partials.

To get things working we need to change the viewsPath to locate the template files within the views sub-directory. To tell handlebars where our partials are located we would create another variable to store the partials path location and then use the hbs.registerPartials() method to configure handlebar to look correct path for the partials directory. The configuration is now complete and we can now use partials to create more advanced templates.

Below is a simple partial example on how we can use partials with our view templates:

web-server > templates > partials > header.hbs:

1. <h1>{{ title }}</h1>

web-server > templates > partials > index.hbs:

1. {{>header}}

What goes inside of a partial is not a complete HTML document, instead it is just part of a webpage i.e. part of a mark up. To render a partial within our views file, we continue to use the two curly brackets like we did when we wanted to inject a value into the template in a specific location. For a partial, we need to add one additional character which is the greater than sign > followed by the partial name (i.e. the partial file name without the file extension or the file path) inside of the curly brackets.

If we are using nodemon to automatically restart our node web server whenever there is a change to files, we would need to make one slight change to our nodemon command, as currently this will not load in the new/updated .hbs files which will cause our application to crash. To fix this we would run the same command as before but adding the -e flag which stands for extensions. This allows us to provide a comma separated list of extensions that nodemon should watch. This will now monitor and restart the server for any changes to any .js and .hbs files we make in our project.

```
:~/.../web-server$ nodemon src/app.js -e js,hbs
```

We can continue to inject values into our partial files and this will continue to render dynamic text even when the partials are used inside of our rendered views templates. This allows us to break up our website into different smaller components which we can use across multiple pages of our website. This will allow us to write our code once which is easier to maintain and update in a single location which helps to scale our website should it grow in complexity.

## 404 Pages

If someone provides a URL that we have not explicitly setup in our Express server, we can show a 404 page instead of the generic Cannot GET /... text message coming from Express. We can use the 404 page to render a nice useful HTML along with a link back to the homepage so that the user can find a page that actually exists.

We need to setup another route handler using the app.get() method but place this right at the end just before starting up the server with the app.listen() method.

### web-server > src > app.js:

1. ...
2. app.get('\*', (req, res) => { app.render('404') } )

In the first argument using the wildcard (\*) means to match anything that has not been matched so far. This is why it is crucial to add this route towards the end of our code before we start the server up.

When Express gets an incoming request, it starts to look for a match starting from the top of our code working its way down in order. Therefore, it is going to look for the incoming route at the public directory first to see if there is any files matching the request to server and if it does not find any matching file it continues down the list looking at the next handler for a match. If it finds a matching handler with the incoming URL request, Express will render that handler's response and will not continue to look at the other routes below because it has found a matching route.

Where an incoming request does not match anything within the public directory and then all the routes defined, Express will finally come to the last defined route which contains the wildcard, which means everything is a match, and as a result this will return our 404 page back as a response to the requestor. This is all it takes to setup a route handler for a 404 page in Express.

We can take this further, and add another route combining the wildcard. This route must come before the final all capturing wildcard route to work:

```
app.get('/help/*', (req, res) => { app.send('Help article not found.') })
```

If we now follow the url link for example to <http://localhost:3000/help/test/> we will now see a more specific 404 message. We can therefore use the wildcard to match every request or match a bunch of requests that matching a specific pattern, in the above example matching anything after /help/ as long as it was not matched previously.

## Section 7

# ACCESSING API FROM THE BROWSER

### Introduction

We have now learned on the backend how to take an API address and convert it into useful data such as a forecast and on the front end we know how to get our web application up and running in the browser using a Express web server. The real question is how do we integrate these two together.

What we need is the browser to be able to communicate with the server, passing an address along. The server needs to convert that address into a useful data such as a forecast and pass it back to the browser so that the browser can actually render that data to the screen.

We are going to learn how to create our own HTTP JSON end points with Express.

## The Query String

We have already explored query string when we looked at making a http request to the geocode API. We used a URL and added a query string to provide the access token along with one of the options called limit which limited the return results to a single result. We can also use a query string with the end point we can create to accept the address.

Below is an example of using a query string with our express routes:

web-server > src > app.js:

```
1. ...
2. app.get('/products', (req, res) => {
3.   res.send( { products: [] } )
4. })
```

In the above example, we created a end point that sends back products to be displayed in the browser on a e-commerce website. We still obtain request and response and we send back some JSON data using the `res.send()` method. This will return back an empty static products array whenever we visit <http://localhost:3000/products>.

If we wanted to add some form of search feature, we would add a query string which get provided at the end of the url, starting them off with a question mark ? and then key=value pairs to pass additional

Information to the server. So in a e-commerce website we might want users to be able to search amongst the list of products for a specific one and for that we could setup a search query argument for example `search=games` (i.e. <http://localhost:3000/products?search=games>). Since we are the developer creating the backend, we can choose to support as many or as little query string options as we would like. We can add more key=value pairs into the query string by separating them with the & sign for example <http://localhost:3000/products?search=games&rating=5>. This would pass two pieces of information along to the server and the question now becomes, how does the server actually get that information?

The information passed along the query string is available to use inside of our Express route handler. We have both `req` and `res` and we have so far used the response to send back something to the requester. Information about the query string lives inside of the `req` argument. The request object has a `query` property on it which is also an object that contains all of the query string information.

#### web-server > src > app.js:

```
1. ...
2. app.get('/products', (req, res) => {
3.   console.log(req.query)
4.   res.send( { products: [] } )
5. })
```

If we log the `req.query` property object to the terminal console, we can see what key values are passed through our url to our server within the terminal console.

The query string would be parsed by express and the data is made available in the `.query` property object and we can get the value of a specific key:

web-server > src > app.js:

```
1. ...
2. app.get('/products', (req, res) => {
3.   console.log(req.query.search)
4.   res.send( { products: [] } )
5. })
```

In the above we can narrow by the value by entering the key name such as `search`. Working with `req.query` is how we are going to access those additional values passed along with a request. Express does not provide us a way to force a given query option to be provided. If we wanted for example the `search` to be a value that must be provided but we do not need to provide the `rating`, we would have to use `if` statement to add a little conditional logic inside of our callback function.

web-server > src > app.js:

```
1. ...
2. app.get('/products', (req, res) => {
3.   if (!req.query.search) {
4.     res.send( { error: 'You must provide a search term' } )
5.   }
6.   res.send( { products: [] } )
7. })
```

We will notice that in the browser we will get back the error object in our browser if we were to search the url <http://localhost:3000/products> without providing a search=value pair, however, in our terminal we would see a error message of "Cannot set header after they are sent to the client" printed.

We will see this error message when we try to respond to a request twice and we cannot do that. HTTP requests have a single request that goes to the server and a single response that comes back. In the above example, we are indeed trying to respond twice i.e. first if there are no search string query we use res.send() method to send back the error, but down below the code continues to run to send res.send() method again to try to send back the other empty products object which is the cause for the error message in the terminal.

Therefore, if we ever see the "Cannot set header after they are sent to the client" error message, this would mean we are sending two responses back when we can only send one. To address this issue all we need to do is add return to the res.send() which will stop the callback function execution.

#### web-server > src > app.js:

```
1. ...
2. app.get('/products', (req, res) => {
3.   if (!req.query.search) {
4.     return res.send( { error: 'You must provide a search term' } )
5.   }
6.   res.send( { products: [] } )
7. })
```

We can also add on an else statement to place the success case code in there, however, the above is the more common method used by most developers within their code.

This is the basic of query strings. The client whether it is us typing in a URL in the browser or us providing a URL via the client side JavaScript, we can setup that query string which gets sent off to the server and the server can use that information with the request and it can send a response back. Currently, we are not using the search term to do anything meaningful, however, we could imagine we have an array of products and we use the array filter method to actually filter.

## CHALLENGE:

**Update weather endpoint to accept address**

1. No address? Send back an error message
2. Address? Send back the static JSON
3. Test your work. Visit <http://localhost:3000/weather> and <http://localhost:3000/weather?address=london>

## SOLUTION:

web-server > src > app.js:

1. ...

```
2. app.get('/weather', (req, res) => {
3.   if (!req.query.address) {
4.     return res.send( { error: 'You must provide an address' } )
5.   }
6.   res.send( { address: req.query.address } )
7. })
```

## ES6 Default Function Parameter

This ES6 syntax will allow us to set a default value for a function parameter should no argument be passed in. Below is a simple function example called greeter which takes in the name as a variable.

playground > app.js:

```
1. const greeter = (name) => {
2.   console.log('Hello ' + name)
3. }
4. greeter('Andy')
5. greeter()
```

```
:~/playground$ node app.js
Hello Andy
Hello undefined
```

We would see undefined printed when we run our function without passing in an argument value.

Therefore, when we pass a name value in the function runs as expected but when no name value into the function we have receive undefined.

The reason we are seeing undefined is because that is the default value of a function parameter if no argument is passed in. If we wanted to address this we could obviously put in an if statement within our greeter function to check if the name value exists and do something if it does/doesn't have a value.

However, we can achieve this with a default parameter which allows us to write less code.

To add a default parameter for a argument value we use the equal sign right after the argument name followed by the default value we would like to use which could be anything from a function, array, object, number, boolean or string:

playground > app.js:

```
1. const greeter = (name = 'User', age) => {  
2.     console.log('Hello ' + name)  
3. }  
4. greeter()
```

:~/playground\$ node app.js

Hello User

If a value is now provided to the function it will be used; however, if a value is not provided then the function will now use the default value for that argument we setup rather than the undefined default.

We can also add a default object when using destructuring:

playground > app.js:

1. const transaction = (type, { label, stock } = { }) => {
2. console.log(type, label, stock)
3. }
4. transaction('Order')

```
:~/.../playground$ node app.js  
Order, undefined, undefined
```

If we did not setup a default value when destructuring and tried to run the function only passing in the type this would cause a JavaScript error of Cannot restructure property of 'undefined' or 'null'. This is because we are trying to destructure undefined when no values are passed in and we cannot destructure undefined. We can address this error by setting a default destructure value, so in the above we fix this by setting a default value of an empty object. We are always going to get an object whether it is passed in or not. This will make sure the code works whether or not an object is passed in as the second argument.

We can take this further and set the default parameter of each individual value:

## playground > app.js:

```
1. const product = { label: 'Notebook', stock: 260 }
2. const transaction = (type, { label, stock = 0 } = { }) => {
2.     console.log(type, label, stock)
3. }
4. transaction('Order')
4. transaction('Order', product)
```

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

```
Order, undefined, 0
```

```
Order, Notebook, 260
```

So in this case, when no values are passed in, the label value will have a value of undefined from the restructure default value of an empty object while the stock will receive a default value of 0 as we explicitly set the default value for this destructure property to 0 when no value is present. When a value is passed in though, these will override the default values.

To conclude, we can use default values for our function parameters which would help our code to work when no parameter values are passed in without having to write a logical if statement.

## Browser HTTP Requests with Fetch

The app.js JavaScript file located within the public/js directory is the client side JavaScript, which is going to run on the browser. The goal inside of this file, amongst other things, is to be able to fetch the forecasted passed information.

To make a HTTP request in the client side JavaScript, we will be using the Fetch API. Fetch is not part of JavaScript, it is a browser based API, which means it is something we can use in all modern browsers but it is not accessible in Node.js. Therefore, the code we write inside of our client JavaScript is not something we are able to use in a backend Node script.

Fetch is a function and we pass in the URL string we are trying to fetch from. In the below example this is fetching from the puzzle API URL which returns a single JSON object giving a random puzzle string:

```
web-server > public > js > app.js:  
1. fetch('http://puzzle.mean.io/puzzle')
```

Running fetch in our client side JavaScript is going to kick off an asynchronous i/o operation much like calling request in Node.js did. This means we do not have access to the data right away, instead we provide a function and that function will run at some point in time when the data is available. With the request function in node we passed a callback as the second argument to the function, with the fetch API it is slight different. We use the .then() method on the return value from fetch and we provide to it a callback function

we want to run. We get access to the response as the first and only argument and within the code block we can use the response to do whatever we want to do such as extracting the data and render it to the browser or just dump it to the client side console.

web-server > public > js > app.js:

```
1. fetch('http://puzzle.mead.io/puzzle').then( (response) => {  
2.     response.json( ).then( (data) => {  
3.         console.log(data)  
4.     })  
5. })
```

In the above we are essentially using fetch API to say: fetch data from the URL we passed in and then run our callback function using whatever is data in the response we got back from the URL.

With the .json() method on response we do not pass in any arguments to it. This is also designed to work with the .then() method which runs a callback function when the JSON data has arrived and has been parsed by the .json() method. We will now have access to the parsed data, i.e. a JavaScript object, as the first and only argument passed in. We can use the data which we dumped in the client side web console.

The .then() method is part of a much larger API known as promises which we will explore in later sections where we will explore promises and its companion async/await in detail when we learn how to connect Node.js to a database.

With a few lines of code we were able to fetch JSON data from a URL, parse that data into a JavaScript object and then do something with it, in the above dumping it to the client browser console.

## CHALLENGE:

### Fetch weather

1. Setup a call to fetch weather for London
2. Get the parse JSON response
  - If error property, print error
  - If no error property, print location and forecast
3. Test your work. Visit <http://localhost:3000/>

## SOLUTION:

### web-server > public > js > app.js:

```
1. fetch('http://localhost:3000/weather?address=london').then( (response) => {  
2.     response.json( ).then( (data) => {  
3.         if (data.error){  
4.             console.log(data.error)  
5.         } else {
```

```
6.           console.log(data.location)
7.           console.log(data.forecast)
8.       }
9.   })
10. })
```

## Creating a Search Form

We can use HTML (and CSS for style) to create a search form which will allow our users to interact with our applications, for example we can create a form allowing the user to input an address and submit the form in order to allow the application to fetch the weather data for the given address.

Below is a snippet code creating a form within the template views:

[web-server > template > views > index.hbs](#):

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.      <head>...</head>
4.      <body>
5.          <div>
6.              ...
7.              <form>
8.                  <input type="text" placeholder="Location">
```

```
9.          <button>Search</button>
10.         </form>
11.       </div>
12.       <script src="/js/app.js"></script>
13.     </body>
14. </html>
```

We now have a very simple form whereby a user can enter a location/address to search and retrieve the weather forecast for. However, if we were to submit the form using the search button, nothing would happen because it is not wired up with our client side JavaScript code we created in the previous section.

#### web-server > public > js > app.js:

```
1. fetch('http://localhost:3000/weather?address=london').then( (response) => {
2.   response.json( ).then( (data) => {
3.     if (data.error) {
4.       console.log(data.error)
5.     } else {
6.       console.log(data.location)
7.       console.log(data.forecast)
8.     }
9.   })
10. })
```

```
11. const weatherForm = document.querySelector('form')
12. const search = document.querySelector('input')
13. weatherForm.addEventListener('submit', (e) => {
14.     e.preventDefault()
15.     location = search.value
16.     console.log('testing')
17. })
```

We use `document.querySelector` method passing in a string argument to select a html element we want to style, similar to how we select elements in our CSS file. In the above we selected the form element. What comes back from this method is a JavaScript representation of that element and we can use that to manipulate the element or to do things when a user interacts with the element. We store this in a `const` variable so that we can reference the variable and access it at any point in the future.

We add an event listener onto our element and there are all sorts of event listeners available such as hover, scrolling, clicking a button or submitting a form. The `.addEventListener` method takes in two arguments, the first is a string which names the event we are trying to listen for and the second is a callback function which runs every single time an event occurs. In the above the name of the event is `submit` and the callback function in the above will log testing to test that the submit event listener is working.

The default behaviour of submitting a form is to reload the page which made sense a long time ago before we had access to good client side JavaScript. We can now update the fetch function to fetch the data and we can dynamically add the data on our web page so that we do not need to refresh the page, thus

preserving the page so that it does not confuse the user. To prevent the page from refreshing on submission and instead having our callback function to run, we can grab the argument that is provided to our event callback which is an event object that can be called event or more typically called e and we can run a single method on this event object. This only method we are ever going to use on the event (e) object is called .preventDefault() which prevents the default behaviour of the page refreshing allowing us to do whatever we want by letting the function run. This will therefore now allow us to use the search form and the text testing will appear in the console and the search input preserved without refreshing the page.

The const search variable stores the input text which we can get access to when the form is submitted. We can use this value to fetch the forecast for the location value passed in making our code dynamic. The .value allows us to get the value from the search variable which we now store in a new variable called location. Instead of logging the testing we could console.log(location) instead to log the input from the input field provided by the user.

We can now update the code so that we can dynamically update the URL string for our fetch function so that the location passed in the input field is also added to the URL.

The fetch function can now go into the callback function of our eventListener and update the URL string to take advantage of the location variable we created which stores the users input value.

web-server > public > js > app.js:

```
1. const weatherForm = document.querySelector('form')
2. const search = document.querySelector('input')
3. weatherForm.addEventListener('submit', (e) => {
4.     e.preventDefault()
5.     location = search.value
6.     fetch('http://localhost:3000/weather?address=' + location).then( (response) => {
7.         response.json().then( (data) => {
8.             if (data.error) {
9.                 console.log(data.error)
10.            } else {
11.                console.log(data.location)
12.                console.log(data.forecast)
13.            }
14.        })
15.    })
16. })
```

We now have a fully functional application with all the important features setup. We can customise the application to display the results in the browser using HTML and CSS rather than printing to the terminal.

## Section 8

# APPLICATION DEPLOYMENT

### Introduction

Currently we have learnt how to create a Node.js application which runs locally on our machine; however, the problem is that it is running locally and no one else can access the application.

In this section we are going to explore how to deploy our Node.js applications to a production server so that anyone with the app URL can pull up the application in the browser and start to use it.

To get this done we are going to view three popular tools which are Git, Github and Heroku. Git is going to allow us to put our application under version control which is going to allow us to track the changes to our app code over time. Github is going to allow us to backup our source code and collaborate with other developers. Finally, Heroku is going to give us everything we need to deploy our app to a production server so that our app is accessible via a single URL.

## Joining Heroku and Github

Github is the very popular software development platform which gives us all of the tools necessary to manage software development projects whether we are working by ourselves or whether working with others as a team. We can sign up on <https://github.com/>

Heroku gives us all of the tools and infrastructure needed to actually deploy our Node.js applications to a production ready server. Heroku is a application deployment platform which will take our application code and deploy to their servers. Heroku is not Node.js specific and we can use it with other languages such as PHP, Python, Java, etc. We can sign up on <https://www.heroku.com>

At the end of the day both Github and Heroku are businesses which mean they do have paid offerings but both also have generous free tiers allowing us to get started without needing to pay any money or provide any payment credentials. The free tiers are going to be more than enough to explore the tools and for us to actually get our applications deployed to production.

Finally, we are going to install Heroku command line tools which are going to provide us access to various commands that we can use from the terminal to do things like deploy our latest code changes to the production server, allowing users to see the latest changes to the application. The tools can be accessed by heading over to <https://devcenter.heroku.com/articles/heroku-cli>. We can install using a installer or running the code in the terminal. The easiest option is to install using the installer.

Once the installer has completed, we can test out the new commands in our terminal by running a few commands:

```
:~$ heroku -v
```

This -v flag will print the heroku version we have installed including the node version on our machine. If we get an error message, this would mean that the heroku CLI was not installed correctly on our machine.

```
:~$ heroku login
```

This login command is going to link our commands we run from our terminal with our heroku account so that we can actually manage our projects from within our terminal. This command is going to ask us to open up a new tab in the browser by pressing any key. In the new tab within the browser we can login to the heroku CLI using our account credentials. Once we have logged in successfully we can close the browser tab and from the terminal we should receive some new output informing us that the logging in is done and which heroku account email we are logged in as.

We should now have a GitHub and Heroku account all setup along with access to the Heroku Command Line Interface (CLI) and are now ready to deploy our Node.js applications projects to the web.

## Version Control with Git

Version control allows to manage the versions of our application over time. As we add new features or change code, with version control we can track those changes. We can create what are essentially save points along the way for the various versions of our application. This is useful because if we deploy a new feature which has a bug, we can revert back to the previous state while we continue to work on fixing the new feature and when fixed we can deploy back to the existing application. This is a very simple workflow but without a version control we are not going to have an easy way to revert back to a previous working state of our application as we are only going to have the broken code version of our application on our local machine. With version control we are able to revert back to a previous working state within a few seconds. Version control provides us flexibility and confidence to allow us to experiment with our code but always get back to that working state we had before without wasting time and resources.

There are other version control tools out there, however, Git is by far the most popular and most widely used version control tool because it is super fast and easy to work with and we can use it regardless of the programming language we are working with. We can go to the following link <https://git-scm.com/> to install the Git tools to allow us to integrate version control into our programming projects. We can use the installer for our OS system to install the version control on our machine. The Pro Git by Scott Chacon and Ben Straub is available to read online for free on the same link and is a recommended read to understand how to use Git and version control as you continue on with your development career.

If we are installing on Windows, we would need to click on the option to install something called Gitbash on our Windows machine. This would allow us to perform operations that we otherwise could not have run

on Windows. After completing the installation, we can open up a new terminal and within the terminal we can run the following command:

```
:~$ git -v
```

This should print the git version we have installed on our machine to indicate that we have successfully installed git and can now run various git commands within our terminal on our machine.

## Exploring Git

We need to start up git in every project where we want to use it. Once git is initialised, we can start to add files into our projects and this is where Untracked Files, Unstaged Changes, Staged Changes and Commits terminology will come into play.

### Untracked Files

By default git does not track files we add to our application, we are going to have to run commands telling git that we want to track specific files. If we add new files to our project they will appear as untracked files.

### Unstaged Changes

Where we make changes to a file that git is already tracking, i.e. files that have previously already been committed, these files will come under Unstaged Changes.

## Staged Changes

We use a git add command to allow us to take one or more files from either Untracked Files or Unstaged Changes and bring it over to the Staged Changes. The files contained in the Staged Changes are files that are going to be included in the next commit i.e. the next save point.

## Commits

Commits are like save points, and this is a two part process. The first step requires us to add files to the Staged Change. The next step in the process is to actually create a commit using the commit command. The commit command is going to take all of the files in the Staged Changes and bundle them up into a single Commit and each Commit has a unique identifier. We can now revert back to a previous commit to get the application back to the state at that point in time, akin to a save point.

## **Integrating Git**

To use git we first need to initialise it within the project we are trying to use it. This would mean we need to change directory into the root project folder and then run the following command:

```
:~projectDirectoryName$ git init
```

This will print message in the terminal of Initialized empty Git repository in followed by the path of our project folder. This will create a new .git folder where git will store everything that makes up our git project. This is a directory we are not going to manage ourselves as this is managed and altered by git commands.

A repository is defined as a place where things are stored. Therefore, a Git repository is nothing more than a place where things related to git are stored. We now have a local repository in a place called .git on our local machine. As we integrate with Github and Heroku, we will have remote repositories where our code is backed up. For example, Heroku needs the code to deploy to the server and Github needs the code so that it can actually use it part of its software/project management platform.

In Visual Studio Code we will notice all files within this project folder are appearing green. These are all Untracked Files. We will also notice that the .git file is not showing within the project directory and this is because VS Code hides that directory by default because it is not something we are manually going to change or manage as this will be managed/changed by git and git commands.

Now that we have git initialised we can now start to use the add and commit commands to actually track things with git.

