The Complete Developers Guide

Section 1

INSTALLING & EXPLORING NODE.JS

Installing Node.js

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

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

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

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

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

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

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

Installing A Code Editor

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

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

Links:

https://code.visualstudio.com/

https://atom.io

https://www.sublimetext.com/

What is Node.js?

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

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

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

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

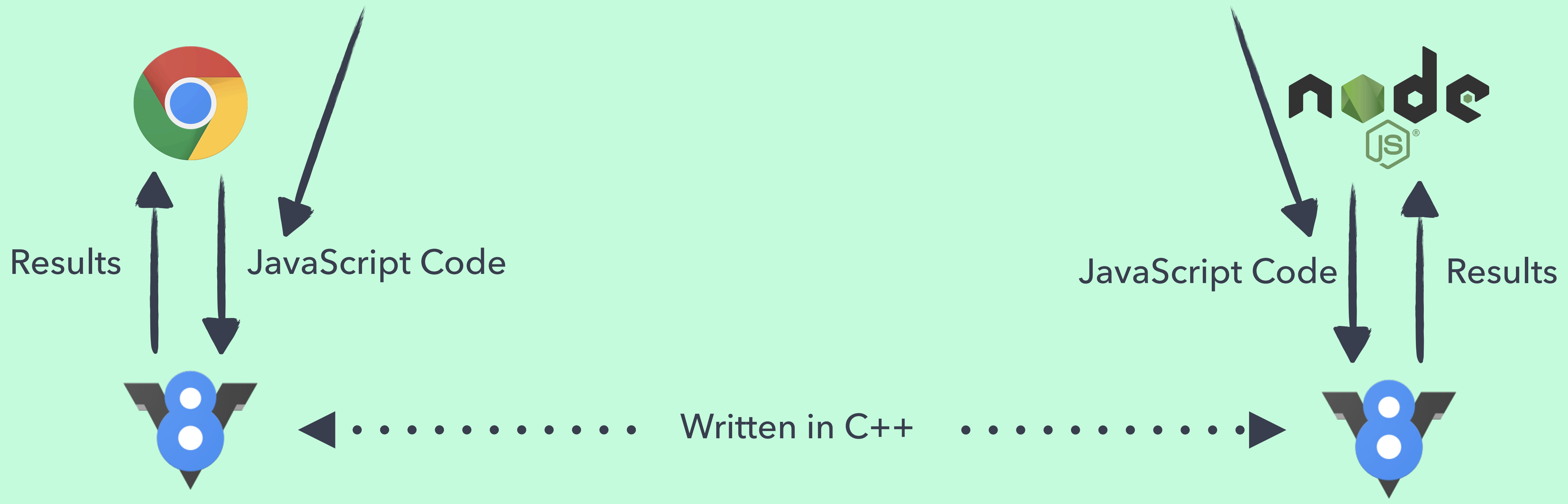
Node.js is not a programming language. The runtime is something that provides custom functionality i.e. various tools and libraries specific to an environment. So in the case of Chrome, the runtime provides V8 with various objects and functions that allow JavaScript developers in the chrome browser to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

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

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

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

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



We can run the following code in our terminal

```
:~$ node
```

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

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

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

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

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

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

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

Why Should You Use Node.js?