```
:~projectDirectoryName$ git status
```

The git status command prints to the terminal the current status of our project, which is going to show files in the Untracked Files, Unstaged Changes and Stages Changes categories.

There is a single folder in our project we would not want git to track and that is the node\_modules

directory. We do not want to track this folder because this is a generated directory that we can always recreate by running npm install command which is going to use the contents of package.json and package-lock.json to create that directory back to its exact state. We can create a new file in our project repository root called .gitignore. This name is very important and must be exactly spelt correctly. In this file we can list our all the files that git should ignore:

web-server > .gitignore:

1. node\_modules/

In VS Code the files we have listed in our .gitignore file will now be greyed out to indicate that while this file exists it has been ignored by version control.

```
:~projectDirectoryName$ git add src/  
:~projectDirectoryName$ git add src/app.js  
:~projectDirectoryName$ git add .
```

We can use the git add command to add files to the Staged Changes area. With this command, we need to list all the files we wish to add. We can list out either individual files (as seen in the second example above) or an entire directory (first example above). Git provides a shortcut to add all files to the commit staging area as seen in the third example above. Alternatively we can use the -A flag to add all files. We can run git status to view all the files within the staging area and any files in the untracked files area.

```
:~projectDirectoryName$ git commit -m "initial commit"
```

Finally, the git commit command allows us to commit our files within the Staged Changes. When running this command we must provide a commit message using the -m flag followed by our message in double quotes, describing what exactly has changed. Initial commit is typically used for the very first commit.

When we run this command we see a bunch of output logged to the terminal letting us know how many files are being changed.

We will now notice in VS Code all of the files will now go back to their default colour i.e. white because all of the files are no longer new/untracked files. As we make changes to our files the colour of these files will change for example changes made to committed files will show up as orange in the file tree to indicate as modified Unstaged Changes. If we run git status command this will also show files that we have previously committed for version control but have been now been modified since the last commit.

We now have a local git repository with all of our commits. We can use the clear command to clear the output in our terminal. We can now take these commits and send them off to the third party services we want to use such as Github and Heroku so that they can access our latest code.

## Setting Up SSH Keys

To transfer our code between our machine and other third party service servers in a secure way is to use

Secure Shell (SSH). This provides us with a means of securely communicating with another machine. We would setup something called a SSH key pair. This is a set of two files which we would use to facilitate a secure communication. To generate these files we need to run a few commands in our terminal. If we are on Mac or Linux we can continue with our standard terminal, however, on Windows we need to use the gitbash programme as these commands are not available in the standard Windows command prompt and we need gitbash to access them.

```
:$ ls -a -l ~/.ssh
```

The ls command allows us to list out the contents of a directory which is going to allow us to check for existing keys. The -a flag make sure that even hidden files and folders show up, these would be files that start with a dot . in front which are also known as dot files. The -l flag will make the format much easier for us to read as it will list everything out as rows/top to bottom rather than columns/side by side. Finally, we provide the path to the folder of which we are trying to print its contents. The Tilda (~) is a shortcut for our user directory and then we are looking for the .ssh folder.

If we do not have this folder, the command is going to fail and return an error of directory not found which is ok. If the folder does exists it will inform us if there are any files within these folders. The single dot represents the directory and the double dot represents the parent directory. If we have a file called id\_rsa and another called id\_rsa.pub this would mean we already have a set of SSH keys and we can choose to

use those instead of creating new ones. There are no need to create new ones if we already have some in place.

If we have no id\_rsa and id\_rsa.pub files or event the folder, we can run some other commands to create some SSH keys we can use.

```
:~$ ssh-keygen -t rsa -b 4096 -C "email@email.com"
```

The ssh-keygen command is going to allow us to create a new SSH key pair. The first argument of three we would provide is the -t which stands for type. There are various protocols we could use but in the above we are using the very popular rsa protocol. The rsa is the initials for the three creators of the algorithm.

The second argument -b stands for bits. We can specify how many bits for this key and want enough to be secure. The most common value is 4096 bits.

The final argument -C argument stands for comment. This allows us to create a comment for the key such as a label for our SSH key pair. It is common to use our email address within this argument string.

Running this command will bring us through a little wizard which is going to ask us a few more questions. The first is to enter the file we want to save the key and we can see the default being the .ssh/id\_rsa which will create the id\_rsa file within the .ssh directory within our user profile folder. Pressing enter on our keyboard will accept the default value shown.

The second question is to enter a passphrase, we can press enter on our keyboard to use the default which is no passphrase.

The final question is to enter the same passphrase to confirm the passphrase in which case we can press enter on our keyboards again to accept our passphrase and then the SSH key will be created.

We would now rerun the first command again:

```
:~$ ls -a -l ~/.ssh
```

This will now show us that we have two files within the .ssh directory which are id\_rsa and id\_rsa.pub. The first file is a secret file which we are going to keep on our machine and will never share this key with anyone. The second file is a public file which we can share with third party servers so that we can secure the communication between our machine and the third party servers.

Finally, we need to run the following command

```
:~$ eval "$(ssh-agent -s)"  
:~$ eval $SSH_AGENT_PID
```

The first command is for Mac/Linux while the second command example is for Windows. This command is going to try to start up SSH agent, and if it is running it will print the process id. After running this command the last thing we need to do is to register the private key file.

Mac command:

```
:~$ ssh-add -K ~/.ssh/id_rsa
```

Linux/Windows command:

```
:~$ ssh-add ~/.ssh/id_rsa
```

We need to provide the path of the private key file. When we run the command we should see the Identity has been added and now when we try to facilitate a SSH communication we will be able to do so securely using our key pair. At this point we have not actually used the key pair to do anything; however, we can now use it to push our code to Github servers.

## Pushing Code to Github

Now that we have our SSH keys in place, we are now ready to share our code with third party services. In this section we will explore pushing our code to Github servers. We will need to sign into our <https://github.com/> account. So far we have explored creating a local repository on our machine. We also now need to create a repository for the project on Github, which will allow GitHub to get access to our code so that it can show it in the user interface. Creating a repository on Github is very easy and we will fill out a simple form to provide a repository name which we could name anything we would like.

Once we have created a repository on Github, Github will provide us with some instructions on the various

ways we could get started. We could either:

- a) create a new repository on the command line and send that to Github; or
- b) push our existing repository from the command line to Github; or
- c) import it from another repository/version control system

In the most case we would want to use the second option as we would already have a repository with commits locally on our machine and we simply want to send it to the Github servers. There are two commands we must run in the terminal within our project directory location. The first is the git remote add origin command followed by a long URL:

```
:~projectDirectoryName$ git remote add origin git@github.com:repositoryPath.git
```

The git remote command allows us to manipulate our remote. A remote is nothing more than a version of our project hosted somewhere else. Therefore, we are going to have a version of our project hosted on [github.com](https://github.com) servers. The next tells git how we are going to work with our remotes and in this case we are trying to add them (for example we can use remove to remove a remote) because we are setting up our very first remote. The next is the name for the remote which can be any value we would like but by default the remote wold have the name of origin and is a common convention we will see across the web. Finally, we have a long URL which contains our Github username followed by the repository name we created on Github. When we run this command we are not actually sending our code to Github. All this command

does is setup the channel of communication (we can think of it as setting up a contact on a phone i.e. a contact name and their phone number).

The next command will push our project code to our remote.

```
:~projectDirectoryName$ git push -u origin master
```

The git push command allows us to push our commits up to a given remote. We provide the remote name which we named origin. The master is the default name for what's called a branch in git. Branching is a much more advanced subject and will not be covered in this guide. The -u flag allows us to set the upstream which is essentially the default which we are setting to origin as we are going to push more often to this remote. So in the future after running the above command once, we can use git push command to send our latest commits up to Github.

Before running these code we need to finalise our SSH configuration. We need to take the public key file and give that to GitHub so that it can create that secure connection. We do this in our account settings. In the left hand side bar we should see SSH and GPG Keys. We can click on this and add a new SSH Key. This will ask for two things, first the Title to identify this key and the next is the contents of public key file. To get the contents of that file we can run the following command:

```
:~$ cat ~/.ssh/id_rsa.pub
```

The command simply concatenates the content of a file out to the terminal i.e. it is going to print the contents of our public key file. We follow this by the path of the public key file to actually print to the terminal. We will get a really long string which starts with ssh-rsa and it ends with the value we put for the comment (i.e. our email address). We need to copy all of this text to the clipboard and paste into the Key textfield in Github. Clicking the Add SSH key on GitHub will save our SSH key to our account.

We can test our connections by running the following command in the terminal (if on Windows, we should make sure we run the command on git bash):

```
:~$ ssh -T git@github.com
```

This command is going to test our SSH connection to the GitHub servers and if it works we know that our key has been setup successfully and we can push our commits up. If this command does not work then we know that we have a problem. The terminal may ask us to confirm that we wish to continue connecting to the GitHub server even though it cannot authenticate the host server, we can simply enter yes and continue. We should see a message of:

Hi githubAccountName! You have successfully authenticated, but Github does not provide shell access. If we see this message, this is a good thing because it means we are able to correctly authenticate with Github and we are now ready to push our code up to Github using SSH with the terminal commands. Setting up the SSH key and configuring it with Github is something we only have to do once. It is setup and we can use it for all of our future projects. We should now see our project files in the Github repository.

## Deploying Node.js to Heroku

Just like with Github, we are going to create a new remote and we are going to use the push command to push our code up to this remote. There are a few changes we would have to make to our application code to allow it to work with Heroku. The great thing about Heroku is that we can manage our applications from the terminal and we do not need to touch the Heroku web application at all. We installed the Heroku CLI tools and we can use those commands to create our applications, manage them and to push new versions of our application up to the production environment. We need to navigate in the terminal to our project directory and run the following commands:

```
:~projectDirectoryName$ heroku keys:add
```

When we run this command, it is going to look through our SSH directory and ask us which keys we want to upload. When we select our key (if we have more than one) we can confirm the upload of our key and this will now make our SSH key associated with our Heroku account.

```
:~projectDirectoryName$ heroku create lastname-weather-application
```

We would want to run this command at the root of our project and this will create our heroku application. After the create we can specify the name for the project and if we do not specify one heroku is going to generate a random name for us. This needs to be a unique name not against your own account but also

against all heroku applications. We can use our last name to make sure this is unique. This command is going to create a new application on the heroku servers and it is splitting our two URLs. The first URL is a location where we can view our application (i.e. live url for the app) and the second is the URL for the git repository where we should be pushing the code we want to deploy. We are now ready to get our code up on the heroku servers.

Before we upload our application code to the heroku servers, there are a few very important changes we should make so that heroku can actually know how to run our application. We need to provide it with some basic instructions of what to do when it gets our code.

The first change is the package.json file. In order for heroku to start up our application, we have to tell it which file to run. From the terminal locally we have been telling node to run src/app.js file and what we would want heroku to do as well. We do this by specifying it within the scripts:

#### web-server > package.json:

1. ...
2. "scripts": { "start": "node src/app.js" }

The script is a key:value pair whereby the key is the script name and the value pair is the command to run in the terminal. The key must be named start which will tell heroku how to start up our application. The command will tell heroku which file to start the application correctly. The script is not limited to heroku and

we can run them locally for ourselves by running the command in our terminal within the root project directory using the npm run command followed by the script name.

The second change we need to make is to the src/app.js file:

web-server > src > app.js:

1. ...
2. const app = express( )
3. const port = process.env.PORT || 3000
4. ...
2. app.listen(port, ( ) => { console.log('Server is up on port' + port) } )

When we call app.listen() we specify the listen on port 3000. We have to change this because heroku is going to provide us with a port value that we have to use when our app is running on heroku. This is not a static value we can hardcode in the project as this is a value that changes over time and it is provided to our application via an environment variable. An environment variable is a key:value pair set at the OS level. In this case heroku sets one for the port where the value is the port number to use.

We create a new const variable that will store the port number provided by heroku. We use process.env to access the environment variables and we access the PORT to set our port variable to the port provided by heroku. If we now run our code locally things are going to fail because the process.env.PORT will not exist on our local machines; however, we can fix this by using the logical OR operator in JavaScript and provide

a default fallback of port 3000 if the first value does not exist. Instead of passing 3000 we can use port.

The final thing we need to address before we are ready to push our code up to heroku is to fix our client side JavaScript.

web-server > public > js> app.js:

1. ...
2. `fetch('/weather?address=' + location).then( (response) => {...} )`

Within our fetch request, the URL string passed in cannot point towards the localhost as we are no longer using our local machine but are using the heroku server. To address this issue we want to remove the domain completely so that if we are on our local machine we want to make the request to localhost but if we are on our special heroku app URL we wan to make the request to that URL.

We remove the domain and just leave the route extension. This will make sure that the domain is correct whether using the app on heroku or localhost followed by the route for our API request.

Now that we have made the following three changes to our code, we can now commit and push our code up to heroku. From the terminal we can use the following command to push our code to heroku:

`:~projectDirectoryName$ git remote`

When we created our application using heroku, it setup our remote for us. Using the above command we

a default fallback of port 3000 if the first value does not exist. Instead of passing 3000 we can use port.

The final thing we need to address before we are ready to push our code up to heroku is to fix our client side JavaScript.

web-server > public > js> app.js:

1. ...
2. `fetch('/weather?address=' + location).then( (response) => {...} )`

Within our fetch request, the URL string passed in cannot point towards the localhost as we are no longer using our local machine but are using the heroku server. To address this issue we want to remove the domain completely so that if we are on our local machine we want to make the request to localhost but if we are on our special heroku app URL we wan to make the request to that URL.

We remove the domain and just leave the route extension. This will make sure that the domain is correct whether using the app on heroku or localhost followed by the route for our API request.

Now that we have made the following three changes to our code, we can now commit and push our code up to heroku.

When we created our application using heroku, it setup our remote for us. Using the below command within our terminal, we can view all of our git remotes created:

```
:~projectDirectoryName$ git remote
```

We should see one called heroku and another called origin which was setup when we created our Github repository remote. To deploy all we have to do is use the below command:

```
:~projectDirectoryName$ git push heroku master
```

This command is going to push the master branch i.e. latest commits to our heroku git remote. When heroku sees that new commits have been pushed, it is going to deploy our application again. This will install all of our dependencies for our application and then it is getting the process kicked off which is going to allow us to view our server when we refresh things in the browser. If we now navigate to the URL where our application lives we should now see our application replacing the Heroky welcome screen.

We now have our application deployed to a production environment. We now have a public URL that anyone in the world with an internet connection would be able to visit to view our application. If we go out to purchase our own domain name, we can configure our DNS records to work with our heroku application and have a completely custom URL rather than the [.herokuapp.com](#) extension. There are some great documentation articles that in the heroku dev centre that can walk you through that process.

We now have an application deployed to a live production server accessible via the web.

## New Feature Deployment Workflow

When creating a new feature to our application code we would want to test the feature first on our local machines to ensure that it is working correctly without breaking the application. Once we are happy with the new feature, we are now ready to push our changes to Github and Heroku so that they contain the latest code for the application.

We would first push our code to Github using the following commands:

```
:~projectDirectoryName$ git add .  
:~projectDirectoryName$ git commit -m "commit message"  
:~projectDirectoryName$ git push
```

Git push is the same as us writing git push origin master but in shorthand form as we set our git push command to use this remote as the main most used remote to push our code to.

Now that we have our code on Github we can push our code to Heroku using the below command:

```
:~projectDirectoryName$ git push heroku master
```

This will redeploy our application on heroku using the latest code and our changes will be made available in the production server. This is the new workflow for deploying new features to both Github and Heroku.

## Avoiding Global Modules

Within our package.json file we can create scripts which we can run using node rather than typing out the whole command in the terminal. For example we can have a script for nodemon:

web-server > package.json:

1. ...
2. "scripts": { "dev": "nodemon src/app.js -e js,hbs" }

We can run the script using the npm run command followed by the script name:

```
:~projectDirectoryName$ npm run dev
```

As we collaborate with other developers, they will be able to use the scripts that we have in our projects i.e. they can use the dev script to start up a local development server easily. If there are commands we are using often in a project, it is best to create a script that runs it so that it is reusable and accessible to everyone. However, the catch to this solution is that the only reason the dev script will work is if we have installed the nodemon dependency as a global module on our machine. When we have global modules installed, it is difficult for other people to know that they need to install them. The problem with global modules are that they are not local dependencies. So if we are using them in a specific project, it is best to try to install everything locally.

To make sure our local dependencies and scripts work right out of the box when someone downloads our

code from a Github repository without the need to install global modules, we should install our dev dependencies as local dependencies to our project. To do this we can run the following code...

Uninstall global dependencies:

```
:~projectDirectoryName$ npm uninstall -g nodemon
```

This command is going to remove all global modules and the nodemon command will no longer work globally in our terminal. The next command will install nodemon dependency as a local development dependency:

```
:~projectDirectoryName$ npm install nodemon@1.19.1 --save-dev
```

When we install something as use --save-dev flag, this lists the local dependency as a dev dependency in our project. Within package.json we will see a dependencies object listing all of our application dependencies as well as a new devDependencies object listing all of our development dependencies. Development dependencies are dependencies we only need on our local machines while we are developing and these dependencies are not installed in our production environment, which would mean something like nodemon will not be installed on Heroku, which is OK. This is because Heroku will never use the dev script as it only uses the start script to start our application. By installing nodemon as a dev dependency we are saving time for heroku to not install modules that it is not going to use. The dev script should now continue to work out of the box without having to install any dev modules globally.

## Section 9

# MONGODB AND PROMISES

### **MongoDB and NoSQL Databases**

MongoDB is a NoSQL database we can use as a database solution for our application to store data rather than saving JSON files. We can view the MongoDB on the web at <https://www.mongodb.com/>.

MongoDB is an open source database and is available to all operating systems. We can install MongoDB on our machines and use the MongoDB native driver to connect to our database from Node.js and we will be able to start the process of writing and reading data.

Note: we can use other databases with Node.js and we are not limited to using MongoDB and we can use MongoDB with other programming language other than Node.js.

MongoDB also provides Node.js developers with a NPM module that can be easily used to read and write from the database.

A NoSQL database is a Non Structured Query Language whereby we do not have the traditional structured tables as found in a SQL database. Below is a visual representation on a high level of the difference between a SQL and NoSQL databases.



Both a SQL and NoSQL databases can have many databases as we want, however, the difference between the two is the way in which we store our data. In a SQL database we have something called Tables which we can have many tables storing various data. In a NoSQL database we have something called a Collection which stores data in a JSON object format. We can have as many collections as we need. In a SQL table we have Rows of records whereas in NoSQL we have what are known as Documents. Each individual item in a table are known as Columns while each item in a Document are known as Fields.

It is important to understand the basic vocabulary for a NoSQL database.

## Installing MongoDB on MacOS and Linux

To install MongoDB on a MacOS and Linux machine we can visit the MongoDB website at: <https://www.mongodb.com/> and clicking on the Try free button which will take us to the MongoDB Download Center page. Clicking on the Server Option we can download and install the MongoDB Community Server tools to run on our local machine which is the free option. The MongoDB Enterprise Server option is a paid server option which provides some additional features for enterprise.

We can select the version, OS and package options before clicking on the download button. We want to make sure the package is set to TGZ package to ensure everything is installed correctly on either Mac or Linux. Once the download is complete we can open the archive file to extract its content.

All these files are what we need to manage our MongoDB server. We have a bin directory which has a bunch of executable which we can use to perform various tasks. The main executable is mongod which we can use to start up the mongoDB database server. Before we start to run the executable, we want to take this directory and move it to a more permanent place on our machines. We can update the folder name to something simple like mongodb and then place this directory to the user folder on our machine.

The next thing we need to do is create a place for our data to be stored. By default MongoDB expects us to create a data directory at the root of our hard-drive and in there it expects a db directory. However, this is not ideal for many users as they are going to run into a lot of different permissions errors. It is much better to create a directory inside of the user folder to store the data.

We can create a new directory in the same location as the mongodb directory and we can call this mongodb-data. Therefore, we have mongodb which contains the executables necessary to start things up and manage our database and we have mongodb-data to contain our databases data.

### Directory Structure:

> User

  > mongodb

  > mongodb-data

Now that we have this in place we are ready to actually start up the database and we do that by running a single command from the terminal. First we need to navigate to the mongodb directory. If we know the path we can cd to it else we can use the following commands:

```
:~$ cd ~
```

This command will change directory to our user directory.

```
:~$ pwd
```

This command will print the current work directory i.e. /Users/YourUsername

We need to list out the path to the executable and run the following command to start the process:

```
:~$ /User/YourUsername/mongodb/bin/mongod --dbpath=/Users/YourUsername/mongodb-data
```

The above command locates the mongod executable from the file path location and we use the --dbpath flag to provide a path to the data folder we created. This is the complete command to initialise the database and gets the database server up and running. The database server is now waiting for connections to the database where the connector can read write from the database such as adding, querying, updating, deleting documents. We will also notice in the terminal the default port 27017 which our MongoDB server is using.

If we now look at the mongodb-data folder this will now have a bunch of files to just initialise things and there are no actual documents as we have not created any just yet.

This completes the downloading and installation of the MongoDB server on our machine and the relevant command we can use to get the server up and running.

## Installing MongoDB on Windows

To install MongoDB on a Windows machine the process is similar to the previous section of navigating to the download page however in the OS we need to select our Windows Operating system and the package ZIP option. When we download the archive file we need to extract the contents of that archive to install MongoDB server executables on our machine.

Again we are going to rename the directory to mongodb and move this directory to a more permanent location. We are going to place this in our user directory for our profile located within Local Disc (:C) > Users > YourUsername.

Within our user profile directory where we moved the mongodb folder, we need to create another directory called mongodb-data which will store our database data.

With this in place we are now ready to actually startup and initialise our MongoDB server from the terminal by running the following command to start the mongod executable:

```
:~$ /User/YourUsername/mongodb/bin/mongod.exe --dbpath=/Users/YourUsername/mongodb-data
```

This command is going to start up the server for the very first time and we will see a bunch of output showing up. The important thing to note is the string displayed 6 lines before the end which show the message of waiting for connections on port 27010 which is the default port that our MongoDB database server is using.

This completes the downloading and installation of the MongoDB server on our machine and the relevant command we can use to get the server up and running.

## Installing a Database GUI Viewer

We can install a MongoDB admin tool which is going to be a Graphic User Interface (GUI) for managing our MongoDB database and the data it contains. The goal of the Database GUI tool is to provide us with a nice visual interface for managing our MongoDB data and is not meant to be a replacement for connecting to our database from Node.js. The purpose is to provide us with an easier way to help us to manage our data as developers as we are building out the application.

We can download the Database GUI Viewer tool called Robo 3T from <https://robomongo.org/>. This is a free tool available on all operating systems. The Robo 3T is the product we want to install.

We can install the downloaded application installer to our machine. Once installed we can open this application. We will see a connection panel where we can connect to our local MongoDB database. Clicking on Create will allow us to create a new connection with different options we can configure in the various tabs. For now all we need to change is the Name. This is a label for the connection and we can call this anything we want. The address should be localhost and the port of 27017. We can click the test button to test the connection before saving it. Once we save the connection we can double click on the connection to connect to it.

We can right click on the connection and click the Open Shell option. This is going to allow us to run a command against mongodb to check that we have installed Robo 3T correctly and is working.

```
db.version()
```

We are calling on db.version as if it were a method on the db object. What we have looks a lot like JavaScript and this is because it is JavaScript. When we are working with MongoDB and interacting with it via the mongodb shell, we are just using JavaScript to manipulate the database. In the above case we are asking for the version of MongoDB that is running. If we run this command we should get the correct version back confirming that our connection was able to connect to the database and that the database was able to respond. To run the code we click on the green play button to fire off the command and down below we should see the output of the version back.

With this we have confirmed two things. Firstly, we confirm that our MongoDB database is up and running correctly and secondly, we confirm that our Robo 3T application is up and running correctly.

Important Note: we do not necessarily need to install a GUI viewer as we can view our data in the terminal/shell; however, these tools help visually view our database in a GUI viewer. Koala is another such GUI tool for MacOS/Linux that can also be used.

## Connecting and Inserting Documents

We can find the MongoDB server documentation on <https://docs.mongodb.com/manual/> and the drivers documentation for the various programming language on <https://docs.mongodb.com/ecosystem/drivers/>. The driver language we are interested in is Node.js.

We are going to use the official mongodb native driver npm package created and released by MongoDB team which we can find on <https://www.npmjs.com/package/mongodb>. This package is going to allow us to communicate with our MongoDB databases to perform CRUD operations. To install the npm package, we can run the following command:

```
:~task-manager$ npm install mongodb
```

We are now ready to use the driver to connect to our database and perform CRUD (create, read, update and delete) operations. We are now going to explore the first operation of creating a document.

task-manager > mongodb.js:

```
1. const mongodb = require('mongodb')
2. const MongoClient = mongodb.MongoClient
3. const connectionURL = 'mongodb://127.0.0.1:27017'
4. const databaseName = 'task-manager'
5. MongoClient.connect(connectionURL, { useNewUrlParser: true }, (error, client) => {
6.   if (error) { return console.log('Unable to connect to database') }
7.   console.log('Connected correctly')
```

8. })

We first need to require the package in order to use it. What comes back is an object which we store on the `mongodb` variable we created. There is one thing we need to initialise the connection and this is known as the mongo client and so we are going to create a new `const` variable for that and access it using the `mongodb` object. The `MongoClient` is going to give us access to the function necessary to connect to the database so we can perform our four basic CRUD operations.

We need to define the connection URL and the database we are trying to connect to. We do that by using two variables. The first is the `connectionURL` which will have the URL string connecting to our localhost server on port 27017 and the second is the `databaseName` which is a string for the name of our database. We type out the localhost ip in full because using `localhost` causes issues such as slowing down the application and things starting to fail. It is uncertain why that is but using the full ip address fixes the issue and the application continues to work as expected. The name of the database can be anything we want it to be.

Now that we have all this in place, we have all of the information we need to actually connect to our database. We can use `MongoClient` to connect to the database server and `MongoClient` has one method that is called `.connect()` which takes in a few different arguments to actually setup the connection. The first argument is the connection URL which we have stored in a variable. The second argument is an options

object which we are going to use to setup a single option called `useNewUrlParser`, setting this to true. The URL Parser that was used originally by default is not being deprecated and it is now required to pass in this option in order for our URL to be parsed correctly so that we can actually connect to the server. The last argument to pass to the `connect` method is going to be a callback function. The callback function is going to get called when we are actually connected to the database.

Connecting to the database is not a synchronous operation, it is indeed asynchronous and will take a bit of time to get that connection setup and the callback function is going to run when that connect operation is complete. Now this can either fail or succeed and we are going to get access to different arguments depending on what happens. This callback is going to get called back with an error or with a client. If we get an error then things went wrong and will contain an error message describing why it could not connect. If the second argument exists, this will mean that things went well and we now connected to the server and we can start to manipulate our databases.

We use the `if` statement to check if there are any errors and if so we return a `console.log()` which will stop the code from executing any further. The code down below only runs when things went well which prints a success message to the console.

We can now try to run our file to see if it connects to our database correctly. It is important to note that we must have a terminal open that is running our `mongod` server to listen for connection.

We can use a second terminal to run our application code to see if it actually connects to our database server and if it does we would expect the success message printed to the console.

Terminal 1:

```
:~$ /User/YourUsername/mongodb/bin/mongod.exe --dbpath=/Users/YourUsername/mongodb-data
```

Terminal 2:

```
:~task-manager$ node mongodb.js
```

If we look at the terminal that is running mongod, we will see all the connections that come and go. For example, connection accepted from 127:0.0.1:00000 #1 (1 connection now open) which is letting us know someone has indeed connected to our database server.

It is important to note that at times we will only have one connection but we will see a message saying that we have more than one open connection, this is because when we connect with mongodb it uses a connection pool. So there are actually more connections that are open behind the scenes, even though we only called .connect() once. This will make sure that our Node.js application can still communicate quickly even if we are trying to perform a lot of operations at the same time.

In the second terminal we will notice that our code is still hanging and it has not brought us back to command prompt, this is because when the connection is open, our node process is going to stay up and running as long as we let it or as long as that connection remains active. We can always shutdown the

process by running control and c on our keyboards within the terminal.

We have now successfully connected to the database and we can now insert our very first document. We can remove the success console.log() message and replace it with more code:

task-manager > mongodb.js:

```
1. ...
2. MongoClient.connect(connectionURL, { useNewUrlParser: true }, (error, client) => {
3.   if (error) { return console.log('Unable to connect to database') }
4.   const db = client.db(databaseName)
5.   db.collection('users').insertOne( { name: 'Alex', age: 25 } )
6. })
```

We use client.db() method takes in a single argument which is the name of the database we are trying to manipulate and in the above example we stored this name in the databaseName variable. By picking a database name and accessing it, MongoDB will automatically create the database for us. This gives us back a database reference which we typically store in a db variable that we can now use to manipulate that database.

With NoSQL database we have collections and so the next thing we need to do before we can insert a document is figure out which collection we are trying to insert the document in to. We can use collections to track all of the distinct things in our application such as a collection for users, tasks, etc.

In the above example we insert a single document into a users collection. We use db.collection() function which expects the name of the collection we are trying to manipulate. From here we can go ahead and call a method on that collection reference to insert a document which is called .insertOne().

The insertOne() method expects an object as the first argument and this should contain all of the data we are trying to insert.

If we now run our script again using node and again the command will hang and show nothing in the terminal which is OK as it means that it successfully connected to the database server. We can use Robo T3 application to see if anything was added to our database. We should now notice a brand new database showing up called task-manager and within this we can see the collections folder and within this we have a users collection. We can now see the data stored in this collection by right clicking that users collection and selecting View Documents option. This is going to grab all of the documents stored in that collection and dump them to the GUI interface. We should have a single document and if we expand that we can view the data stored in that document. We should have three fields, two of which we created i.e. the name and age and the third field is the \_id which was created by MongoDB to store a unique ID for that particular document in order to identify it. Therefore, when we insert a document it is going to have a unique identifier automatically generated for us which is a good thing. In Robo 3T we can view our documents in different views such as the tree view mode, table mode and text mode.

To conclude we now know how to connect to MongoDB using the mongodb npm library and insert a single document to a database collection using our Node.JS application which we could view in Robo T3.

## Inserting Documents

In the last section we briefly touched up on inserting a single document by using `insertOne` to insert our very first document. In this section we will dive into inserting documents in more detail.

The `.insertOne()` command is an asynchronous operation. We do not have a callback registered because it is not completely necessary for the data to actually get inserted, but it is necessary if we want to handle errors or confirm that our operation worked as expected. We would want to do this in a production application and here we would pass in a second argument to the `.insertOne` command.

```
db.collection('users').insertOne( { name: 'Alex', age: 25 }, (err, result) => {
  if (error) { return console.log('Unable to insert user') }
  console.log(result.ops)
})
```

The second argument is a callback function which will get called when the operation is complete. The callback function for insertions gets called with one of two potential arguments. We either have the error or the result. The result is going to contain our data as well as the unique id that was assigned to the document. If the code returns an error then it will print the error message to the console and end the callback function, else the code below will run. The `.ops` is an array of documents inserted and in the above this will be an array of a single document. If we now run the above code we would see the array object containing the single document data inserted into our database including the `_id` value assigned.

The API documentation for the npm mongodb module is very extensive and super useful (<https://mongodb.github.io/node-mongodb-native/2.0/api/index.html>) and use this document to see all of the methods and objects we can use with the driver and what they return back. For example, we can go under Connections/methods/insertOne to view how we can use the .insertOne() method on our collections i.e. what parameters to pass in and what is returned back. This is how we know that we have a .ops array object when working with result, which is one of five properties we have access to, to access the documents that was inserted into our database.

The insertOne in a nutshell, which allows us to insert a single document into a collection.

To bulk insert documents into a collection, we would use the .insertMany() method on collection.

```
db.collection('users').insertMany( [ { name: 'Alex', age: 25 }, { name: 'Ben', age: 30 } ], (err, result) => {
  if (error) { return console.log('Unable to insert user') }
  console.log(result.ops)
})
```

We use an array of objects to insert more than one document to a collection at the same time. Again we can use the documentation to understand how to use the .insertMany() method, what it returns in the callback function and what properties we have access to. It is really important to get comfortable to read documentations so that we can explore new features and customise everything to fit our needs. This

command is very similar to the `.insertOne()` method with the difference of now inserting an array of object rather than a single object and we have access to a `insertedIds` properties for the multiple `_id` properties for each document inserted.

We now know how to `.insertOne()` and `.insertMany()` documents inside of a collection.

## CHALLENGE:

### Insert 3 tasks into a new tasks collection

1. Use `insertMany` to insert the documents
  - `description` (string), `completed` (boolean)
2. Setup the callback to handle error or print ops
3. Run the script
4. Refresh the database in Robo 3T and view data in task collections

## SOLUTION:

### task-manager > mongodb.js:

1. ...
2. `db.collection('tasks').insertMany( [`

```
3. { description: 'Clean the house', completed: true },
4. { description: 'Renew car insurance', completed: false },
5. { description: 'Pot plants', completed: false }
6. ], (err, result) => {
7.   if (error) { return console.log('Unable to insert documents') }
8.   console.log(result.ops)
9. })
```

## The ObjectId

The `_id` field is a field that is automatically created by MongoDB and it stores a unique identifier for each document that is inserted into the database. We are going to quickly explore what ObjectId's are, what are they used and how they are going to play an important role in MongoDB.

The value we see for the unique id is much different to what we would see in a traditional SQL database and this is by design. In a SQL database the server generates the id for new records and it follows a auto increment integer pattern whereby the first document has an id of 1, the second 2, the third 3 and so on.

With MongoDB, the id's are known as GUIDs which stands for globally unique identifiers which are designed differently. They are designed to be unique using an algorithm without needing the server to determine what the next id value should be. Switching from auto incrementing integers over to GUIDs allowed MongoDB to achieve one of its main goals which is the ability to scale well in a distributed system. So we have multiple database servers running instead of one, allowing us to handle heavy traffic where we

have a lot of queries coming in. With GUID's there is no chance of an id collision across those database servers unlike with auto-incrementing integer, it is definitely possible that we could have a user with an id of 1 in one database server and another user with an id of 1 in another database server and we can eventually run into an issue where those id's conflict. With MongoDB we do not run into that problem.

Another great advantage is that we could actually generate the id's for our documents before we ever insert them into the database. Therefore, our database does not need to be the one that generates the id and we can use the `mongodb` library to generate a `ObjectID` of our own.

We can create a new const variable for `ObjectId` and get its value off of a property from the `mongodb` object called `ObjectID`.

task-manager > mongodb.js:

1. `const mongodb = require('mongodb')`
2. `const MongoClient = mongodb.MongoClient`
3. `const ObjectID = mongodb.ObjectID`

We can use destructuring in this use case as we are grabbing a lot of individual things off of `mongodb` and storing them in individual variables:

task-manager > mongodb.js:

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

We destructure variables from an object which is the `mongodb` object that we require. This is a shorthand of grabbing things off of an object.

We can now create our own id by creating a new variable and using the constructor function using the `new` keyword and we call it without passing any arguments:

task-manager > mongodb.js:

1. `const mongodb = require('mongodb')`
2. `const id = new ObjectId( )`
3. `console.log( id, id.getTimestamp )`

This is a function that is going to generate a new id for us. The `new` key is optional because the MongoDB library has a little defensive programming built in to add it if we don't but it is in general good idea to add it in ourselves. We can now do something with the id which we can dump to the console.

```
:~task-manager$ node mongodb.js  
5d892f50d070f10556f766eb  
2019-09-23T20:47:12.000Z
```

What we have is not just one long randomly generated series of characters, there are a few different pieces of information inside of here. Embedded in here for example is a time stamp. We can see the documentation which provides a breakdown of the GUID on <https://docs.mongodb.com/manual/reference/method/ObjectId/>.

The `.getTimestamp()` method on `ObjectId` gets the timestamp that is stored inside of the first 4 bytes for the `ObjectId`. We can use this method to actually print the timestamp out and this method does not need any arguments passed in. Therefore, a timestamp is embedded in the id letting us know when the document was created.

We can now generate our own id using the new `ObjectId()` which we stored in a variable called `id`, when we insert a document by setting the variable as the value to `_id` as seen below:

```
const id = new ObjectId()
db.collection('users').insertOne( { _id: id, name: 'Ellen', age: 40 }, (err, result) => {
  if (error) { return console.log('Unable to insert user') }
  console.log(result.ops)
})
```

We can create our own id and set that to the `_id`, however, this is extra code which is not necessary because MongoDB would generate a `_id` value for us automatically if we do not provide one when inserting our documents.

Finally, we are going to explore how `ObjectId`'s are stored because that plays an important role as well. When we see the `ObjectId` in the terminal, we see the series of characters that make it up. What we have is a function call whereby the string of characters is provided as the first and only argument. This is a

visualisation making it easier to see the ObjectId value because id's are binary data. The reason MongoDB is using binary data over a traditional string has to do with the size of each. By using a binary instead of a string, they are able to cut the size of an ObjectId in half. The id object has an .id property which stores the binary data in the buffer:

```
console.log( id.toHexString( ), id.toHexString().length );
```

```
5d892f50d070f10556f766eb, 24
```

```
console.log( id.id, id.id.length );
```

```
<Buffer 5d 89 2f 50 d0 70 f1 05 56 f7 66 eb>, 12
```

As we can see the binary data is much smaller than the string data as the binary is 12 bytes while the string data is double i.e. 24 bytes. This is the reason we are seeing the ObjectId() function because it is a way to visualise easier for humans what exactly is happening. Therefore, it is providing a string representation wrapped in the ObjectId function to let us know that it is not actually a sting but is what comes back from the function call which would be binary data.

The reason we dived into ObjectId's into detail is because we need to known about ObjectId's in order to fetch our documents by \_id which is something we are going to explore in the next section.

## Querying Documents

We are going to learn how to read documents from our MongoDB database. This is going to bring us from

C for create to R for read in CRUD. We are going to be able to read individual documents by `_id` or any other field and we are also going to be able to fetch multiple documents limiting them to a specific subset for example tasks that have yet to be completed.

There are two main methods we can use to fetch documents from our database. These are `.find()` and `.findOne()` methods. The first method is going to allow us to find multiple documents and the second method is going to allow us to fetch an individual document from the database.

```
db.collection('users').findOne( { name: 'Alex' }, ( error, user ) => {
  if (error) { console.log('Unable to fetch') }
  console.log(user)
})
```

We first specify the collection we want to `.findOne()` document on. The `.findOne()` method accepts two required arguments, the first being an object and the second is a callback function. The object is used to specify our search criteria. The callback gets called when the operation is completed and we get back an error or the actual document object. The document object can be named anything we would like and in the above case we appropriately called it `user`. We can use the `user` object to print to the console.

If we run the code, we should find the one document printing to the console if a document matches the criteria.

We can narrow our search by specifying more fields and values within the object list. If we search for a document whereby the criteria will return no results, we would not see an error because this is a valid operation. Instead, we would see null printing to the terminal. Therefore, searching for a document and not finding a document is not an error because it is giving back what we asked for but there was just nothing to give back.

The `.findOne()` method will always return back one document and if our query matches multiple documents it is only going to return the first result. If we want to search for a specific document, we can always do this by its id. However, it is not enough to provide a string id and this is because MongoDB stores the id as a binary data and therefore we must use `ObjectId` to convert the string representation into binary data in order to search and retrieve a document by its id:

```
const { MongoClient, ObjectId } = require('mongodb')
db.collection('users').findOne( { _id: new ObjectId("5d892f50d070f10556f766eb") }, ( error, user ) => {
  if (error) { console.log('Unable to fetch') }
  console.log(user)
})
```

We can use `.find()` to search for more than one document based on our criteria object. The method call is very similar to the `.findOne()` method. Typically, we are using `find` to search for a field rather than the `_id` this is because it does not make sense to use `find` on `_id` because this will always return back one result. Below is an example of the `.find()` method being used to return all documents where the age is equal to 28.

```
db.collection('users').find( { age: 28 } ).toArray((error, users) => {
  if (error) { console.log('Unable to fetch') }
  console.log(users)
})
```

The `.find()` method is different to the `.insertOne()`, `.findOne()` or `.insertMany()` as it does not take in a callback function as its second argument. With `.find()`, what we get back as the return value is actually a cursor. The cursor is not the data we asked for, it is a pointer to that data in the database. The reason we are getting a cursor back is because MongoDB is not going to assume that every time we use `find`, we always want to get back an array of documents. There are a lot of other things we may want to do such as getting just the first 5 documents or something much different of getting just the number of matching documents and not the document data itself. So when we get a cursor back, it opens up a lot more possibilities. We can explore this by going over to the documentation.

What this means is that `find` returns a cursor and we can use a method on the cursor object for example the `.toArray()` method which converts the cursor into an array of documents. There are many more methods that we could use on the cursor object which we can read up on in the documentation. The `toArray` takes back the callback function where we get the error or our matching documents which we can call anything we want and in the above we call this `users` appropriately.

The advantage of the cursor is that it allows us to fetch back the data i.e. all the individual fields of the

documents back and in that case the `.toArray()` method gets the job done, but if we want to do something simple like count, there is no need to fetch all those records, store them in memory in Node.js only to get a single number back. MongoDB can handle that for us instead of needing to transfer all of the documents across the network, it can just transfer that single integer using the `.count()` method. The count method also takes in a callback function like the `.toArray()` method but it returns an integer of the number of records:

```
db.collection('users').find( { age: 28 } ).count((error, count) => {
  if (error) { console.log('Unable to fetch') }
  console.log(count)
})
```

Therefore, the cursor is a really nice tool for us to use and hopefully the above will demonstrate why we might not always want to get back all of the data back from the database. We now know how to create and read data from our MonoDB database.

## Promises

Promises make it easier for us to manage our asynchronous code and they were designed to solve many of the problems that we run into when using callbacks in our application. Promises actually build on the callback pattern and therefore it is necessary to understand how callbacks work and how we can use them before being able to understand promises. Promises is nothing more than an enhancement for callbacks, making it more easier to manage our asynchronous code. To understand promises, we will compare and contrast it with the traditional callbacks.

## playground > app.js:

```
1. const doWorkCallback = (callback) => {
2.   setTimeout(() => {
3.     callback('This is an error!', undefined)
4.   }, 2000)
5. }
6.
7. doWorkCallback( (error, result) => {
8.   if (error) { return console.log(error) }
9.   console.log(result)
10. })
```

The above is a function that performs some sort of made up asynchronous task and we are using `setTimeout` to simulate a delay. The whole point of using the callback pattern, is a way to allow the caller of `doWorkCallback` to get the results. So we have a callback function which runs when we have either the error or the result.

The `setTimeout` when resolved after two seconds can provide a callback which we pass two arguments. The first being the value when things go wrong and the second is the success value which we are explicitly setting to `undefined` (although we do not need to provide this value and it will be implicitly set to `undefined`).

With the callback pattern we need to add a little conditional logic because this is the only function that runs for both failures and success cases. We use the `return` for the error so that the code will stop running if an

error is returned from the callback, else we will display the result in the console for a success. So in the above we have a very basic example of the callback pattern.

In the above example if we ran the code, the callback would return the error as we defined the error and set the success to undefined but if we changed it to the below, the result would be displayed instead as we provided a value for the success case.

3.       callback(undefined, [1, 4, 7])

This is the callback pattern we have seen many times before and to summarise, with callbacks we can see that the order we call the callback is important i.e.the first being the error and the second being the result and we can see that within the conditional logic to figure out if things went well or if things failed. We are now going to explore the same code up above but now using the promises API.

playground > app.js:

```
1. const doWorkPromise = new Promise((resolve, reject) => {  
2.     setTimeout(() => {  
3.         resolve([1, 4, 7])  
4.     }, 2000)  
5. })  
6.  
7. doWorkPromise.then((result) => {
```

```
8.     console.log('Success', result)
9. } ).catch(error) => {
10.    console.log('Error', error)
11. }
```

To create a promise we use the new Promise( ) constructor function. As node developers we are not going to be the ones creating promises as they are typically created by the library we use. However, to understand how promises work it is essentially to understand the syntax. When we call a new Promise, we provide to it a single argument which is a function. We can use either a standard or an arrow function and this function gets called by the promise API and we get access to two arguments. The first is resolve and the second is reject.

Again we pass in a setTimeout function for 2 seconds and when it is done, we are ready to signify that we have completed the asynchronous process. Using the callback pattern we would have signified this by calling callback in one of two ways i.e. we would have provided a value in the first argument if things failed or provided a value in the second argument if things succeeded.

With promises, we have two separate functions which are resolve and reject. If things went well we would call resolve and if things went poorly and did not get the result we were expecting, we would call reject. We can see clear semantic when working with promises compared to callbacks because the order of arguments plays an important role for callbacks whereas the name of the functions clearly signifies that for promises. We can pass to resolve and reject functions a single value.

The doWorkPromise is the actual promise and a promise is nothing more than an object with a few methods that we can access.

The .then() allows us to register a function to run when things go well i.e. when resolve is called. We can setup that function and we get access to the property/data that the promise was resolved with via the first and only argument which we can call anything we want but logically is called result. Therefore, the .then() function only gets called when things go well.

We can chain the .catch() method onto the .then() method and this is going to allow us to register a function to run when reject is called. We pass in a function and get access to the error and we can go ahead and use it.

3.       `reject('This is an error!')`

There are clear advantages of working with Promises. Firstly, there are clearer semantics and it is easier to understand the intention of the code. With Promises we have two functions called for either success/failure cases which makes it easier to parse the code whereas with Callback patterns we only get a single callback function and we have to look at all calls to callback and then figure out which of the two arguments was provided which is more error prone and easier to run into issues with.

The second benefit has to do with the setup we have for doing something when an asynchronous task is

complete. With a callback pattern we have a single function and it is up to us as developers to determine whether or not an error occurred and run the correct code using conditional logic. When working with promises we have two separate functions and only one would ever run. With promises it is not up to us to add the conditional logic into place as we provide two separate functions making it easier for us to manage our asynchronous tasks.

The final advantage of working with promises is that it makes it easier to not mess up. This is because in promises we have to call either resolve or reject. We cannot call both and we cannot call one of them twice i.e. we cannot reject two times or resolve two times. Once the reject or resolve is called, the promise is done and its value or state cannot be change. Therefore, we cannot change the reject message by calling reject again and we cannot bring it from a failure to a success by calling resolve below. In contrast there is nothing built into the callback pattern to prevent this from happening i.e. there is nothing from stopping us running callback twice. Therefore, with promises there are rules enforced behind the scenes that makes it slightly easier to get the desired behaviour we are looking for and squash those asynchronous bugs.

### Terminology:

When we first create the promise, the promise is known as pending. So the promise above is pending for the two seconds before resolve or reject is called. Therefore, a promise is pending until either resolve or reject is executed. If resolve is called, the promise is considered fulfilled and if reject is called the promise is considered rejected.

## Updating Documents

The updateOne method allows us to update a single document. Below is an example of updating a single document but using the promise API:

```
const updatePromise = db.collection('users').updateOne({  
  _id: new ObjectId("5d892f50d070f10556f766eb")  
}, {  
  $set: { name: 'Nelson' }  
})
```

The updateOne takes in an object as its first parameter similar to the find method to filter the document we wish to update. In the above we target the document by its `_id` value. We use the new `ObjectId` constructor and pass in the id string value to target our documents accordingly. Now that we are filtering/targeting our documents correctly, we can add an object as the second argument to provide the updated values.

We use update operators to define the behaviour we want to perform. The most common operator is `$set` which allows us to set new values for the fields in our documents. The `$set` operator only impacts the fields we have explicitly listed out and updates those values only.

If we were using a callback pattern we would add a third argument to the `updateOne`, which would be the function and we have access to the error or the result. When we use promises, we do not provide that function and instead what `updateOne` returns is indeed the promise. We create a new variable that stores the `updateOne` method and we can call on the methods on this variable.