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

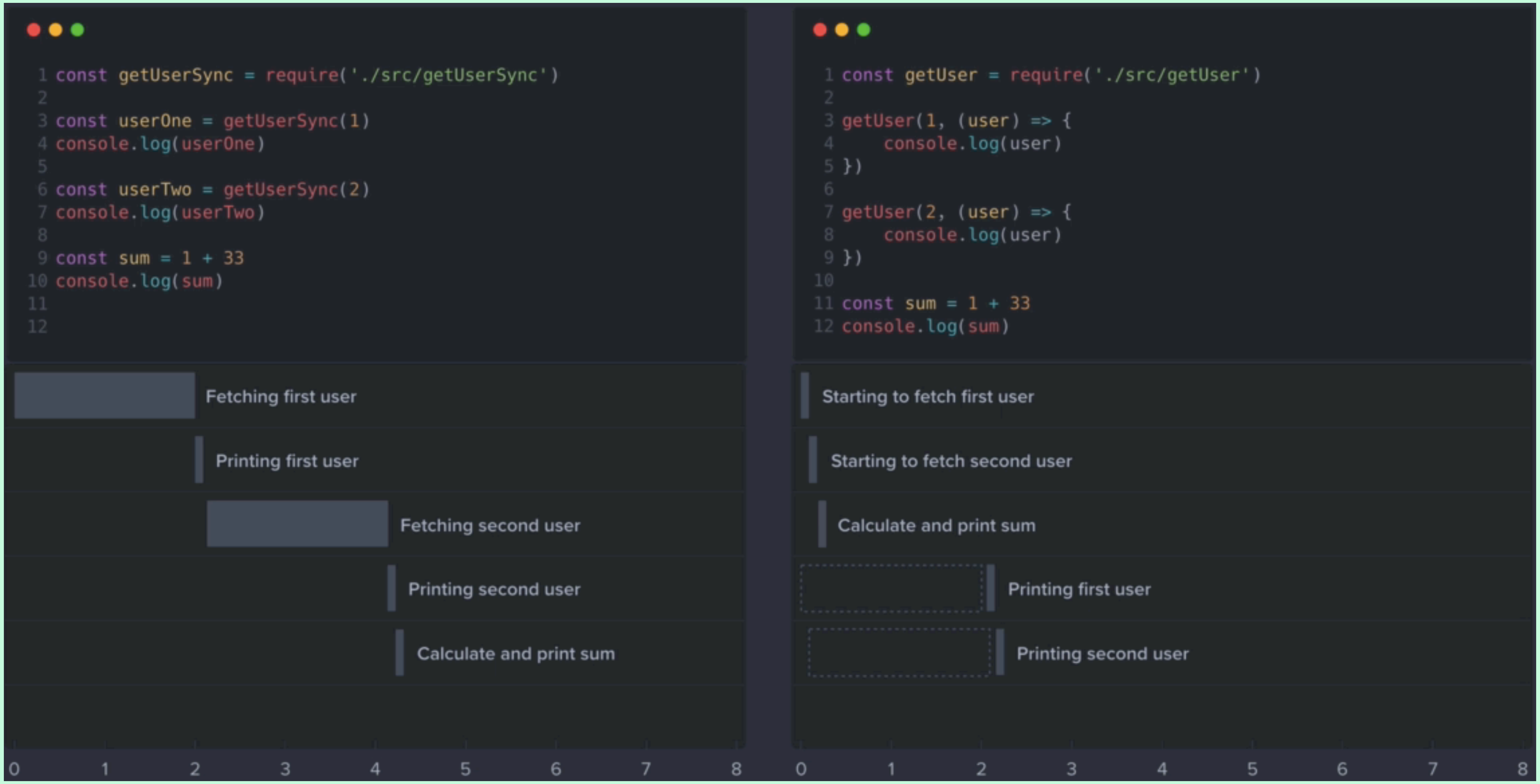
> *"Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js package ecosystem, npm, is the largest ecosystem of open source libraries in the world."*

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

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

Blocking I/O

Non-Blocking I/O



```
const getUserSync = require('./src/getUserSync')

const userOne = getUserSync(1)
console.log(userOne)

const userTwo = getUserSync(2)
console.log(userTwo)

const sum = 1 + 33
console.log(sum)


```

```
const getUser = require('./src/getUser')

getUser(1, (user) => {
    console.log(user)
})

getUser(2, (user) => {
    console.log(user)
})

const sum = 1 + 33
console.log(sum)
```

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

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

Your First Node.js Script

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

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

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

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

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

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

Section 2

NODE.JS MODULE SYSTEM

The Node.js Module System

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

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

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

We will learn more about this module system.

Importing Node.js Core Modules

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

notes-app > app.js:

```
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:

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

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

```
:~…/notes-app$ node app.js
```