```
updatePromise.then( (result) => {
    console.log(result)
} ).catch( (error) => {
    console.log(error)
} )
```

So on the variable we can use the `.then()` method to register a function to run when things go well and the document is updated and then below we can chain the `.catch()` method to register a function to run when things fail. In the success case we get access to the result and in the fail case we get access to the error.

When running this code, we will see the name field updating but the age field will remain the same.

We can actually chain the `.then` and `.catch` methods without having to create a variable so the code would look something like:

```
db.collection('users').updateOne({
    _id: new ObjectId("5d892f50d070f10556f766eb")
}, {
    $set: { name: 'Nelson' }
}).then( (result) => { console.log(result) } ).catch( (error) => { console.log(error) } )
```

This is a very common pattern we will see where we add the `.then` and `.catch` onto the MongoDB method.

We can view all of the different update operators on the MongoDB documents page: <https://docs.mongodb.com/manual/reference/operator/update/>. The update operators all begin with a dollar sign. We can use update operators such as \$unset to remove a field, \$rename to rename a field, \$inc to increment a number by a specified amount and many more operators.

The updateMany() works similar to the updateOne(); however, we provide a filter document in the first argument that would capture documents that meet the criteria and update those cases. Using \_id would not make sense as this would narrow the search to a single document.

```
db.collection('tasks').updateMany( {  
    completed: false  
, {  
    $set: {  
        completed: true  
    }  
} ).then( (result) => {  
    console.log(result)  
} ).catch( (error) => {  
    console.log(error)  
} )
```

We now know how to create, read and update MongoDB documents.

## Deleting Documents

The final CRUD operation is to delete a document. Just like with the Create operation we can delete a single document or delete multiple documents at the same time. The two methods we can use are `deleteOne()` and `deleteMany()` and below is an example of both.

```
db.collection('users').deleteOne( {  
  _id: new ObjectId("5d892f50d070f10556f766eb")  
 } ).then( (result) => { console.log(result.deletedCount) } ).catch( (error) => { console.log(error) } )
```

```
db.collection('users').deleteMany( {  
  age: 28  
 } ).then( (result) => { console.log(result.deletedCount) } ).catch( (error) => { console.log(error) } )
```

We provide a filter as the first and only argument to delete which will filter by the criteria and delete the document that matches it. We can add a options argument as a second parameter but this can be ignored. We can either use a callback function as the third argument, but in the above we used the promise API instead which is provided to use when there is no callback function provided as the third argument.

We now know all four CRUD operations and how they work with MongoDB. We should now be comfortable using MongoDB and promises so that we can use these together with Express to build out a more complete and real world applications.

## Section 10

# REST APIs AND MONGOOSE

### Setting up Mongoose

We are going to explore a new npm module called mongoose. Mongoose is directly related to MongoDB and it is going to allow us to perform validation for our documents i.e. what fields are required or optional and what type of data are we expecting for them. We can also use mongoose to set a document to a given user and use authentication to prevent others from viewing that data. We can use mongoose to perform new things while still being able to perform our basic CRUD operations on the MongoDB database. We can learn more about mongoose from their website at <https://mongoosejs.com>.

Below is a 5 line example we can examine:

1. `const mongoose = require('mongoose');`
2. `mongoose.connect('mongodb://localhost:27017/test', {useNewUrlParser: true});`
3. `const Cat = mongoose.model('Cat', { name: String });`

```
4. const kitty = new Cat( { name: 'Zildjian' } );
5. kitty.save().then( () => console.log('meow') );
```

In this example, mongoose is connecting to the database and the next thing it does is it creates what's known as a model. A model allows us to model something in the real world that we want to be able to store in a database. This could be a user, task or in the above case a cat. We create models for all of the collections that we want and we use the model to describe the data. In the above, a Cat has a name and the name should be a string. A new instance of Cat is created and stored in the kitty variable which stores the new instance and this sets the name equal to Zildjian. Finally, various methods are used on that model to manipulate it i.e. .save() method is used to save the data which provides a much nicer interface for interacting with MongoDB instead of using all of the low level API methods from mongodb module.

Mongoose falls into a broader category of tools known as ODM (Object Document Mapper) and as the name suggests, the goal is to allow us to map our objects in our code over to documents inside of the MongoDB database. This is what Mongoose is going to allow us to do.

To install the mongoose library, within the terminal in our project directory we can run the following command:

```
:~task-manager$ npm install mongoose@5.7.1
```

Once we have installed mongoose we can start to play around and use the library to create our models.

### task-manager > src > db > mongoose.js:

```
1. const mongoose = require('mongoose')
2. mongoose.connect('mongodb://127.0.0.1:27017/task-manager-api',
3. { useNewUrlParser: true, useCreateIndex: true } )
```

First we need to load in our module into our application and we then use mongoose.connect to connect to our database. This is similar to the MongoClient.connect where we provide the URL as the first argument and also provide the options object as the second argument. Everything we learnt in the previous section mongodb module is relevant to mongoose. The only difference is that we provide the database collection name within the URL rather than separately. There is also another option we are going to configure which is useCreateIndex which will make sure that when mongoose works with MongoDB our indexes are created which allows us to quickly access the data we need to access.

Once we have mongoose setup, we can now use it to define our very first model. In the below example we create a user model:

### task-manager > src > db > mongoose.js:

```
1. ...
2. const User = mongoose.model('User', { name: { type: String }, age: { type: Number } } )
3. const me = new User( { name: 'Simon', age: 35 } )
4. me.save( )
5.     .then(( ) => { console.log(me) }
6.     .catch((error) => { console.log('Error!', error) } )
```

We use a variable for our model and always use a capital letter as the first character to resemble a model. We use mongoose.model() to generate a model. This method accepts two arguments, the first is the string name for our model and the second argument is the definition where we define all of the fields we want.

We setup each field and set it to an object and within the object we setup things about that field such as configuring validation, custom validation and the type for the fields we are working with.

We use the constructor functions from JavaScript to set the value type. Mongoose supports plenty of other types as well such as boolean, dates, arrays, binary data, ObjectIDs and more.

Now that we have our model defined, we can create instances of that model to actually add documents to the database. So if we want to create a new user, we would create a new variable to store that user and we set its value using the new operator followed by the const model name variable we created above. This is a constructor function for that model. We pass to the constructor function an object with all of the data for this particular user. We also ensure the field name matches to what we have setup in the model.

We now have an instance of our model created but at this point, nothing is actually getting saved to the database. To save to the database, we use methods on our instances and we have many methods we could use to do many different things. However, the .save() method allows us to save our model instance to the database. The save method does not take any arguments as it simply saves the data we stored in the me variable and this returns a promise which we can use to run after the saving process finishes.

In the `.then()` promise we get access to our model instance once again. So in the above this would be the `me` model instance which we can choose to name it, but since it is the same thing we can always access the existing `me` which we already have (i.e we can do either `.then((me) => {...})` or `.then(() => {...})`) which will work in both cases).

So to conclude, we are defining a model, creating a instance of that model and then saving that instance to the database. We can now run this script to see the document saved to the database:

```
task-manager$ node src/db/mongoose.js
{ _id: 5d8e8860cffccc034f4c4407, name: 'Simon', age: 35, __v: 0 }
```

We would see the `_id`, `name` and `age` showing up in the console when things went well, but we would also see an additional property added and managed by mongoose which is the `__v` property. This stores the version of the document and in the above we are starting at 0.

When we use mongoose we do get a base validation from the start so for example if we created a new instance of `User` and for the `age` added a string, we would see in the console a validation object describing what exactly went wrong as the error. We can customise the base level validation and we will learn this as we progress. Mongoose will not save invalid documents to the database.

## Creating a Mongoose Model

Below is an example code for creating a Mongoose model for a tasks collection:

task-manager > src > db > mongoose.js:

```
1. const mongoose = require('mongoose')
2. mongoose.connect('mongodb://127.0.0.1:27017/task-manager-api',
3. { useNewUrlParser: true, useCreateIndex: true } )
4.
5. const Task = mongoose.model('Task', { description: { type: String }, completed: { type: Boolean } })
6. const task = new Task( { description: 'Learn the mongoose library', completed: false } )
7. task.save( )
8.   .then(() => { console.log(task) }
9.   .catch((error) => { console.log('Error!', error) })
```

```
:~task-manager$ node src/db/mongoose.js
```

```
{ _id: 5d8e8d52daa2b803c3961873, description: 'Learn the mongoose library', completed: false, __v: 0 }
```

We will notice that when mongoose saves the model instance as a document, it takes the model name and uses it to create a new collection. However, mongoose converts the model name to a lower case plural name for example the Task model will be used to create a tasks collection within our database and it will save the model instance as a document in this tasks collection. Mongoose does this all automatically as part of the `.save()` method.

## Data Validation and Sanitisation

With data validation, we can enforce that the data conforms to some rule. For example, we can say that the user's age requires to be greater than or equal to 18 i.e. the user must be an adult.

Data sanitisation allows us to alter the data before saving it to the database. For example, removing empty spaces around the user's name.

We are going to explore data validation and sanitisation so that we know how to create a more fully featured models. Currently we have only learnt only the very bare bones validation using the type.

### Basic Validation:

By default we do not need to provide all of the fields when creating a new instance of a model. However, using the required validation we can enforce that certain fields have to be provided while others are optional. To customise the field options of our model, we just provide other properties on the object for the given field:

```
const User = mongoose.model( 'User', {  
    name: { type: String, required: true },  
    age: { type: Number }  
})
```

In the above we have added two validators for the name property i.e. type of string and field is required.

When we want to create a new instance of the User model, we have to require the name property but we can choose to leave the age off since the required option for age is not set to true.

The other validators we have available in mongoose are min and max validators for Numbers and enum, match, minlength and maxlength validators for Strings. In a real application, none of the aforementioned validators are really going to provide the sort of validation we would be looking for. For example, is the field actually storing a phone number, a valid email, credit card number, etc. Mongoose was not support to have all those built in, but it does provide us with a way to setup custom validation which is going to allow us to validate literally anything we would like.

To setup a custom validator for a field we can use the validate() function and as the first argument we get the value we are trying to validate.

```
const User = mongoose.model('User', {  
    name: { type: String, required: true },  
    age: { type: Number, validate(value) { if (value < 0) { throw new Error('Age must be a positive  
    number') } } }  
})
```

We use a logical if statement to throw an error if we have an invalid value. We throw a new Error and provide our error message. Therefore, while mongoose does not provide a tonne of built in validation, we can indeed customise it to fit our needs, in the above making sure the age is always a positive number.

When it comes to validating more complex things for example emails, phone numbers, social security numbers and others, it is typically best to use a well tested library that already handles all of that for us. There is a very popular npm library for this called validator (<https://www.npmjs.com/package/validator>).

We can install the package using the following command within our project directory:

```
:~task-manager$ npm install validator@11.1.0
```

We can use this npm library for more complex validation. Below is an example of validating a email:

```
const validator = require('validator')
const User = mongoose.model( 'User', {
  name: {...}, age: {...}
  email: { type: String, validate(value) { if (!validator.isEmail(value)) { throw new Error('Email is invalid') } }
})
```

In the above we use the validator object to run a method called `isEmail` which takes in a string and returns true or false if the string passed in is an email. We check using the if statement to see if the value passed in is not a valid email using the logical not ! operator and if this returns true then the email is invalid and we throw a new Error message.

Using the custom validator is going to allow us to define our own logic or we are able to use other libraries that have predefined logic to validate things that a little more complex.

The theme of this guide is to write code specific to our app and use npm modules when we are doing something generic that other applications need to do as well.

We can use choose to set a default value for a field should no value be provided. We set default equal to the default value to use.

If we look at the mongoose documentation at <https://mongoosejs.com/docs/guide.html> and head over to the schemas types section we can see all of the types of data we can work with along with some of the validators we can use such as required, default, select, validate, etc. which we have explored.

If we continue to scroll down below we can view different options available to the specific data types such as for Strings we have options for lowercase, uppercase, trim, etc. and these are ways we can sanitise our data. This allows us to manipulate the string before we save the document to the database collection.

This page gives a really nice rundown of all of the things we can use.

```
name: { type: String, required: true, trim: true },  
age: { type: Number, default: 0, validate(value) { ... } }
```

This provides us more control over the data we are allowing into the database. This will ensure the data we save is consistent and valid. Therefore, we can use mongoose to easily add validation and sanitisation to our application document models which are two essential topics when it comes to accepting input from the users.

## Structuring a REST API

The REST API stands for Representational State Transfer Application Programming Interface (a.k.a REST API or RESTful API).

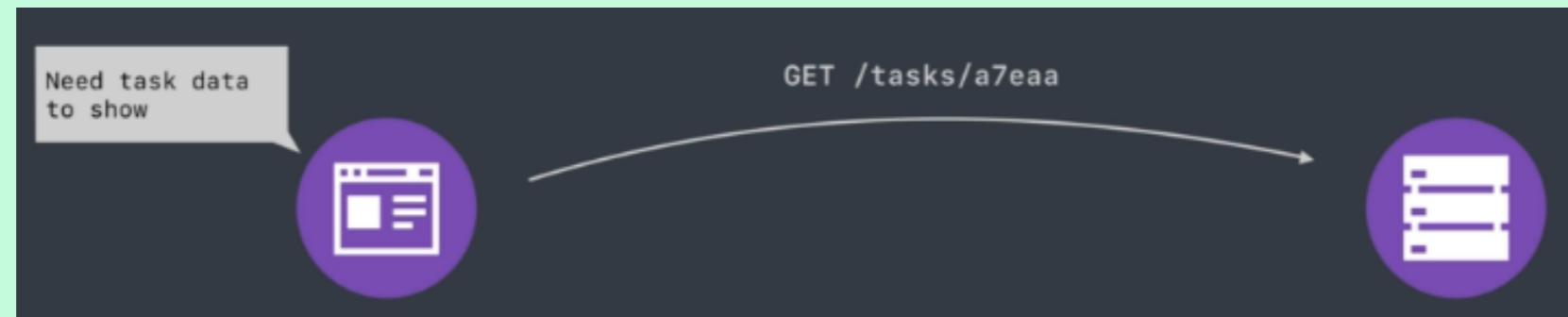
An API is nothing more than a set of tools that allow us to build software applications. It is a very broad term. Therefore, we can say Node provides us with an APIs and we can also say our NPM Modules also provides us with APIs for example Express provides us with tools that allow us to build software applications. The REST API we are creating is also going to provide a set of tools allowing others to build out their software.

The REST API allows clients such as a web application to access and manipulate resources using a set of pre-defined operations. What is a resource? This is something like a users or a tasks. What is a pre-defined operation? This is a pre-defined operation for users and tasks something like the ability to create a new task or to mark a task as complete or to do something more advanced like uploading a profile picture for a user account. These pre-defined operations are going to allow a client like a web app to go through the process of creating a front-end for a task manager.

Representation with a REST API, we are getting and working with representations of our data. The data is stored in the database but using the REST API, we can still fetch, manipulate, and perform CRUD operations with the data. Therefore, we are working with representation of users and tasks data.

When it comes to State Transfer, the REST API/server is stateless. The State has been Transferred from the server to the client. Therefore, each request from the client such as a request from a web application, contains everything needed for the server to actually process that request. This will include the operation the user is trying to perform, all of the data the operation actually needs in order to work and it also includes things such as authentication to make sure that the user who is trying to perform the operation is actually able to do so.

This will make more sense when put into practice. In practice the requests are going to be made via HTTP request and so this is how a client like a web app is going to be able to perform those pre-defined operations.



In the above diagram, the client requires data to show on the page so it is going to make a HTTP request to a specific URL on the server. In the above, the client wen app is using the GET HTTP method to make a request to /task/a7eaa where the a7eaa is the id of the task it is trying to fetch. The server is going to go through the process of fulfilling the request. It is going to find the data in the database and it is going to send it back as part of the HTTP response.



In the above we have a status code indicating everything went well and we have the JSON response with the data requested. There are other status codes such as 404 for page not found and we will explore the complete list of status codes available as we learn how to build out our own API. Once the data is sent to the client, the client is ready to render things.



With a REST API, we can use more than just the GET HTTP request method to request data, we are going to be creating, deleting and updating data. We still have a client and a server and we are still making HTTP request but we would use different methods such as POST, UPDATE and DELETE request methods.

POST is used for creating data, so in the below we can POST to /tasks and sending along the JSON data

with the request to create a new task. When the servers gets the request, it is going to make sure to authenticate the account exists and then it is going to go create the new task. Once the task has been created, we will get the response back.



Here we would see a different HTTP status code of 201 which signifies that a resource was created and we are also getting a JSON response which is the new task that has been created. The client will get the response and will be able to use it to signify to the user in the user interface that things went well and that the task was created. We will explore the various HTTP methods and status codes available to us in later sections.

In order for anyone to be able to do anything meaningful with our API, we need to expose the necessary set of pre-defined operations for things such as the CRUD operations. Below is an example of pre-defined CRUD operations exposed for the Task Resource:



Every REST API operation is defined with two pieces of data which are the HTTP method and the path. So in the above example we are using the HTTP POST method to /tasks path which is setup for creating a resource. We would typically have two read operations, one for fetching a individual resource and another for fetching all of the resource using the GET method. The :id is a placeholder in the path which would get replaced with some value for example the task id we are trying to fetch.

The above example is the most common REST API structure we will see and use when building out a REST API for our applications.

Finally, a HTTP request made back and forth between the client and server is just simply some text. The structure of a HTTP request is text based as seen in the below example:



There are three main pieces to a HTTP request text. The first line is known as the request line. This contains the HTTP method being used, the path and the HTTP protocol.

After the request line, we have as many request headers as we need. In the above example we have three which are Accept, Connection and Authorisation. Headers are no more than key:value pairs which allow us to attach meta information to the request. So in the above we are using the Accept to say that we are expecting JSON data back, we are using Connection to say that we are likely to make other requests shortly and so keep this connection open to keep things fast and finally we are setting Authorisation to setup authentication. After we are done with the headers we have an empty line followed by the request body. So when we POST data to /tasks, we have to provide that data and we provide that data as JSON inside of the request body.

Once the server gets this request, it is going to be able to parse it and Express does great work for us by giving us access to all of the request in a much easier interface and it sends back a response which looks quite similar to the request.



The first line contains the protocol followed by the status code followed by the text representation of the status code. On the second line we have the response headers and can have as many as we want. In the above we have three headers which are the Date (represent the time it happened), Server (represents the server e.g. Express) and Content-Type (represents the meta-data of the response body). Again we have an empty space followed by the response body which in the above case is the complete task.

This is the basic theory of what makes up a HTTP request and response and we can take this and put this into practice and build our own REST APIs.

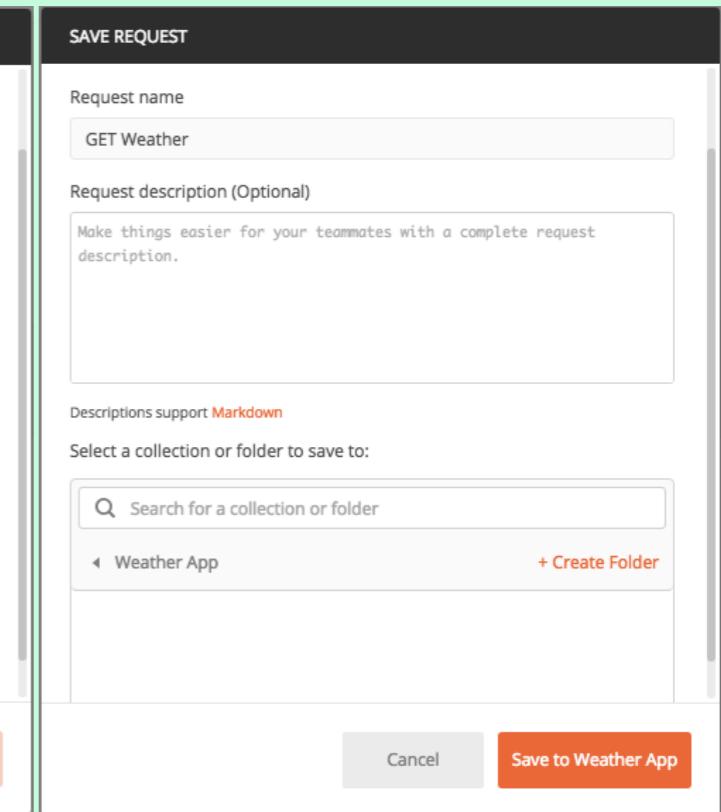
## Installing Postman

Postman is an industry standard tool that makes it easier to fire off all of the HTTP request we need, allowing us to test and verify our REST API ensuring it is working as expected. To install Postman, we can go to <https://www.getpostman.com> and install the tool. This is available for all operating system and is indeed free. There are paid plans but these are features we would not need for example data synchronisation if working on an API with multiple developers which may be useful later on. We would download and install the application on our machine and we can fire off a HTTP request using Postman to make sure that it is actually working.

The goal of Postman is not to replace a client, the goal is to allow us to test our REST API without having to also create a client to test it with. This is going to allow us to automatically test things such as signing up a user with valid data and invalid data to make sure we get the correct response.

Once downloaded we can open up the application, it may ask to signup to their web services but there is no need for this and we can skip the signup process. We would then be brought to the postman tool. We can click on the Request from the Create New Tab to create a new basic Get Request.

This will bring up a form where we would add a request name, add a optional request description and provide/create a collection to save all of our API request within a collection folder.



This screenshot shows the main Postman interface. The top navigation bar includes 'New', 'Import', 'Runner', 'Collections', 'APIs BETA', 'My Workspace', 'Invite', and various status indicators. The left sidebar shows 'History' (selected), 'Collections', and 'APIs BETA'. A message says 'You haven't sent any requests' and 'Any request you send in this workspace will appear here.' A 'Show me how' button is present. The main workspace displays a collection named 'GET Weather' with a single GET request. The request details show 'Method: GET', 'URL: Enter request URL', and tabs for 'Params', 'Authorization', 'Headers', 'Body', 'Pre-request Script', 'Tests', 'Settings', 'Cookies', and 'Code'. The 'Params' tab is active, showing a table for 'Query Params' with one row: 'Key' and 'Value'. The 'Response' section is below the request details.

Once we create the collection, this will bring us to the Postman interface to create the basic Get Request.

We have the history tab which displays the requests we have made in the past and we also have a collections tab to view all of the different collection folders storing our requests. We can click on the request to bring it up on the main menu to either configure or fire off the request.

We have two main information, the HTTP method we would want to use i.e. GET and we have the URL path we are trying to make the request to. Below we have different ways of configuring the query string, authorisation, headers, request body and more.

If we use the GET method and use a path to test with, we will notice when we add queries in the URL string they are automatically added to the Params. Alternatively, we can add the params key:value pairs and let Postman do the heavy lifting of updating the URL string above.

The screenshot shows the Postman interface with the following details:

- Request Type:** GET
- URL:** https://mead-weather-application.herokuapp.com/weather?address=London
- Params Tab:** Address is set to London.
- Body Tab:** Shows a JSON response with forecast, location, and address fields.
- Status Bar:** Status: 200 OK, Time: 1177ms, Size: 508 B.

```
1 "forecast": "Light rain until morning, starting again in the evening. It is currently 66.97 degrees out. This high
  today is 67.27 with a low of 57.57. There is a 0% chance of rain.",
2 "location": "London, Greater London, England, United Kingdom",
3 "address": "London"
```

We can save this using the save button and then use the send button to send and fire off the request. This will communicate with the server and get the weather response. We can view the pretty/raw JSON response. This would mean Postman is installed and working correctly. There are many features to Postman we can use while testing our APIs.

## Resource Creation Endpoints

We are going to explore how to create our very first endpoint for our REST API. The goal is to focus on our endpoints that involve resource creation for example creating a new users and new tasks.

Firstly, we will need to install nodemon as a dev dependency and install Express as a regular dependency by running the following commands within our project directory:

```
:~task-manager$ npm install nodemon@1.19.2 –save-dev  
:~task-manager$ npm install express@4.17.1
```

Once we have these tools installed, we can start using them within our projects. The first thing we would need to do is create a index.js file which is going to be the starting point for our application and also where we are going to initialise the Express server.

task-manager > src > index.js:

1. const express = require('express')
2. const app = express( )
3. const port = process.env.PORT || 3000
4. app.listen(port, ( ) => { console.log('Server is up on port ' + port) } )

We now have the basic server in place and would now need to setup our package.json scripts to make sure that we can easily use nodemon once it is installed as a dev dependency locally.

### task-manager > package.json:

1. ...
2. "Scripts": { "start": "node src/index.js", "dev": "nodemon src/index.js" }

We now have two scripts that we can use to start our application. The start script would be used by the server if deployed on a web hosting platform such as heroku, while the dev script will allow us to run our server on our localhost on port 3000. There is no need for the handlebar extensions to watch for changes to other .hbs template files as we are not using handlebars. Now that we have this in place, we can start up our server and create our very first route.

```
:~task-manager$ npm run dev
```

We should see the message in the terminal to show that our Express server is running on port 3000 but if we were to visit this in the browser, we would not see anything because we have not configured the express application to actually do anything. If we go back into our index.js file we can start to add more code before the app.listen() method code line.

### task-manager > src > index.js:

1. const express = require('express')
2. ...
3. app.post('/users', (req, res) => { res.send('testing!') })
4. app.listen(port, () => { console.log('Server is up on port ' + port) })

Express provides us with methods for all of the HTTP methods we can use. So we have HTTP methods on our app object such as `app.get()`, `app.post()`, `app.patch()` and `app.delete()` to perform the CRUD operations.

All these methods have the same call signature. The first argument is the path and the second is our callback function which runs when someone tries to access that particular route. We get access to the request and response arguments within our callback function. We can use the response object and send back a test response text.

If we were to now save this code, we can now go into Postman to test this response. We can create a brand new collection for the application e.g. Task App to save all our test API requests for this project. We can add a new request and configure it to test the POST request with our Express server. When running the request we should see the response of testing. Below Postman example:

The screenshot shows the Postman application interface. On the left, there's a sidebar with a search bar, tabs for 'History', 'Collections' (which is selected), and 'APIs BETA'. Below these are buttons for '+ New Collection' and 'Trash'. Under 'Collections', there are two entries: 'Task App' (1 request) and 'Weather App' (1 request). The main workspace shows a 'POST Create user' request. The 'Method' dropdown is set to 'POST' and the 'URL' field contains 'localhost:3000/users'. To the right of the URL are 'Send' and 'Save' buttons. Below the URL, there are tabs for 'Params', 'Authorization', 'Headers (8)', 'Body', 'Pre-request Script', 'Tests', 'Settings', 'Cookies', and 'Code'. The 'Body' tab is selected. Under 'Body', there's a 'Query Params' section with a table:

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

At the bottom of the workspace, there are tabs for 'Body', 'Cookies', 'Headers (6)', and 'Test Results'. The 'Test Results' tab is selected. It shows a status message: 'Status: 200 OK Time: 1198ms Size: 211 B Save Response'. Below this, there are buttons for 'Pretty', 'Raw', 'Preview', 'Visualize BETA', 'HTML', and a search bar. The preview area shows the response body: '1 testing!'

At this point we are able to run the given route handler. The question is how do we provide the data necessary from the client such as a the name, email and password to send to the server. As we previously learnt we would send the JSON data as part of the request body. Postman provides us with a way to configure the request body via the Body tab.

By default none is selected. We would switch this from none to raw and then from the dropdown we would select from Text to JSON which will allow us to send JSON data from postman over to the express server.

The screenshot shows the Postman interface for a POST request to 'localhost:3000/users'. The 'Body' tab is active, showing the option 'none' selected. Below it, there are other options: 'form-data', 'x-www-form-urlencoded', 'raw', 'binary', and 'GraphQL BETA'. A note says 'This request does not have a body'. The status bar at the bottom indicates a successful response: 'Status: 200 OK'.

The screenshot shows the Postman interface for a POST request to 'localhost:3000/users'. The 'Body' tab is active, showing 'JSON' selected. The body field contains the following JSON object:

```
1 {  
2   "name": "John Doe",  
3   "email": "johndoe@exampleEmail.com",  
4   "Password": "Example123!$"  
5 }
```

The status bar at the bottom indicates a successful response: 'Status: 200 OK'.

This is where we are going to provide everything we need to create a new user ie. name, email and password fields. We do not need to provide age as the age is optional. If we send this off, we would still see our request being sent off with testing being returned as the response and a status code of 200 meaning everything went OK.

The question now becomes how do we get this data to create a new user? This is a two step process and the first is to configure Express to automatically parse the incoming JSON for us so that we have it accessible as an object we could easily use.

task-manager > src > index.js:

```
1. const express = require('express')
2. ...
3. app.use(express.json())
4. app.post('/users', (req, res) => {
5.   console.log(req.body)
6.   res.send('testing!')
7. })
8. app.listen(port, () => { console.log('Server is up on port ' + port) })
```

We set this up by using `app.use()` to customise our server and express provides us something to pass in by using `express.json()` calling it as a function. This one line is going to automatically parse incoming JSON to an object so that we can access it in our request handlers. We can do this using `req.body()`. If we save the code and nodemon restarts the server, we can run Postman again with the POST request to view the `req.body` object printed to the console to view what we get back from a HTTP request. We should now see the parsed JSON data in a object within the terminal exactly how they were provided in Postman. We now know how to grab the incoming request body data and we can now use it to actually create a new user. To get this done we need to ensure mongoose has connection to the database and we need to get access to our User model from inside of the `index.js` file.

We would need to do some restructuring as to what lives inside of our mongoose.js file. This file is going to only have the necessary code to connect to the database. Our models will be located in other files of which we are going to create a new directory within the src folder called models which will hold all our models and this director is going to sit alongside of the db directory that contains the mongoose.js file.

Each model will sit in its own file within the models directory. The directory structure will look something like the below:

task-manager > src > db > mongoose.js:

```
1. const mongoose = require('mongoose')
2. mongoose.connect('mongodb://127.0.0.1:27017/task-manager-api', {
3.   useNewUrlParser: true, useUnifiedTopology: true, useCreateIndex: true
4. })
```

task-manager > src > models > user.js:

```
1. const mongoose = require('mongoose')
2. const validator = require('validator')
3. const User = mongoose.model('User', { name: {...}, email: {...}, password: {...}, age: {...} })
4. module.exports = User
```

We use module.export to export the User variable so that other files such as index.js can use it to create new users.

We can now go back into the index.js file and load in a couple of things:

task-manager > src > index.js:

```
1. const express = require('express')
2. require('./db/mongoose')
3. const User = require('./models/user')
4. ...
5. app.use(express.json( ))
6. app.post('/users', (req, res) => {
7.     const user = new User(req.body)
8.     user.save( ).then((user) => { res.send(user) }).catch((error) => { res.status(400).send(error) })
9. })
10. app.listen(port, () => { console.log('Server is up on port ' + port) })
```

We do not want to actually grab anything from the mongoose.js file and so by simply calling require, this is going to ensure that the file runs and it is going to ensure that mongoose can connect to the database.

The next thing we would do is load the User model into our index.js file. With this in place we can now create a new instance of the user inside of our route handler and we can make sure that they get saved correctly. The new User() constructor function requires a object which we know lives within req.body which we can used to pass is an the argument. We can use this instance and call .save() method on the object to save the user to the database. We use the .then() and .catch() promises to handle the success and errors of the operation.

We can now save the changes made to the code and run the API request again using Postman; however, we must ensure the request body data in Postman is setup correctly per the User model validations and we must have our database server running in order to save to the database without throwing any errors.

In Postman, when running the post request we should see the newly created user object returned as the success response and If something failed we should see the error response.

In the success case response we should see the user object including all of the fields we passed into the request body including the additional fields that MongoDB adds automatically to a document i.e. ObjectId. We can also head over to the Robo 3T application to confirm that our HTTP POST request has actually created a new user within our user collection of our database.

In the error case not only do we want to send the response object for the error but we also want to change the status code. When working with REST APIs we want to send back the most accurate status code as possible. For a list of all of the status codes we can work with we can find them on <https://httpstatuses.com/>. We add the status code on the response using res.status() method. It is important we do this before we call the res.send() method, otherwise the status is not going to get sent correctly. We can chain both commands together e.g. res.status(400).send(error).

This is our very first REST API which we are able to send data via a HTTP request off to the express server and using a pre-defined operation we are able to perform some form of manipulation on the database, in this case creating a new user.

## Resource Reading Endpoints

Below is an example code of using mongoose to create a Reading endpoint whereby we can fetch the data from the database. Reading data will have two end points, one to fetch all data in the database collection and another to fetch a single document.

task-manager > src > index.js:

```
1. const express = require('express')
2. require('./db/mongoose')
3. const User = require('./models/user')
4. ...
5. app.get('/users', (req, res) => {
6.   User.find({ }).then((users) => { res.send(users) } ).catch((error) => { res.status(500).send( ) } )
7. }
8. app.listen(port, () => { console.log('Server is up on port ' + port) } )
```

The .find() method is used to return all documents and it uses a query object just as we saw with the native driver. We use an empty object as the query object to retrieve all data from the database collection and we have access to the promise .find() method returns. We do not necessarily need to send an error back as the status code could be more than enough to let the user know that the service is currently down.

The request for fetching a single document looks similar to the above but with a slight difference. We require the query object to filter the request and we can choose the id as the filter to narrow the search.

### task-manager > src > index.js:

```
1. const express = require('express')
2. require('./db/mongoose')
3. const User = require('./models/user')
4. ...
5. app.get('/users/:id', (req, res) => {
6.     const _id = req.params.id
7.     User.findById(_id)
8.         .then((users) => { res.send(users) })
9.         .catch((error) => { res.status(500).send() })
10. })
11. app.listen(port, () => { console.log('Server is up on port ' + port) })
```

The path is the same as the above; however, we add /:id to make the URL dynamic. The id is going to change as we make requests to this end point. We need to capture whatever value is put after the second forward slash and get access to it in the route handler so we can fetch the user correctly.

Express gives us access to what are known as route parameters. These are parts of the URL that are used to capture dynamic values. We use a colon : followed by a name which we can choose whatever we like (in the above we used id). Essentially what we are saying, a user is going to make a get request to /users/ and then something. When that happens we want to grab this route handler to run and we want to be able to access the dynamic value provided inside of the URL.

The request gives us access to `req.params` which contains all of the route parameters that were provided and in this case it is an object with a single property of `id` and the value for `id` is whatever was put in the URL. We can use the `req.params` object targeting the `id` property and store it within a variable.

Finally, we can use either the `.findOne()` or `.findById()` methods provided by mongoose to find a single user document that matches the `id` passed in the URL parameter. Note with the `.findOne()` we use the query object to filter by our search criteria, whereas the `.findById()` method takes in a single string of the `id`. We can then provide our promises call to either return the single user document or show an error status.

We use a little bit of conditional logic for the scenario whereby the results finds no matching results as this is considered a success by mongoose when using the `.findOne()` method. The `findById` will error if an incorrect `id` string value is passed in triggering the `.catch()` promise.

We now know how to create two end points for reading data from the database.

## Promise Chaining

Promise chaining allows us to work with multiple asynchronous operations where one needs to happen and then another. Promises provides an advanced syntax for actually getting that done. Below is a demonstration of how we can use promise chaining to perform multiple asynchronous operations.

## playground > app.js:

```
1. const add = (a, b) => {  
2.     return new Promise( (resolve, reject) => {  
3.         setTimeout(() => { resolve(a +b) }, 2000)  
4.     } )  
5. }  
6. add(1, 2).then((sum) => { console.log(sum) } ).catch((error) => { console.log(error) } )
```

The add function will take in two parameters of a and b and all it will do is return a promise. After two seconds that promise will be fulfilled with the sum. Now that we have this in place, we know how we can go through the process of using this a single time by calling add and passing in 2 numbers. We can use the .then() and .catch() method calls to do something when that promise resolves. The above is what we have done before when learning promises. From here we will be learning the new concept of promise chaining.

```
:~playground$ node app.js
```

```
3
```

When running the programme we would see the sum printed after two seconds. This is one asynchronous function running. We are now going to build on this asynchronous function by taking that sum and use it with add again to add another number into the mix. Therefore, this would be two calls to add() or two asynchronous operations.

If we did not know about promise chaining, we could have solved the problem by nesting the two asynchronous calls like so:

playground > app.js:

```
1. ...
2. add(1, 2)
3.   .then((sum) => {
4.     console.log(sum)
5.     add(sum, 3)
6.       .then((sum2) => { console.log(sum2) })
7.       .catch((error) => { console.log(error) })
8.   })
9. .catch((error) => { console.log(error) })
```

```
:~playground$ node app.js
```

```
3
6
```

This will work and we would see after two seconds 3 printing and then after another two seconds 6 printing to the terminal. The problem with the above solution is similar to the problem we had when using the Callback Pattern without using promises. The more asynchronous tasks we try to perform the more nested and complex our code gets (so far we are nested two levels deep). We also have duplicate code for catching errors which is not ideal either. The better way to get this done is using Promise Chaining.

If we did not know about promise chaining, we could have solved the problem by nesting the two asynchronous calls like so:

playground > app.js:

```
1. ...
2. add(1, 2)
3.   .then((sum) => {
4.     console.log(sum)
5.     return add(sum, 3)
6.   })
7.   .then((sum2) => { console.log() })
8.   .catch((error) => { console.log(error) })
```

:~playground\$ node app.js

```
3
6
```

When using promise chaining, what we do is we return the next promise from our then callback. So we need to return a promise. The question then becomes where do we put our other .then() call? In the previous example we nested the .then inside each other where each asynchronous function gets us further and further nested as we moved on. With promise chaining the second then call comes after the first. We get access to the result of that promise and we can use it in the second call to the .then() callback. Finally, we have a single .catch() promise.

This is promise chaining in effect. It is chaining because what we are doing is chaining together multiple .then() calls and each working with a different promise. The first .then() call runs when the first promise is fulfilled and the second .then() call runs when the return promise us fulfilled and we can add on a .catch() at the end to catch those errors and therefore we have the .catch() call called a single time.

The nice thing about promise chaining is that our code is now not nested and we can call 10 calls to add() promise function and it will never be nested. With promise chaining, we are able to achieve something similar to the previous example but with an improved syntax. All we need to know is that we can return a promise from one of our .then() callbacks allowing us to chain another .then() call on.

## Async/Await

Async/Await makes it even more easier to work with our promise based code by writing code that looks more synchronous than asynchronous. Async/Await is not a whole new thing, it is simply a small set of tools that makes it easy to work with promises. So we are going to continue to use the same old promise methods we have used before so nothing will change there, the only thing that will change is how we manage our code when we have a lot of asynchronous things going on.

Below is an example code of using the Async/Await syntax to demonstrate the new ES6 syntax.

playground > app.js:

1. const doWork = () => { }
2. console.log(doWork( ))

The doWork function is an empty arrow function that does nothing. Below we are going to call this doWork function returning its value to the terminal. We know in JavaScript that if we do not explicitly return something from a function, undefined is implicitly returned. So if we were to run the above code we would see undefined in the console.

```
:$ playground$ node app.js  
undefined
```

What we are about to do is use `async/await`. The first part of the feature is `Async`. This allows us to create a `Async` function and in that function we can use the `Await` feature. This will make more sense when we have a more complete example.

We first need to mark a function as a `Async` function and this is done by asking the `Async` keyword right before the function declaration.

playground > app.js:

1. const doWork = async () => { }
2. console.log(doWork( ))

With this simple change, we are actually changing the behaviour of our programme. We are no longer going to get undefined printing in the terminal.

```
:~playground$ node app.js  
Promise { undefined }
```

If we now run the script, we no longer get an undefined, instead we see doWork is returning a promise and that promise has been fulfilled with the value of undefined. This is the first important thing to note about async functions i.e. async functions always return a promise and that promise is fulfilled with the value us as the developer choose to return from the function. So below if we explicitly return something we would see the promise return our return value:

```
playground > app.js:  
1. const doWork = async () => { return 'Andy' }  
2. console.log(doWork( ))
```

```
:~playground$ node app.js  
Promise { 'Andy' }
```

So the above async function returns not a string value but instead a promise that gets fulfilled with the string value. Since a promise is being returned, we can use the .then() and .catch i.e. our promise methods instead of just taking the promise and dumping it to the terminal.

playground > app.js:

1. const doWork = async () => { return 'Andy' }
2. doWork().then((result) => { console.log('result', result) }).catch((e) => { console.log('error', e) })

We can use the .then() method to get access to the result. Here the result is whatever was returned in the doWork async function. In the above, because the return value gets resolved we expect to see the .then() callback called and is exactly what we get.

```
:~playground$ node app.js  
result Andy
```

To get the .catch to run, if we throw an error from our async function that is going to be the same as rejecting the promise that comes back from the async function. So when we return a value, we are fulfilling the promise with a value and when we throw an error we are rejecting the promise with that error.

playground > app.js:

1. const doWork = async () => { throw new Error('Something went wrong') }
2. doWork().then((result) => { console.log('result', result) }).catch((e) => { console.log('error', e) })

```
:~playground$ node app.js  
e Error: Something went wrong ...
```

This is the basic structure of an asynchronous function and the question is why is this useful?

The other half Async/Await which is the await operator. The await operator can only be used in async functions. The whole point of asycn/await is to make it easier to work with asynchronous promise based code.

When working with Async/Await, we do not have to change how our promises function internally. All we need to do is change how we work with them. As a consumer of the promises we can use async/await if we wanted to.

playground > app.js:

```
1. const add = (a, b) => { return new Promise((resolve, reject) => { setTimeout(() => { resolve(a + b), 2000) }) } }
2. const doWork = async () => {
3.   const sum = await add(1, 2)
4.   return sum
5. }
6. doWork().then((result) => { console.log('result', result) }).catch((e) => { console.log('error', e) })
```

With async/await we are going to remove all those callback functions when calling the add() functions a couple of times. What we get access to the async is the await operator. The await operator gets used with a promise. We can get a promise because the add() function returns a promise. Now the sum is 3, but the question is where do we get access to this data? If we were not using async/await we would have used a .then() call, provide our callback function and we get access to the data inside of there.

With `async/await`, it looks like `add()` is a standard synchronous function and we can actually create a variable to get access to the value that the promise is fulfilled with. We still have to wait those two seconds the advantage is syntactical. It is a lot easier to reason about and parse this code than it is with the code we had in the promise chaining. We can return the sum.

If we now run the code we would see our result of 3 after the two seconds:

```
:~playground$ node app.js  
result 3
```

We can take this further to show the benefits of `async/await` with the example below of adding more numbers:

```
playground > app.js:  
1. const add = (a, b) => { return new Promise((resolve, reject) => { setTimeout(() => { resolve(a + b) },  
2000) }) }  
2. const doWork = async () => {  
3.   const sum = await add(1, 2)  
4.   const sum2 = await add(sum, 3)  
5.   const sum3 = await add(sum2, 4)  
6.   return sum3  
7. }  
8. doWork().then((result) => { console.log('result', result) }).catch((e) => { console.log('error', e) })
```

```
:[~playground$ node app.js  
result 10
```

We have three asynchronous operations all taking two seconds each to run and therefore after 6 seconds we should see the value 10 printing to the terminal. All of our code runs in order, even though asynchronous things are indeed happening behind the scenes. We have code here much simpler than the code in the promise chaining code we saw in the previous section. Async/Await was created to fight the confusion that could occur in promise chaining as it is much easier to see what is going on once we understand what `async` and `await` actually do.

Another problem with promise chaining is to have all of the values in the same scope. So if in the first function we have access to the first sum and in the second function we have access to the second sum, but what if we wanted to have access to both of those sum values at the same time to do something like saving them to a database or send them to the user. There is no easy way to do that. What we would have to do is create variables in the parent scope and re-assign them in the promise chaining scope and it turns into a mess really quickly. With `async/await` there is no need to worry about any of that. We have access to all of the individual results in the exact same scope in our `async` function, allowing for a lot of flexibility.

Finally, we need to discuss what happens if one of the `async` functions rejects instead of it fulfilling. So in the below we can only add numbers if they are positive and if not we would throw a reject.

## playground > app.js:

```
1. const add = (a, b) => { return new Promise((resolve, reject) => { setTimeout(() => {  
2.     if(a < 0 || b < 0){ return reject('Numbers must be non-negative') }  
3.     resolve(a + b) }, 2000) } )  
4. }
```

The code will continue to work in its current state as it did before; however, if we make one of our numbers negative then we can see what happens. With `async/await` things occur one at a time so the code will run the first line waiting two seconds and everything is fine, we wait another two seconds and everything on the second line is fine. We finally wait another two seconds on the third line and it is here where we get that error and so after 6 seconds we are going to see that a problem occurred.

```
5. const doWork = async () => {  
6.     const sum = await add(1, 2)  
7.     const sum2 = await add(sum, 3)  
8.     const sum3 = await add(sum2, -4)  
9.     return sum3  
10. }  
11. doWork().then((result) => { console.log('result', result) } ).catch((e) => { console.log('error', e) } )
```

```
:~playground$ node app.js  
e Numbers must be non-negative
```

If we added the negative sooner i.e. on the first line, we would have seen the error message sooner. If the code rejects, then none of the following code below is going to run and it is going to skip right to the .catch() call printing the error message.

To conclude, we have explored async/await in much depth and this is very crucial to understand as we continue to use it further in developing our codes and will play a very critical role in performing asynchronous functions.

Below is a example of refactoring our Express code within the index.js file:

task-manager > src > index.js:

```
1. const express = require('express')
2. ...
3. app.post('/users', async (req, res) => {
4.     const user = new User(req.body)
5.     try {
6.         await user.save()
7.         res.status(201).send(user)
8.     } catch (error) {
9.         res.status(400).send(error)
10.    }
11. })
```

First, we turn the function we pass into Express into a async function to use await. When we add that async definition it changes the behaviour of the function. The functions goes from returning whatever value you returned to always return a Promise. The good news is that Express does not use the return value from the promise function. Express does not care what we return, instead we use request and response to tell express what we want to do, so in this case we added the async functionality without changing the behaviour at all. We would now refactor the code to use await.

We await the promise that comes back from calling that .save() method. We can now use the await keyword as we are now in an async function. At this point we have saved the user and everything that comes after is only going to run once the user was saved either successfully or un-successfully. So we know the code below await user.save() will only run if the promise of the above is fulfilled. If it is rejected, the rest of the function is going to stop. We can actually handle individual errors from individual promises using the standard try catch statement as we would in regular JavaScript code. So we are going to try to run the await user.save() promise to fulfil and if anything in the try block throws an error, we will use the catch block to catch the error. We can do one thing or another using the try catch statement. The code below the user.save() will only run if the promise fulfils and if it throws an error the code in catch will run so essentially we can do one of two things using the response to send a response back to the client.

We now have a refactored async/await code for our POST request for creating a new user. We can refactor all of our Express code in the index.js file to take advantage of the async/await syntax.

## Resource Updating Endpoints

Below is an example for a HTTP endpoint for updating resources using the async/await syntax.

task-manager > src > index.js:

```
1. const express = require('express')
2. require('./db/mongoose')
3. const User = require('./models/user')
4. ...
5. app.patch('/users/:id', async (req, res) => {
6.     const updates = Object.keys(req.body)
7.     const allowedUpdates = ['name', 'email', 'password', 'age']
8.     const isValidOperation = updates.every((update) => allowedUpdates.includes(update) )
9.     if (!isValidOperation) { return res.status(400).send( { error: 'Invalid updates!' } ) }
10.    try {
11.        const user = await User.findByIdAndUpdate(req.params.id, { req.body }, { new: true,
12.            runValidator: true } )
13.        if (!user) { return res.status(404).send() }
14.        res.send(user)
15.    } catch (error) {
16.        res.status(400).send(error)
17.    }
18. app.listen(port, () => { console.log('Server is up on port ' + port) })
```

The `app.patch()` method is designed for updating an existing resource. Again we set the path for the endpoint and whereby we want to update an individual data by its id. We then setup our `async` function. Mongoose has many different update methods we can use and all can be found on the documentation. The above uses the `findByIdAndUpdate()` method. This method takes the filter of id as the first parameter and the second parameter is the object for various fields we want to update. Unlike the `mongodb` native driver, `mongoose` does not require us to use the `$set()` operator as it handles this for us in the background.

We want to take the update in as the request body, so much like for creating a user, the data will be passed through via the HTTP request and so we access the `req.body` to grab this information.

The third argument is an option object to help get things working the way we want them. The first option is `new` which we set to `true`. This is going to return the new user as opposed to the existing one that was found before the update and so we get back the latest object. The second option is `runValidators` which is going to make sure that we do run validation for the update.

We now have a connection setup for a resource updating endpoint that can be tested using Postman tool. It is important to note, if we try to update by using a field we do not track for example the `height`, we would see our user object but we would not see a `height` property. So any properties that do not exist on the model will be completely ignored as part of the update.

We can create an array of valid properties that we expect can be updated and this is stored in a variable. We can grab the request body from the HTTP request and store the key values as strings within another array object variable. We use the `.every()` array method which iterates through each property to check if the property meets the statement and returns true or false. So on the HTTP request body we can loop through each field passed in to check if the property matches/includes our expected field array. If a user passes in a field property that we are not expecting we can then send our own HTTP status error to indicate to the client that their update for a field was ignored.

In general the routing logic for updating endpoints are the most complex because we require additional validations to ensure the user is providing the necessary correct data before updating an existing record within the database.

## Resource Deleting Endpoints

The below code is an example for deleting endpoint using the `async/await` syntax:

task-manager > src > index.js:

```
1. const express = require('express')
2. require('./db/mongoose')
3. const User = require('./models/user')
4. ...
5. app.delete('/users/:id', async (req, res) => {
6.   try {
```

```
7.     const user = await User.findByIdAndDelete(req.params.id)
8.     if (!user) { return res.status(404).send( ) }
9.     res.send(user)
10.    } catch (error) {
11.      res.status(500).send( )
12.    }
13.  )
14. app.listen(port, ( ) => { console.log('Server is up on port ' + port) } )
```

Express provides us with a `.delete()` method which allows us to use the HTTP delete method. Like with all the other HTTP methods this takes in the URL path and a callback function as its two parameters. The URL for deleting is going to require the id for the document to filter a specific document to delete. Just having the id is enough for deleting a document.

In the mongoose documentation there are a few ways of deleting documents and in the above example we used the `.findByIdAndDelete()` method. This method requires the id only which we get from the request parameters id value. We use a if statement to check if there is a user to delete to add a conditional logic. If a user is found we send back the default 200 status along with the user data that was deleted from the database. If there was an error we would send back the error. This completes the delete endpoint.

We now know how to create endpoints for each of the CRUD operations for our resources.

## Separate Route Files

We are going to explore how we can re-structure our index.js file which contains every single route that our API supports. If we were to add more routes to the index.js file, this would make the file code very long and hard to manage. When we find ourselves in a situation with a single file that always gets longer and longer as we add more features, it is typically best practice to break that code up into many smaller files. For example, we can have one file containing all of the routes for users and another file containing all of the routes for tasks and so on. Therefore, we would setup multiple express routers and we will be combining them together to create the complete application. This will allow us to have as many routers as we need but it is typically make sense to categorise them by the resource. This allows us to stay more organised.

The goal in the index.js file is to use a few simple line of code to create a router which express provides to us on the our express object:

task-manager > src > index.js:

1. const express = require('express')
2. ...
3. const router = new express.Router()
4. router.get('/test', (req, res) => { res.send('Test from other router') } )
5. app.use(router)

We first create a variable to store the new express.Router() without passing in any arguments. We can now use methods on our router variable to customise it. The router is going to have the same methods we have

used such as `router.get()`, `router.get()`, `router.patch()` and `router.delete()`. The above example demonstrates the `router.get()` method.

By default the router created is not being used at all and if we were to go to the URL we would see the message `Cannot GET/task`. This is because although we have created the route, we have not registered the route with our express server. To fix this we need to register our new little router with our existing application. We do this via `app.use()` method passing in the router. With this in place we would be able to visit the exact same URL we had before but this time it is going to work without showing the error message.

So the basic structure is creating a new router, setting up those routes and then register it with the express application. The only thing we are going to do to stay organised, is to create the new router and set up those routes in a separate file.

We would create a new directory within the `src` directory called `routers` which is going to act as a folder storing all of our routers for example we can save `user.js` as a router for our `Users` resource. So this file code will look something like the below:

task-manager > src > routers > user.js:

1. `const express = require('express')`
2. `const router = new express.Router( )`
3. `router.get('/test', (req, res) => { res.send('Test from other router') } )`

### 3. module.exports = router

So in the above we are creating a new router, defining it and then exporting the router so that it can be registered in our index.js file in order to use that router.

task-manager > src > index.js:

1. const express = require('express')
2. const userRouter = require('./routers/user')
3. app.use(userRouter)

So in order to use the router from another file, we need to require it within our index.js file and then use/register that variable we stored it in using app.use() method. This now allows us to define the routers in a separate file and use them within our index.js keeping our code more neat and organised.

We now have a fully functional REST API that a client can use to interact and manage data within a database using the CRUD operations through the HTTP operations.

## Section 11

# API AUTHENTICATION AND SECURITY

### Introduction

In this chapter we are going to learn Authentication and data security. Currently, all of the API endpoints we create are publicly accessible. This means anyone can come along and do something such as delete ever data within the database and obviously this is a problem. So the goal of this chapter is to lock all of the API endpoints down putting it behind authentication. This would mean users would need to sign up and log in before they are able to do something such as creating a new document or fetching a list of documents from the database. By forcing a login, this will also going to allow us to setup a relationship between a user and the documents they created. This adds another level of security whereby a user 2 cannot fetch, update or delete documents created by user 1.

### Securely Storing Passwords

Securely storing user passwords is the most basic security that every application should do.

Current when taking passwords in from the user and store it in the database, we end up storing the password exactly as the user provided. This is known as storing the password in plain text and for passwords, this is a terrible idea. The problem with this is that for many users, they typically use the same password for multiple accounts. If the database got hacked, this would mean the hacker would have the data for all users which is not ideal, but worse we have exposed the user to further hacks because all of their credentials are out in the open and may have been used for other applications such as online banking, social network, email, etc. Therefore, we do not want to store the user password in plain text because this would leave users exposed to further problems should this information get out.

The solution is to store the password not as a plain text password but as a hashed password. The hashed value is going to look nothing like the plain text password and if someone was to hack the database and get a hold of this, it would be useless because the algorithm used to generate the hashed password is not reversible.

The algorithm we would use to create the hashed password is known as bcrypt which is a very secure and widely used hashing algorithm and is good for all sorts of cryptographically use cases including storing user passwords. We can use this by installing a npm module <https://www.npmjs.com/package/bcryptjs> using the following command:

```
:~task-manager$ npm install bcryptjs@2.4.3
```

This is going to install bcryptjs as a dependency within our application and once it is installed we can start using it within our code. Below is a code snippet to understand how the hashing encryption works:

```
bcrypt = require('bcryptjs')
const myFunction = async () => {
  const password = 'Example123!$'
  const hashedPassword = await bcrypt.hash(password, 8)
  console.log(password)
  console.log(hashedPassword)
}
myFunction()
```

The bcryptjs module uses promises and so we can use `async/await` functions to manage those promises. Above, we have a plain text password which a user would pass us stored in the variable called `password`. The `hashedPssword` variable would store the hashed password that we would end up storing. This is where we use the method from the bcryptjs library called `.hash()` which sends back a promise. This method takes in two arguments; the first is the plain text password which we have access to via the `password` variable and the second is the number of rounds we want to perform the hashing algorithm on the password. A nice balance is 8 which balances speed and security and is the recommended value by the creator of the algorithm.

We can `console.log()` out both variables to view the plain text password and its hashed password to see

what we get. Below is something we would see in the terminal if we were to run the script and the hashed password is what we would end up storing in the database:

Example123!\$

\$2a\$08\$1IZpyNJdhZYIAeEJs5/sOmJtc3N33qVTo4aeKEb8QDe6piNtAUne

There is an important distinction between hashing algorithms and encryption algorithms. With encryption we can get the original value back from the encrypted value. Hashing algorithms are one way algorithms which means we cannot reverse the process. Hashing passwords by design are not reversible.

Encryption Algorithm:

Original	Encrypted Output	Reverse Output
Password123	DSvu63vkx34E5gDdg	Password123

Hashed Algorithm:

Original	Encrypted Output
Password123	fDKw1e\$rgd%fvIVMjh83cnx/qreo

So we are probably wondering, how do we do something like logging in using a the users entered password and check to see if it matches the hashed password we stored in the database?

Hashing algorithms like bcrypt provide us an easy way to do that by hashing the plain text password that the user provides when logging in and we can compare that hash with the hash stored in the database. We can use the bcrypt method of .compare() to perform this:

```
const isMatch = await bcrypt.compare('Example123!$', hashedPassword)
console.log(isMatch)
```

The isMatch variable will be true when the hashed passwords matches and it will be false when they do not match the original hashed password used when signing up. The .compare() method also returns a promise so we can use the await to handle the promise. This method takes in two arguments, the first is the plain text password entered when logging in and the second is the hash which for us would be stored in the database but we have accessible via the hashedPassword variable.

This will give us a boolean of true and false when comparing the following hash and so behind the scene the bcrypt algorithm rehashed the plain text password and compared it with the original hashedPassword we had in place to determine whether there is a match or not. Therefore, we can use bcrypt .hash() and .compare() methods to securely store passwords while still being able to figure out if someone is logging in with the correct credentials.

Mongoose supports what is known as middleware. Middleware is a way to customise the behaviour of our mongoose model and it is going to allow us to do some pretty interesting things. Within the mongoose

documentation we have under guides a section called middleware. With middleware, we can register some functions to run before or after a given events has occurred. For example, we can run some code just before or just after a user is validated and we can also run some code just before or just after a user is saved and we have other events we could use as well.

If we take the example of the save event, our job is to run some code just before a user is saved to the database. What we can do is check for a plain text password and if there is, we can go ahead and hash it. To get something like this done, we can go into the model and do a little bit of restructuring to take advantage of middleware feature.

When we create a mongoose model, we are passing in an object in as the second argument. As we pass the second argument object, mongoose behind the scenes converts this into what is known as a schema. In order to take advantage of the middleware functionality, all we have to do is create the schema first and pass that in. This takes one line of additional code but once done it will allow us to take advantage of this advanced features.

task-manager > src > models > user.js:

1. const mongoose = require('mongoose')
2. ...
3. const userSchema = new mongoose.Schema( { name: {...}, email:{...}, password: {...}, age{...} } )
4. const User = mongoose.model('User', userSchema)
5. userSchema.pre('save', async function(next) {

```
6.   const user = this
7.   if (user.isModified('password')) {
8.     bcrypt.hash(user.password, 8)
9.   }
10.  next( )
11. }
12. module.exports = User
```

We use the new `mongoose.Schema()` constructor function to create a schema of our user and we pass in an object to this method as its first and only argument. When we do this, we now have access to this `userSchema` and we can pass that in as the second argument to `mongoose.model()` method.

We are therefore creating a separate schema and a separate model and this is what is going to allow us to take advantage of middleware. We can use a method on `userSchema` to set the middleware up. There are two methods accessible to us for middleware which are `.pre()` for doing something before an event and `.post()` for doing something just after an event. These methods take in two arguments, the first is the name of the event and the second is the function to run. We can only use a standard function and cannot use ES6 arrow function because the `this` binding plays a very important role and as we know, the arrow function does not bind `this`.

We have access to the `this` keyword which is the document that is being saved, so we are saying that we want to do something before the user are saved and the `this` variable gives us access to the individual user that is about to be saved. We can store the `this` document in a new variable and make reference to this new variable to make it easy to read our code.

We have access to the `next` argument within our function which allows our event know when we are done running our middleware code before it performs the event. If we do not provide the `next()` within our middleware, the programme will hang forever thinking that we are still running our middleware code before it continues to do the event i.e. `save`. Therefore, it is important to make sure that `next()` gets called in our middleware.

The `.isModified()` method provided by mongoose allows us to check whether a property which we pass as an argument has changed. This method returns a true or false value comparing the value with the original object data for the property in question. If this is true, we can run `brypt` to hash the password before saving it to the database.

Important Note: there are some mongoose methods such as `.findByIdAndUpdate()` which bypass the middleware code and directly interacts with our database. Therefore, we may need to change the way we perform some operation so that our code is consistent and does not bypass the middleware code when the API request are made. The tasing of passwords will now work when creating a new user or updating a user.

Middleware allows us to enforce the hashing of password without having to add the password hashing logic into multiple places like the two routes that are actually related to this situation. We provide it once and it works everywhere (provided we do not use those mongoose methods that bypass the middleware).

We now have a way to securely store passwords into our database without exposing our users original password and this adds a level of security to our applications.

## Logging In Users

Now that we know how to hash user passwords, we now need a new endpoint that will allow users to login with their existing account providing their credentials i.e. their email and password. It will be the job of that route to verify that there is a user with those credentials.

We would need to create a route within the users.js route and also setup a re-usable function within the user model for finding a user by their credentials. Below is an example code of achieving this.

task-manager > src > routers > user.js:

```
1. ...
2. router.post('/users/login', async (req, res) => {
3.   try{
4.     const user = await User.findByCredentials(req.body.email, req.body.password)
5.     res.send(user)
```

```
6.     } catch(error) {
7.         res.status(400).send( )
8.     }
9. })
```

#### task-manager > src > models > user.js:

```
1. ...
2. userSchema.statics.findByCredentials = async (email, password) => {
3.     const user = await findOne( { email } )
4.     if (!user) {
5.         throw new Error ('Unable to login')
6.     }
7.     const isMatch = await bcrypt.compare(password, user.password)
8.     if (!isMatch) {
9.         throw new Error ('Unable to login')
10.    }
11.    return user
12. }
13. const User = mongoose.model('User', userSchema)
14. module.exports = User
```

There are two ways in which we can structure this, the first way would be to add all of the code inside of the routers async function. So we would find the user by their email and then we go ahead and use the bcrypt.compare() method to compare the plain text password provided with the hashed password stored in the database. The alternative option would be to create a reusable function that does all of that for us and is the approach taken in the above example.

Within the try catch we setup a new method that we would call within the try block. We call something that we defined within our models user.js file. We have pre-built options/methods but can also define our own methods/options which we called .findByCredentials() which we made to take in the user email and password and it will try to find a user by the email, verify the password and it will either return the user or an error. The email and password will be provided by the body of the request.

The function will only work within the models user.js file so long as create a separate schema first and then pass that schema in to the model. If we were passing in an object as the second argument for the model then we would not be able to do what the example has done up above.

We use .statics. followed by the method name which is this case we are creating our own .findByCredentials() method operation (note: we could have called the function anything we want). We can set this as a async function and also use the arrow function as the 'this' binding is not going to play any role and we have our arguments which is the email and password.

Our function's job is to attempt to find the user by those pieces of information and we would start by finding the email first and then separately verify the password.

We are using the `.findOne()` method to find a single record where the criteria is to match the email with the value passed in by the user. We can use the shorthand ES6 syntax because the field name is the same name as the value name. If no user was found, we would throw a new error that no user was found and will end the execution of this function and trigger the catch block of our route.

If there is a user found we then want to verify that password against the hashed password using the `bcrypt.compare()` function. We can store this value within a variable value `isMatch` which will return either a true or false boolean value. The compare function takes in the plain text password which we get from the function argument i.e. `req.body.password` that is passed by the user and as the second argument we provide the hashed password. This comes from the user object we found from our `.findOne()` operation which we can use to access the hashed password to perform the validation.

If there are no match we would throw an error which will stop the execution of the function. We use the opposite of `isMatch` to check if the value of the variable is false and return true to run the truthy code block i.e. we are checking for not a match.

When providing error messages for things such as the login, it is best not to be specific because it exposes

more information than what we might want to expose. Someone who is trying to gain access to the account now knows more than they otherwise would. In general it is best to provide a single error saying the operation did not work. So unable to login is nice and generic.

The last thing we would do is return the user if they were found and the password is a match.

With this in place within the models user.js file, our route can have access to this method and in the above example we return the user object so we have a way to login to a account and be return the account details. We can now build off this to create a complete system. The catch block will catch any errors.

One of the basic things we should setup when creating a login system is a restriction on the email. If a user already has an account registered with a specific email, then another user should not be able to come along and use that same email to register an account again. This is usually accomplished by adding to the User schema a property called unique which is set to a boolean of true. This will create an index in the MongoDB database to guarantee uniqueness, similar to how our IDs are unique our emails now also need to be unique as well. The catch to this is that the database has to be wiped and recreated so that the index can be setup. This will now ensure users cannot create an account where the email is already registered and login show always work as expected.

This is not a complete process for logging in but it is a great step towards the right direction. We have a new route setup and the route knows how to verify a user credentials.

## JSON Web Tokens

We now know how to create a HTTP request for logging in and we are validating the credentials that are provided which is a great start. We are going to continue to explore logging in and truly allow users to login so that they can perform other actions later on such as creating a new task.

Every Express route we define will fall into one of two categories. It will either be public and accessible to anyone or it will sit behind our authentication and the user will have to be correctly authenticated to use it.

In most cases the only two route that are public are the signup and logging in routes while everything else would typically require the user to be authenticated. For example, if we are trying to delete some data from the database such as a task, we would want the user performing the action to be authenticated so that we can make sure that the user is the one who created the task that is being deleted.

We need to setup the login request to send back an authentication token. This is something that the requestor will be able to use later on with other requests where they need to be authenticated. Therefore, we can login with our account and we get the token back and then we can go off and edit our user profile or do something else like create a new task. So the question is how do we create and manage these tokens?

We would use what is known as a JSON web token or JWT for short. The JWT standard is very popular and can be used for all sorts of things including authentication. With JWT we can setup everything we want for

our authentication system. We can do things such as have a token expire after a certain amount of time so that users cannot stay logged in forever. Optionally, we could never expire the token and allow a user to use it indefinitely.

We can use a npm module called jsonwebtoken and its documentation can be found on <https://www.npmjs.com/package/jsonwebtoken>. This library provides us everything we need for creating authentication tokens and validating them making sure they are still valid i.e. making sure they have not expired. To install this module we can run the following command within the project directory:

```
:~task-manager$ npm install jsonwebtoken@8.5.1
```

Now that we have this installed we can explore the library and how it works.

```
const jwt = require('jsonwebtoken')
const myFunction = async () => {
  const token = jwt.sign( { _id: 'abc123' }, 'thisisauthenticationtoken', { expiresIn: '1 second' } )
  console.log(token)
  const data = jwt.verify(token, 'thisisauthenticationtoken' )
  console.log(data)
}
myFunction()
```

To create a new JSON web token we would use the `.sign()` method. This method takes two arguments, the

first is an object and the second is a string. The object contains the data that is going to be embedded into our token. So in the case of authentication the user id would typically be used to provide a unique value. The second argument is going to be a secret which is used to sign the token making sure that it has not been tampered with or altered in any way. All we need to provide is a random series of characters.

The return value from `.sign()` is our new token, in the above scenario this will be our authentication token. This token will be provided to the client and they can use the token later on to perform those privileged operations/routes.

Whatever we get back from `.sign()` is the value we are going to provide to the user. We can log this to the console to see what we would get back.

```
eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9.eyJfaWQiOiJhYmMxMjMiLCJpYXQiOjE1Njk5NjkzNzN9.3IPzQAE  
wBfE1iLiwAD1pk4aBiMPFA6B9zw1WWvmw5sc
```

This long series of character is our JSON Web token. The Web token has three distinct parts separated by the two periods. The first part is a base 64bit encoded JSON string which is known as the header. This contains meta information about what type of token it is i.e. a JWT and the algorithm used to generate it. The second piece is known as the payload or body. This is also a base 64bit encoded JSON string and this contains the data we provided, in the above this would be the `_id` value. The last piece is the signature and is used to verify the token.

The goal of the web token is not to hide the data we provided as the first argument, as this is publicly viewable to anyone who has the token and they do not need the secret to see that. The whole point of the JWT is to create data within the first argument object that is verifiable via the secret signature i.e. second argument. So if someone else was to come along and change the object data, they are not going to be able to do so successfully because they will not know the secret used with the algorithms and so things will fail.

We can take the middle part of the JWT and go to the following website <https://www.base64decode.org/> and paste this value in to decode it. We would be able to view the JSON object with two values. The first key:value is the one we provided in our code and the second "iat" value is the issued at which is a time stamp letting us know when the token was created.

```
{"_id":"abc123","iat":1569969373}
```

This is exactly what is embedded in the JWT. Now that we know how to create a token we can now focus on how to verify them. The `.verify()` method takes in two arguments, the first is the token we are trying to verify and the second is the secret to use. We would need to use the exact same secret that the token was created with. The `verify` method will return the payload for the token if things went well and the token is valid else it will throw an error. If we `console.log()` the data we should see the data back if verified. If things failed we would get a `JsonWebTokenError: invalid signature` error.

The final thing we can do is make a token expire after a certain amount of time and to do this, when we create the token, we can provide a third optional argument which is an object which we can customise with options. One option is `expiresIn`. This allows us to provide as a string the amount of time we want our token to be valid. For example, we could write '7 days', '2 weeks', '1 month' or in the above example '1 second'. We could set this to 0 seconds and this would expire as soon as it is created.

If the token expires and the `.verify()` method is run, this would throw a `TokenExpiredError: jwt expired`.

Now that we know how the `jsonwebtoken` library works we can now use this to create authentication tokens using the `.sign()` method and then later on we can use the `.verify()` method to make sure the user is authenticated correctly.

## Express Middleware

Every request to the API is going to require authentication with the exception of signing up and logging in. For everything else, the client is going to need to provide that authentication token and the server is going to validate it before performing whatever operation the user is trying to do. Express middleware is at the core of allowing us getting this done.

Below is a diagram of what Express middleware is going to allow us to do.

Without middleware: new request --> run route handler

With middleware: new request --> do something --> run route handler

So without a middleware a new request comes in to the server and the first thing that runs is our route handler. This is the only thing we setup to execute when Express maps the incoming request to the correct route handler and that function gets executed.

With middleware, we can customise the behaviour of the server to fit our needs and so we have something similar but slightly different with the step added in the middle. We still have a new request coming into the Express server but we then do something and this something is nothing more than a function that runs and we can setup this function to do whatever we like. This function could log out some statistics about the request so that we can keep track of it in our server logs or maybe we could check for a valid authentication token. Once the middleware runs we can continue to choose to run the regular route handler so that the given operation is completed successfully. Express middleware provides us with a lot of fine grain control over how we can customise our application. We do not need to setup our middleware function for every single route in the Express application as we can target individual routes.

To register a Express middleware, we would need to add this to the index.js file before calling app.use() to use the routers we created. This is very important when setting up a middleware function as this function will run before the routes functions are executed.

### task-manager > src > index.js:

```
1. ...
2. app.user( (req, res, next) => {
3.   console.log(req.method, req.path)
4.   next( )
5. }
6. app.use(userRouter)...
```

We have always used `app.user()` and providing something that was provided by express, however, we have never explicitly provided a function that we have defined ourselves. This is exactly what we would do when setting up our Express middleware. We pass in a single function which will run between the request coming to the server and the route handler actually running. This function will have the same information as the route handler i.e. the request and response along with an additional argument called `next`. The `next` argument is specific to registering the middleware.

With the above setup, if we were to run our request to the routes in Postman, we will notice that our request will hang on **Sending Request** until Postman times out and gives up. Our middleware function can do as much or as little as we want it to, but it is our job to call `next` if the next thing in the chain should run. If our middleware function never calls `next`, the route handler is never going to run. If we do want our route handlers to run we call `next()` without passing in any arguments to it. This will let Express know that we are done with the middleware function.

We can now add logic to our middleware to run or stop running our routes depending on the incoming request as demonstrated below:

task-manager > src > index.js:

```
1. ...
2. app.use( (req, res, next) => {
3.   if (req.method ==='GET') {
4.     res.send('GET requests are disabled')
5.   } else { next( ) }
6. })
```

So in the above example, we use the logical if statement to check the incoming request is a GET request. If this is true we would send back a response for why the request is not working. For all request that is not a GET request, the else clause will run next() which will run our route handlers. If we were to now run a GET request in Postman, we would now see the return message rather than the request hanging.

This is the exact same technique we would use to enable authentication to our routes.

Important Note: defining our middleware functions inline inside of the index.js file works well for experimentation but when we are creating middleware to be used throughout our application, it is best to define this in a separate file so that we can keep things nice and organised. Within the src directory, we would create another folder/directory called middleware.

This folder will contain each piece of middleware we are trying to define. We can then load this into our routers to put some of the routes behind authentication.

## CHALLENGE:

### Setup middleware for maintenance mode

1. Register a new middleware function
2. Send back a maintenance message with a 503 status code
3. Try your request from the server using Postman and confirm status/message shows

## SOLUTION:

### task-manager > src > index.js:

1. ...
2. app.use( (req, res, next) => {  
 res.status(503).send('Site is currently down. Check back soon!')  
6. } )

## Accepting Authentication Tokens

When setting up our middleware these will be stored in their own files within the middleware directory. However, if we would want certain requests to use the middleware wherever we need to, we would not register all of our middleware within index.js, this is because the middleware will be associated and used for every single route in our application. Instead, with our authentication middleware we would apply this in the router.

### task-manager > src > middleware > auth.js:

1. `const auth = async (req, res, next) => { console.log('Auth middleware' next( ) ) }`
2. `module.exports = auth`

### task-manager > src > routers > user.js:

1. `...`
2. `const auth = require('../middleware/auth')`
3. `router.get('/users', auth, async(req, res) => {...})`

We need to first load in the auth middleware into our routes file so that we can access the middleware function to an individual route. To add a middleware to an individual route, all we do is pass it in as the second argument to the route method before we pass in our route handler (i.e. the route handler will now become the third argument). In the above example we are using the auth middleware for this route. So now when someone makes a get request to /users its first going to run our middleware and then it will go ahead and run the route handler provided the middleware calls on the .next() function.

The whole authentication process starts with the client taking that authentication token that they get from the server when signing up or logging in and providing it with the request they are trying to perform.

We can go into Postman to the request where we added the auth middleware to. To actually provide the authentication token we would setup a request header by going to the Headers tab in Postman. From here we can setup key:value pairs to provide additional information to the server.

So when we work with headers we can provide them as part of the request i.e. sending them from the client to the server and we can also have headers sent back as part of the response i.e. the server can send back some headers to the original requestor.

In this case we would setup headers that get sent as part of the request as we are providing the authentication token to the server and to do this we would setup the following key:value pair.



The screenshot shows the Postman interface for a GET request to 'localhost:3000/users'. The 'Headers' tab is selected, showing one header entry:

KEY	VALUE	DESCRIPTION	...	Bulk Edit	Presets
Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfa...				
Key	Value	Description			

The authorisation header is going to have the following value starting with Bearer followed by the authentication token. This is known as a bearer token in which the client provides the token with the request they are trying to perform. This is all the client need to do to actually provide the information necessary to get authenticated.

Our auth function would look something more like the below code:

task-manager > src > middleware > auth.js:

```
1. const jwt = require('jsonwebtoken')
2. const User = require('../models/user')
3. const auth = async (req, res, next) => {
4.   try {
5.     const token = req.header('Authorization').replace('Bearer ', '')
6.     const decoded = jwt.verify(token, 'newjsonwebtokensecret')
7.     const user = await User.findOne( { _id: decoded._id, 'tokens.token': token } )
8.     if (!user) {
9.       throw new Error()
10.    }
11.    req.user = user
12.    next()
13.  } catch (error) {
14.    res.status(401).send( { error: 'Please authenticate.' } )
15.  }
16. }
17. module.exports = auth
```

We would load in the jsonwebtoken and the model file so that we can find the user in the database and compare the token. We use the try catch block to attempt to validate the user and if the user is not

validated we have the catch block to run. If the user is validated we can run our handlers using the .next() function.

We get access to the passed in header by calling req.header() and passing in a string which is the name of the header we are trying to get access to which is Authorization. We use the .replace() method to remove the 'Bearer ' from the token string and replacing it with nothing, leaving the token which we can use to validate. If no Authorization was returned this will give us a value of undefined and the replace method will throw an error, however, our try catch block will catch that no Authorization was provided and return the status 401 to the user.

To validate that the token was created by our server and has not expired we use the jwt.verify() method to check this and store it within our decoded payload variable. The decoded variable will also have the user id as part of the token and we can use this to grab the user from the database by the id. We use .findOne() as apposed to .findById().

The other thing we want to check is that the token is still part of the tokens array. When a user logs out we are going to delete that token and so we want to make sure it actually exists. We use tokens.token key as a string because it contains a special character.

We use the if statement to see if a user was returned from our findOne operation and if none we can throw

a new error without passing in any arguments as this would be enough to trigger our catch block. If the user was found that would mean things went well and we would do two things. The first is to make sure the route handler runs as the user has proven that they have been authenticated correctly. This means calling the next() function. The second thing we would do is to provide the route handler access the user that we fetched from the database. We have already fetched the user and therefore there is no need for the route handler to fetch the user again as that will waste resources and time. We use req followed by a name to store the value e.g. req.user and then store variable value i.e. user in the above case. We can now use the req.user within our handler to access the user object.

task-manager > src > routers > user.js:

```
1. ...
2. const auth = require('../middleware/auth')
3. router.get('/users/me', auth, async(req, res) => {
4.     res.send(req.user)
5. }
```

The /users route no longer serves any purpose as it would expose every users details to those who are authenticated and therefore we would repurpose this route to only return the authenticated user's own profile and we get access to this from the req.user we setup in the auth middleware.

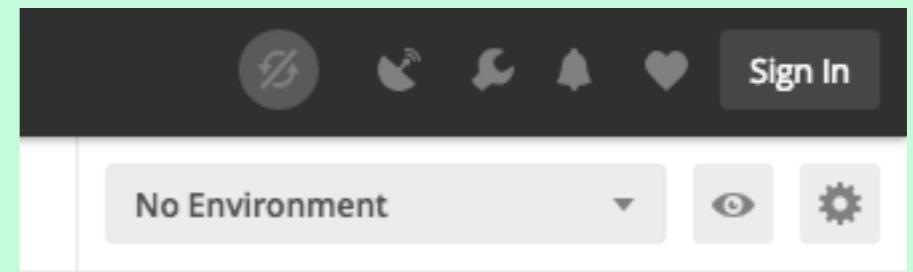
We now know how to accept authentication tokens within our middleware and apply it to a specific route.

## Advanced Postman

We are going to explore more advanced features of Postman to help with our workflow. There are two features we will look at in particular which are Postman Environments and Postman Environment Variables.

Currently all of our requests are being made to localhost:3000 and this is good because this is where our server is running. However, when we deploy the application API to the web host like Heroku, in this case we would have a different URL. We would also want to test those production endpoints from postman and this would mean that we would have to manually swap our URL's for a dozen or so different requests. With Postman Environments we can setup an environment with various variables and these can be used to configure how postman runs.

In the top right hand side of Postman we can see a No Environment dropdown with no other options to pick from but we can go ahead and create our own environment. If we click on the gear icon this will bring up the Manage Environments panel to setup an environment.



We can click on add to create our very first environment. We can give our environment a name and setup various key:value pairs for our environment variables that we would want to create. We can click on the add button to create a new Environment. Below is an example of setting up an environment.

**MANAGE ENVIRONMENTS**

Add Environment

Task Manager API (dev)

	VARIABLE	INITIAL VALUE <small>i</small>	CURRENT VALUE <small>i</small>	...	Persist All	Reset All
<input checked="" type="checkbox"/>	url	localhost:3000	localhost:3000			
	Add a new variable					

i Use variables to reuse values in different places. Work with the current value of a variable to prevent sharing sensitive values with your team. [Learn more about variable values](#)

[Cancel](#) [Add](#)

**MANAGE ENVIRONMENTS**

An environment is a set of variables that allow you to switch the context of your requests. Environments can be shared between multiple workspaces. [Learn more about environments](#)

Task Manager API (dev) [Share](#)  [...](#)

[Globals](#) [Import](#) [Add](#)

We now have a brand new environment and we can always create more. We are now able to select our environments from the dropdown list.

To use the URL variable within our dev environment, we can adjust the request URL. We can remove the localhost:3000 from the request url leaving the / followed by the pathname as this would change for every request. From here we can use an environment variable's value by using double curly brackets and entering the variable name within the curly brackets. With this in place, when we make a request, the url value will be used which the dev environment has defined. If we hover over the variable we can see the value going to be used.

**POST** Create user

▶ Create user

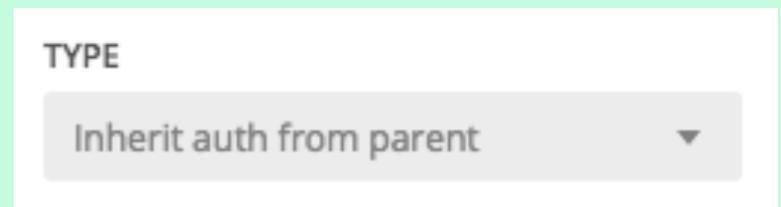
POST  {{url}}/users

We can have different environments with the url variable but with different values and our request would work on whichever environment we are in using the dynamic url variable.

Where our requests require authentication within the header we can take the token value, delete the header and head over to the Authorization tab. This tab provides us with a different way to setup authorisation and it is only different in terms of the user interface. At the end of the day, the exact same request is getting sent from Postman to the express server. From the Type dropdown we can switch it to the following Bearer Token (which is the strategy we are using for Authorisation) and then in the token field we can paste the token value. This is a different way to achieve the same functionality as using the header before.

The screenshot shows the Postman interface with the 'Authorization' tab selected. In the 'TYPE' dropdown, 'Bearer Token' is chosen. Below it, a note states: 'The authorization header will be automatically generated when you send the request.' A 'Learn more about authorization' link is provided. A 'Token' input field contains a long JWT token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiI1ZDkzYzdIOTZkZTEw...'. A 'Preview Request' button is at the bottom.

However, this is not the approach we are going to take. Instead we will change this back to the default of Inherit auth from parent. This is going to allow us to define our authentication scheme once and use it in every single request that has this type under the Authorization tab. This is the default value for all routes.



We can now go to our collection folder located within the collection sidebar and click on the Edit option.

Within the Authorization tab we are seeing the same option as before, however, this is the Authorization for every single request within this collection. Again we select the Bearer token and enter our token key.

We now have the token setup to be used on every request which has the authorisation setup to the type of inherit auth from parent to authenticate using the token without us having to setup the token manually in the header.



Where we have routes that do not use authentication, we can disable them by updating their Authorization type to no auth.

We can write JavaScript code to perform some automation. The one manual part of this workflow is either creating a user and logging in and then getting in that auth token and going back over to the collection menu to update the value and then being able to make the request. We can have all of this done automatically by writing just 3 to 4 lines of code. This is going to live within the test manu for both the Create and Login requests. We have a Pre-request Script tab where we can write our JavaScript code to run before the request is sent off. The Tests tab is where we can write our JavaScript code to run when the

request is received and we can write code to extract the token property from the body and set an environment variable whose value is equal to that token. From there we can use that environment variable inside of the collection to get everything working as expected.

So in the collection, if we edit the Authorization to no longer use the hard coded token value but instead use a environment variable which we can call authToken. Then in the Login request's Test tab we can write the following JavaScript code. Postman provides us access to an object called pm (which stands for Postman) which we can use when writing our pre-request and tests code. The pm.environment() method allows us to set an environment variable in our current environment. This takes in a key:value pair both represented as strings. The first is the environment variable name and the second is its value. Instead of passing in a string value we want to use the value from the pm.response.json(). This method will take the JSON response and convert it into an object and the object is the returned value. Therefore, on this object we only need to access the token value.

The screenshot shows the Postman interface with a collection named "Login user". It contains a single POST request to the URL {{url}}/users/login. The "Tests" tab is active, showing the following JavaScript code:

```
1 if (pm.response.code === 200) {  
2   pm.environment.set('authToken', pm.response.json().token);  
3 }
```

A tooltip for the "Tests" tab explains: "Test scripts are written in JavaScript, and are run after the response is received." A link "Learn more about tests scripts" is also visible.

We now need to use this request when our auth key expires to login and we should be able to run all our other requests that require authentication.

We can set this code up within our Create request but change the status to look for a 201 instead and we now have everything setup for both Creating and Logging in requests.

The screenshot shows a Postman collection named 'Create user'. The 'Tests' tab is selected, displaying the following JavaScript code:

```
1 if (pm.response.code === 201) {  
2     pm.environment.set('authToken', pm.response.json().token);  
3 }
```

Below the code, a note states: 'Test scripts are written in JavaScript, and are run after the response is received.' There is also a link to 'Learn more about tests scripts'.

We now have an automated system, where we have to log in once and everything else is handled behind the scenes for us. No longer do we have to copy and paste values around or manually change URLs and this is going to make it a lot easier to continue to use postman, especially as we setup authentication for the other endpoints that our API supports.

We will notice that within the Environment Management, we would see the authToken variable but this would not have an initial value because we did not manually typed anything in which is fine because the current value will be populated by our JavaScript code. There are no default authToken we would want as it should get set as the response from either Signing up or Logging in.

To conclude Postman environments is nothing more than a name for your environment and a key value pair of values you can access when working in that environment. So in this case, under the dev environment for the task manager API, we have url and authToken. Once we have our environments in place, we can switch between them and we can access those environment variables from anywhere such as the URL or the collections edit page.

## Logging Out

We are going to learn to create a new route to allow users to log out of the application. We already have the ability to login which generates the authentication token and allows verification against existing tokens when a user logs in. We now need to figure out how to remove tokens when a user logs out of the application. Below is a example route:

task-manager > src > routers > user.js:

```
1. ...
2. router.post('/users/logout', auth, async(req, res) => {
3.   try {
4.     req.user.tokens = req.user.tokens.filter( (token) => { return token.token !== req.token } )
5.     await req.user.save( )
6.     res.send( )
7.   } catch (error) { res.status(500).send( ) }
8. }
```

This route is going to require the authentication middleware because the user must be authenticated in order to logout. We would make a slight change to the auth.js middleware by adding token to the request so that it can be accessed later similar to how we did this for the user.

task-manager > src > middleware > auth.js:

```
1. const jwt = require('jsonwebtoken')
2. const User = require('../models/user')
3. const auth = async (req, res, next) => {
4.     try {
5.         const token = req.header('Authorization').replace('Bearer ', '')
6.         const decoded = jwt.verify(token, 'newjsonwebtokensecret')
7.         const user = await User.findOne( { _id: decoded._id, 'tokens.token': token } )
8.         if (!user) {
9.             throw new Error()
10.        }
11.        req.token = token
12.        req.user = user
13.        next()
14.    } catch (error) {
15.        res.status(401).send( { error: 'Please authenticate.' } )
16.    }
17. })
18. module.exports = auth
```

The other route handlers will now have access to that token and we can use it to delete it directly off of that user profile.

Since we are already logged in, we have access to the `req.user` which is the user which means we can change `req.user.tokens` and use the `save` method to save our changes. We want to set the `tokens` array equal to a filtered version of itself by using `req.user.tokens.filter()` and we will provide out callback function. We get access to the individual token which has both the `_id` and `token` values. All we are going to do is return true when the token that we are currently looking at is not the one used for authentication. If they are not equal, we are going to return true to keep it in the `tokens` array and if they are equal we are going to return false, filtering it out and removing it. Nothing will get saved until we call the `save()` method.

We can then send a default 200 response if things went well and if things did not go well, our catch block will send a 500 response. Now that we have this in place, users now have the ability to logout and we can now setup a test endpoint within Postman to test that everything is working as expected.

## **CHALLENGE:**

**Create a way to logout of all sessions**

1. Setup Post /users/logoutAll

2. Create the router handler to wipe the tokens array
  - Send 200 or 500
3. Try your request from the server using Postman and confirm status shows

## SOLUTION:

task-manager > src > routers > user.js:

```
1. ...
2. router.post('/users/logoutAll', auth, async (req, res) => {
3.   try {
4.     req.user.tokens = []
5.     await req.user.save( )
6.     res.send()
7.   } catch (error) {
8.     res.status(500).send( )
9.   }
10. })
```

## Hiding Private Data

We are able to protect/hide data that we are sending back so that we protect the users credentials. For example, we would not want to send back the hashed password even to a logged in user as it is no use to them. Furthermore, we can send back the auth token; however, we would not want to send back the array.

Again this array of authentication tokens is no use to the logged in user as they would only need the current token that they are using to authenticate only.

We can suppress the data returned back to the client by making a few changes. Below is the example for the Login route which sends back the user and token on a successful login.

task-manager > src > routers > user.js:

```
1. ...
2. router.post('/users/login', async (req, res) => {
3.   try {
4.     const user = await User.findByCredentials(req.body.email, req.body.password)
5.     const token = await user.generateAuthToken( )
6.     res.send( { user, token } )
7.   } catch (error) {
8.     res.status(400).send( )
9.   }
10. })
```

We can set the user property to something else using the shorthand syntax:

```
6. res.send( { user: user.getPublicProfile( ), token } )
```

The .getPublicProfile() does not exists but we could go ahead and create that function and set it up to

return the public data that we should be exposing about the user. So over inside of the user.js models file we can go ahead and add a new method on:

task-manager > src > models > user.js:

```
1. ...
2. userSchema.methods.getPublicProfile = function () => {
3.   const user = this
4.   const userObject = this.toObject()
5.   return userObject
6. }
```

We have userSchema.methods for methods on the instance i.e. individual user and we have userSchema.statics for methods on the actual model i.e. uppercase User. The above code creates the method on the individual instance. We set the method as a standard regular function and not an ES6 arrow function because we want to use the this key binding and so we do not want to use an arrow function.

We create a variable to get all of the raw object with our user data attached (*this will remove all of the stuff that mongoose has on there to perform things like the save operation*) by using the mongoose function toObject(). This will give that raw profile data and then we can go ahead and return it.

The code above is not going to do anything special. We have essentially recreated the same behaviour we have had before but with more code in place. However, we are now able to manipulate this userObject to change what we expose. So with the below we can see that we limit what we return back:

## task-manager > src > models > user.js:

```
1. ...
2. userSchema.methods.getPublicProfile = function () => {
3.   const user = this
4.   const userObject = this.toObject()
5.   delete userObject.password
6.   delete userObject.tokens
7.   return userObject
8. }
```

We use the delete operator to delete properties off of that object such as the password and tokens. If we now fire off this request in Postman, we would no longer see the password and tokens in our response data. We now have an object where we provide the data we want to share and hide private data that we do not want to share.

We now have a manual way of getting the job done. It is manual because we have to call our function `.getPublicProfile()` every single time we are sending the user back. However, there is a way to automate this which would not require us to manually make any changes to our route handlers which is the second solution which requires a small tweak to what we have already done.

Firstly we would change the route `res.send()` method back to how it original was which sent back the user

and token without calling the `.getPublicProfile()` method. Then within the `user.js` models file we would change the `userSchema.method.getPublicProfile` to exactly as `userSchema.method.toJSON` instead.

task-manager > src > routers > user.js:

6. `res.send( { user, token } )`

task-manager > src > models > user.js:

2. `userSchema.methods.toJSON = function ( ) => {`

If we now test this out in Postman, this will continue to work and hide the private data. This change would mean all our other routes which sends back the user data will display the data without displaying the password or tokens array.

When we send an object to `res.send()`, express is calling `JSON.stringify()` behind the scenes which converts the JSON object into a string representation. When we setup `.toJSON()` it is going to get called whenever that object gets stringified. The `.toJSON` returns us the object along with the `toJSON` function and we now have a way to manipulate what exactly comes back when we stringify the object by returning a object from `toJSON()` function.

Therefore, we use `JSON.stringify()` on the user object when we call `res.send(user)` and then we have manipulated the object when we called the `.toJSON()` method to remove the password and tokens from the object which ends up sending back just the properties we want to expose and the end result is that we no longer see password or tokens any time the user gets sent back to the client.

## The Resource Relationship

We are going to look at how to create a relationship between resources for example a User and the Tasks that they have created. This is important to make sure users can access and manage their tasks and they cannot mess with someone else's tasks.

The three files we would need to look at to achieve this is the user.js model, task.js model and the task.js router files. Below is the example will demonstrate how to create such as relationship with a user and tasks resources. We would start by looking at the task.js model

task-manager > src > models > task.js:

```
1. ...
2. const Task = mongoose.model('Task', { ...,
3.   owner: {
4.     type: mongoose.Schema.Types.ObjectId,
5.     required: true
6.   }
7. }
```

There are two ways we can setup a relationship between a user and a task. The user could store all of the id's of the task they have created or the individual task could store the id of the user who created it. The above takes the latter approach. This is the better approach and the only change required is to add a single filed to task which will store the id of the user who created it which will allow us to lock down the task

management later on. The type for this field is going to be an ObjectId and this is also going to be a required field. After this change to the data structure, we would want to drop our database and recreate it again to force the task to have Owners.

We can now head over to the task.js router file to make a change to the very first endpoint that is responsible for creating a new task.

task-manager > src > routers > task.js:

```
1. ...
2. const auth = require('../middleware/auth')
3. router.post('/tasks', auth, async (req, res) => {
4.   const task = new Task( { ...req.body, owner: req.user._id } )
5. }) ...
```

The first thing we need to do is load in the auth middleware so we can actually use it as the second argument to our route. We create a new task variable which uses the new operator with the Task constructor function, although just requesting the req.body, we are going to provide our own object. We would want all of the properties and values from req.body with the addition of an Owner property. We would start off with the ES6 spread operator ... followed by the req.body which will copy all of the properties from body over to this object. We can then add the Owner property and get its value from the auth user object which we have access to in req.user.\_id to grab the user id. With this in place, when we tasks get created, we have the body data along with the owner of the task to create the association.

This is the piece of data that links task to their owner. We would use the Owner property to make sure that someone can actually read, update and delete a given task. We need to confirm on the server that the authenticated user is the one who created the task.

With mongoose we can setup the relationship between our models and this is going to provide us with some helper functions that will make this possible with very minimal code. We can continue to build off of our task.js model Owner property and add a additional option of ref.

#### task-manager > src > models > task.js:

```
1. ...
2. const Task = mongoose.model('Task', { ...,
3.   owner: {
4.     type: mongoose.Schema.Types.ObjectId,
5.     required: true,
6.     ref: 'User'
7.   }
8. })
```

The ref property allows us to create a reference from this field to another model. In the above we created a reference to the User model by typing out the model name exactly as we had typed in the file itself to create the relationship. Now that we have this in place, we can fetch the entire User profile whenever we have access to an individual task.

The `.populate()` method allows us to populate data from a relationship such as the one we have for Owner. We pass to `.populate()` the thing we want to populate and we then use the `.execPopulate()` to actually fire this off. Below is a sample code of using this to populate the entire Owner data.

```
const Task = require('./model/task')
const main = async () => {
  const task = await Task.findById( { '5d97957ecd4d0704e9127079' } )
  await task.populate('owner').execPopulate()
  console.log(task.owner)
}
```

This line of code will go off and find the user who is associated with this task and logged `task.owner` will now be there entire profile document as opposed to only the user's id. So by adding both the ref and this `.populate().execPopulate()` lines of code we can get the entire user document for the user who created the task. We now have a relationship between task and user resources. We can use `populate` to figure out which user created which task or which tasks a user owns.

The below is the reverse of the above whereby we take a user and find their tasks.

```
const User = require('./model/user')
const main = async () => {
  const user = await User.findById( { '5d9794ecc4acbc04daee96ba' } )
  await User.populate('tasks').execPopulate()
```

```
    console.log(user.task)  
}
```

Now user.tasks does not exist because users do not have tasks on that document and this would be true as we would get undefined if we tried to console.log() it. We are not going to create a tasks array on the users.js model. The tasks live in a separate collection, instead what we are going to do is setup what is known as a virtual property within our user.js model file:

task-manager > src > models > users.js:

1. ...
2. const User = mongoose.model('User', { ... })
3. userSchema.virtual('ownerTasks', {  
 ref: 'Task',  
 localField: '\_id'  
 foreignField: 'owner'  
})
4. })

A virtual property is not actual data stored in the database, it is a relationship between two entities, in the above case this is between the user and their tasks. We use something on userSchema called .virtual() method and it is virtual because we are not actually changing what we store for the user document. It is just a way for mongoose to figure out how these two entities are related. We pass two arguments to .virtual(), the first being the name for our virtual field (we can choose any name we want) and the second argument is an object. Within the object we would configure the individual field. We would setup ref property as we

did for the task model and set this to the Task.

So inside of the task.js model we have a reference to the User model on Owner which is a real field stored on the database. Inside of the user.js model we have a reference between the user and the task on a virtual which is not stored on the database and it is just for mongoose to be able to figure out which user owns which tasks and how they are related.

On the virtual we need to specify two other fields to get things working. We need to specify the localField along with the foreignField properties both taking a string value. The foreignField is the name of the field on the other thing, in the above case is the Task, that is going to create this relationship which we setup to be the owner. The localField is where the local data is stored. We have the owner field on the task which is associated with the \_id field. Therefore, the localField users id is a relationship between that and the Task owner field.

With this in place we can use the .populate().execPopulate() method again to get all tasks a user created and store it in an array on the user.task property which will mean the code will now work and get printed to the console.

We now have a relationship between tasks and users and we can now use the topics of virtual and populate within our routes to create those relationships.

We can now take this knowledge and update our different task route as seen below:

task-manager > src > routers > task.js:

```
1. ...
2. const auth = require('../middleware/auth')
3. router.post('/tasks/:id', auth, async (req, res) => {
4.   const task = await Task.findOne( { _id, owner: req.user._id } )
5.   try { ... } catch (error){...}
6. }) ...
```

We use the `findOne()` method to find a document filtered by multiple fields. This is to ensure the tasks document found by the id actually belongs to the user requesting it in order to make any updates to the task. This is enough to prevent a user from retrieving a document that they are not the owner of.

Again we can get all of the tasks of a user using the `find` method and narrow down using the `req.user._id`:

task-manager > src > routers > task.js:

```
1. router.get('/tasks', auth, async (req, res) => {
2.   try {
3.     const tasks = await Task.find( { owner: req.user._id } )
4.     res.send(tasks)
5.   } catch (error) { res.status(500).send( ) }
6. })
```

The alternative method is to use the `.populate().execPopulate()` method to get back all the tasks of a user:

task-manager > src > routers > task.js:

```
1. router.get('/tasks', auth, async (req, res) => {  
2.     try {  
3.         await req.user.populate('tasks').execPopulate()  
4.         res.send(req.user.tasks)  
5.     } catch (error) { res.status(500).send() }  
6. })
```

Either methods above would get us our tasks for our authenticated user. We can now have our routes to require authentication and they also take that into account when it comes to the relationship between the authenticated User and the task they are trying to work with. So when a task is created it is associated with the user as the owner of the task and whenever the user tries to read, update or delete a task, we make sure it is a task that the user has actually created adding a layer of security to the application.

## Cascade Delete

If a user wishes to delete their profile from our application, their user documents should be deleted but also their associated resources such as their tasks, otherwise their data will sit around in the database forever. To get this done we can take one of two approaches, the first is to head to the place where the users get deleted i.e. the `users.js` router where we have our delete route. We can update this route to add some code to also remove the users tasks.

### task-manager > src > routers > user.js:

```
1. router.delete('/users/me', auth, async (req, res) => {  
2.   try {  
3.     await req.user.remove( )  
4.     //add code to remove a users task  
5.     res.send(req.user)  
6.   } catch (error) { res.status(500).send( ) }  
7. })
```

The alternative approach is to use our middleware approach. Although the above is the one time we would remove a user and we could add the logic to remove the associated resources for that account, it is good idea to get in the habit of using middleware. So if we were to allow the application to be deleted elsewhere in the application, we would also have their tasks deleted as well. To do this we would head over user.js model to create our middleware:

### task-manager > src > models > user.js:

```
1. ...  
2. const Task = require('./task')  
3. userSchema.pre('remove', async function (next) {  
4.   const user = this  
5.   await Task.deleteMany( { oner: user._id } )  
6.   next( )  
7. })
```

We already have one middleware for hashing passwords and we would now have a second to cascade delete tasks. We would load in the task model and store this in a Task variable. We can then use the userSchema.pre() method to create the middleware and have it trigger on the remove event and call a async function. We can then store the this keyword to a variable called user. We can then take the Task model and use the .deleteMany() to delete multiple tasks. We can set the filter where the owner property is equal to the user id that is being deleted.

We are now done and now whenever a user is removed, this code will run removing their tasks as well.

To conclude, we should now know how to setup authentication whereby we have a system for signing up, logging in, logging out and managing user data all through a secure application routes.

## Section 12

# SORTING, PAGINATION AND FILTERING

### Working with Timestamps

We going to explore how we can enable timestamps for our mongoose models. We are going to enable an options which is going to add two new fields. One is called createdAt and another called updatedAt. Both are going to store timestamps i.e. the moment in time when that event occurred. This would mean we would know when the record was first created and we will also know when the record was last updated. This will provide us with more information to use when building out our application.

To enable this it is very easy, and we simply customise our mongoose.Schema() by passing in a second argument which is also an object. On this object we can add some of the options available to us and the one we are going to explore for the moment is called timestamps which we are going to set to true to enable them in our model. By default this option is set to false. Now that we have this in place, anytime we create a new document using the model, they are going to be

created with those two additional fields allowing us to track when they were created and when they were last updated. So it is best to wipe our database for the last time and recreate it and then add a new document to test this option out.

task-manager > src > models > user.js:

1. ...
2. const userSchema = new mongoose.Schema( {...}, { timestamps: true } )

The timestamp is something we can use on the server and it is also something that the client could use when rendering the content to the browser. Now that we have this in place, we have just a little more data for each of our documents in our database making it more useful and easier to work with. This can be used for sorting, filtering or anything else our application needs.

## Filtering Data

We can provide useful options to allow our users/consumers of our API to better target the data they are trying to fetch. This would help narrow down the returned values from an array of objects that is being returned back from a fetch route that brings back multiple documents.

We do this by using the query string as part of the URL structure. The below provides an example of using query strings to allow our consumers to filter the data by adding options to the URL string and using that information to filter the search and returned data.

## task-manager > src > routers > task.js:

```
1. ...
2. router.get('/tasks', auth, async (req, res) => {
3.   const match = {}
4.   if (req.query.completed) { match.completed = req.query.completed === 'true' }
5.   try {
6.     await req.user.populate(
7.       { path: 'tasks', match
8.     }).execPopulate()
9.     res.send(req.user.tasks)
10.    } catch (error) { res.status(500).send() }
11.  })
```

In the above we use the query string ?complete= to either true or false which will return back the tasks that are either completed or not. We can customise .populate() by providing an object. We still need to let it know that we are trying to populate the tasks and so the path must be set to the property. So far we have only refactored the code to allow us to explore new options. One of the new options we have access to is called match. Match is an object and in here we can specify exactly which tasks we are trying to match for example we can match those where completed is equal to true.

We can run our request in postman and see what document is returned back that matches the completed property of true. At the end of the day we want to customise this object so that it gets its value from the

query string values provided rather than hardcoded.

In express we can get access to the query string by using `req.query`. followed by the string key, in the above case this is `completed`. Now that we have this value, it is our job what to provide to the `match` object. We create a variable called `match` and store the object which we can then pass the variable as the value to `match` and since they use the same name we can use the property shorthand.

We can modify the `match` object so that if `completed` is not provided, we would not change it at all to get all tasks back and if `completed` is provided we will set a `completed` property to the correct boolean value. If someone types in `true` or `false` we do not get back a boolean but a string value. We would need to convert this into a boolean value. To do this we are going to see if `req.query.completed` string is equal to the string of `true` and if this is correct then we would set the `match.completed` value to the boolean of `true`. If the `req.query.completed` string does not equal to `true`, whether it is `false` or anything else, then `match.completed` will be set to equal to the boolean `false`.

This is going to give us exactly what we want and therefore, we are able to get back our data in three distinct ways: either all tasks, the completed tasks or the incomplete tasks. The consumers of our API are now able to filter the data to retrieve more or less data back from the database.

## Paginating Data

We can provide another option for our API to allow a consumer to further refine the data they get back. This option is known as pagination. Google is a good example of pagination. When we search for a subject within the Google search engine this will return the number of results e.g. 6,000,000; however, all 6million results will not be displayed on the page all at once. Instead we would have the first 10 results out of 6million results. If we would want to see more, at the bottom of the page we can click on the page 2 or page 3 to retrieve and view the next 10 result and so on. This is pagination.

The idea of pagination is to be able to request pages of data so that we are not requesting everything all at once. We can implement pagination in many ways such as page numbers at the bottom of the page, or a load more button, infinite scroll, etc. Regardless how we fetch the next set of data, the backend is going to be the same. We need a way for people requesting their tasks to specify which page of data they are trying to fetch. We can do this by setting up two new options for this request. The first is limit and the second is skip.

The limit option allows us to limit the number of results we get back for any given request. If we wanted to limit the results to 10 we would set limit equal to 10 in the query string e.g. GET /tasks?limit=10. The other half of the equation is skip which is also going to get set to a number and this is what allows us to iterate over pages. So as an example:

Get /tasks?limit=10&skip=0 means we are skipping 0 and getting back the first set of 10 results

Get /tasks?limit=10&skip=10 means we are skipping 10 and getting back the second set of 10 results

Get /tasks?limit=10&skip=20 means we are skipping 20 and getting back the third set of 10 results

The above shows how we can use both limit and skip to get a pagination effect which we can customise however we want it to return the number of documents we want. So we can have pages of 1, 50, 1000, etc. By providing the support for skip and limit will allow the client to customise how they want to use pagination.

Now that we have seen how the URL structure is going to look like, we need to figure out how to provide this information to mongoose so that mongoose can correctly populate the tasks and the answer to this is providing an options property:

task-manager > src > routers > task.js:

```
1. ...
2. router.get('/tasks', auth, async (req, res) => {
3.   const match = {}
4.   if (req.query.completed) { match.completed = req.query.completed === 'true' }
5.   try {
6.     await req.user.populate(
7.       { path: 'tasks', match, options: { limit: 2 } }
8.     ).execPopulate()
9.     res.send(req.user.tasks)
```

```
10. } catch (error) { res.status(500).send( ) }  
11. })
```

This options property can be used for pagination as well as sorting the data. We have access on options to both the limit and skip properties. So in the above we set limit to equal to 2 in order to limit the returned data to 2 tasks.

We can change the above code to allow the user to provide the limit by using `parseInt()` which is a function provided by JavaScript which allows us to parse a string that contains a number into an actual integer which mongoose expects for the value of limit. We must remember when a user provides a value in the query string, it is always going to be a string and we saw this in the filtering data where we needed to convert the value from a string to a boolean.

```
7.           path: 'tasks', match, options: { limit: parseInt(req.query.limit) }
```

So here we are taking the limit if one was provided in the URL query string and parsing it and setting it equal to the value. If the value is not provided or is not a number, this is going to be ignored by mongoose which means that we are only going to limit when it is actually a number that is provided.

We can now use the URL `GET /tasks?completed=true&limit=2` to return back 2 data where the completed value is set to true allowing our consumers to limit the number of data returned back from the API request.

We now need to support the option for skip to enable our consumers to get the next lot of data. To do this we would update the above code to add in the skip option.

```
7.      path: 'tasks', match, options: {  
8.          limit: parseInt(req.query.limit),  
9.          skip: parseInt(req.query.skip)  
10.     }
```

We can now use the URL GET /tasks?completed=true&limit=2&skip=2 to return back the second lot of 2 task data where the completed value is set to true.

So now that we have this in place we now have pagination setup for our API route. This will allow our users to really customise their options for retrieving their data and narrow the number of data that is returned back from their request to the API.

## Sorting Data

The final option will allow our consumers of our API to sort their data using the timestamps i.e. they can sort by when documents was last created or last updated or any other values such as the completed tasks first or the incomplete tasks first.

The URL structure for the sort would be GET /tasks?sortBy=createdAt\_asc

We can call this sort or sortBy or anything else we want and this is going to allow someone to specify the sorting criteria and there are typically two pieces to the value. The first is the field we are trying to sort by and the second is the order. The order can be either in ascending (asc) or descending(desc) fashion.

There are a few ways we can add the order value, the above uses an underscore to separate the field from the order but we could use any other special character such as the colon, pipe, etc (as long as it is not a reserved special character such as the & or ? in a query string).

Below is the updated .populate().execPopulate() values to add the sorting option within our API request.

task-manager > src > routers > task.js:

```
1. ...
2. router.get('/tasks', auth, async (req, res) => {
3.   const match = { }
4.   if (req.query.completed){ match.completed = req.query.completed === 'true' }
5.   try {
6.     await req.user.populate( {
7.       path: 'tasks', match, options: {
8.         limit: parseInt(req.query.limit),
9.         skip: parseInt(req.query.skip),
10.        sort: createdAt: -1
11.      }
12.    }).execPopulate( )
13.    res.send(req.user.tasks)
```

```
14. } catch (error) { res.status(500).send( ) }
15. })
```

We add another option to the options object called sort and in here we pass in an object value picking the fields we would want to sort by. The value to the field of 1 will sort by ascending and -1 will sort by descending. This is how we are going to actually set things up in terms of the call to .populate().

This is the basic behaviour of sort and to make the above code dynamic and not hardcoded so that the user can provide the value they wish to sort by within the URL query string. This is going to be similar setup to the find (match) operation. We would create an empty object and use it down below and there is a change we can customise it. So the code will look something like the below:

task-manager > src > routers > task.js:

```
1. ...
2. router.get('/tasks', auth, async (req, res) => {
3.   const match = {}
4.   const sort = {}
5.   if (req.query.completed) { match.completed = req.query.completed === 'true' }
6.   if (req.query.sortBy) {
7.     const parts = req.query.sortBy.split('_')
8.     sort[parts[0]] = parts[1] === 'desc' ? -1 : 1
9.   }
10.  try {
```

```
11.     await req.user.populate( {
12.         path: 'tasks', match, options: {
13.             limit: parseInt(req.query.limit),
14.             skip: parseInt(req.query.skip),
15.             sort
16.         }
17.     }).execPopulate( )
18.     res.send(req.user.tasks)
19. } catch (error) { res.status(500).send( ) }
20. })
```

So we have a sort object variable which we will manipulate based on the user URL query string and set this value to the sort options and therefore can use the ES6 shorthand syntax because the variable and option share the same name. We can then use the if statement to check if the sort value was provided and if it was we can parse the data. If a value was provided we need to break up the value by splitting it by the special separator character so that we can get the two individual pieces of information. We can use the string .split() method to split by a denominator by passing in the special character as the argument. We can use these parts to setup the sort property.

Now we do not have a static name because it is not hardcoded and the user is providing the name of the property we need to set and so we use the bracket notation and the value comes from a variable in the

parts array which is the first one. So we are using the first part in the array and using as the name for the property we are setting up e.g. sort.createdAt.

With that in place, we have to set that property value to equal to either 1 or -1 for ascending or descending sorting. We are going to convert the asc or desc to a 1 or -1 using the ternary operator. To do this we look at the second object in parts and see if it is equal to the following string of desc. With a ternary operator the resolved value is one of two things. So if the statement is true the value is whatever we put after the question mark and if it is false it is whatever value we put after the colon. The ternary operator has three components which are the condition, value for when the condition is true and value for when the condition is false. Therefore, if the value of parts is desc then we would set the value to -1 else anything else would be set to 1. This will ensure the sort object is setup correctly.

With this in place we are actually done with providing a sort option to our consumers. Therefore, following a URL link to something like URL GET /tasks?completed=true&limit=2&skip=0&sortBy=createdAt\_desc will narrow the returned data to where the completed tasks is set to true limiting the number returned of documents to 2, skipping 0 records and sorting by descending order.

Our consumers are now able to string up all of the available options to do multiple things at the same time such as Sorting, Paginating and Finding their data as seen in the above example. This gives the client the ability to really customise the data that comes back from the API request.