**Node.js Notes**

**Introduction**

* JavaScript is normally confined to run in a browser and we’ve never used to be able to run JavaScript directly on a server or computer.
* Background context:
  + Computer can only understand machine code (which is like binary)
  + Assembly language is a bit higher level than machine code which makes it a bit easier to read/write code. Assembly is then assembled down into machine code.
  + Programming languages like C++ are higher level than machine code which makes it a lot easier to read/write code. The C++ code is then compiled down to assembly, then to machine code.
  + Js is a language which is abstracted even more away from machine code than C++ to make programming easier. But computers cannot directly understand JavaScript or compile it down to machine code so we cannot directly run JavaScript on a computer.
* While Js cannot run on a server, it can run on the browser. Running inside the Google chrome browser is an engine called the v8 engine. This v8 engine is written in C++ and it compiles JS into machine code at runtime. Thus, by passing js through the v8 engine, the browser can understand JS. Since there is no v8 engine outside the browser, js cannot run outside the browser which is a problem.
* Node is an environment that allows us to run JavaScript on the server side.
* Node is a program written in C++ that has Chrome’s v8 engine inside it.
* Graphical user interface, application

  Description automatically generated
* Thus, node takes our JS code, runs it through the v8 engine (since node has a v8 engine inside it), compiles our JS into machine code, allowing us to directly run js on our computer.
* Not only does node allow us to run js on the server side, it adds additional functionality such as reading/writing to files on a computer, connecting to a database, and acting as a server. While we get these additional features, we lose js features such as the DOM (but this is fine since node is meant to run js on the backend which doesn’t really need DOM manipulation) and the window object.

**The Role of Node js**

* The role of node in a website is to run JavaScript on the back end. Thus, we're going to be handling requests coming in from a browser.
* For example, a user might visit your website in a browser. That browser will make a request to the server, the node server will run some js to react to the request and it might communicate with a database or files or another server, then its going to formulate some respons enad send it back to the browser. That response could be an HTML page with data embedded inside it, CSS files, images, etc. Thus, node is an alternative to other server side languages like python.

**Why learn Node**

* Since we are already familiar with js from the frontend, we don’t need to learn a new server-side language as we can use js on the backend via node.
* Node is based on version. Unlike the browser apps which depend on the user’s browser, our node app depends only on the node.js version it was built on. Thus, when we are finished building our node app, we won’t worry about it being outdated or no longer supported by browsers since browsers can’t affect the node app.
* Node has a massive community behind it with lots of third-party packages and tools to help

**Installing Node js**

* To check if node is already installed, open the cmd and run ‘node -v’. If node is installed, it’ll return the version number.
* 
* If you don’t get a version number or you want to update your version of node, go to <https://nodejs.org/en/download/> and download node (LTS version). Then close the terminal, restart it, and rerun ‘node -v’.

**Running Node**

* We have two options to run code written via node, a repl (read eval print loop) or a cli executable
* Repl is for playing around and cli is for everything else
* Repl
  + Go to cmd and run ‘node’
  + We can write code in her as shown below
  + Text

    Description automatically generated
  + To exit, click Ctrl+C
* To actually run node,
  + Create a folder which will be our project folder.
  + Open the folder in vs code
  + Create a new js file inside the folder and write your code there
  + Then we run ‘node fileName’ or ‘node fileName.js’ in our terminal. Note that we can actually press the up arrow to go to our previous terminal command.
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + Note that whenever we make a change to our code, we have to rerun the file in our terminal.

**Global Object**

* In node, there are global variables which are variables that can be accessed anywhere in our application. (much like the window object in the DOM with browser side js that allows us to have access to many functions such as querySelector)(note that node doesn’t have the window object since node runs on the server, not browser)
* An example of a global variable is the global object.
* Graphical user interface, text, application, chat or text message

  Description automatically generated
* If we want to use the setTimeout function (a property of the global object), we could say global.setTimeout, but we could also just type setTimeout since it is implied we are calling a property from the global object.
* 
* There are other global variables as shown below (there are more global variables than what is listed, but it is not necessary to know them all)
* Text

  Description automatically generated

Text

Description automatically generated

* As of now, we are running node on our machine. But when we deploy on Heroku for example, process is useful since as of Heroku is a different environment from our local machine.

**Modules**

* Modules allow us to split our code up into different files, so we have smaller files with more structure. As well modules allow us to encapsulate code (hide information) so information from one file is not accessible from another file. Every file in node is a module by default.
* To export/import between modules, we will have to use the module object which is a global variable.
* Text

  Description automatically generated
* Notice the exports property in the module object which lists the different things a file exports. In the above example, our app module doesn’t export anything since the exports property of the module object is an empty object.
* To export, we have to modify the module.exports property as shown below in a new name.js module:
* Text

  Description automatically generated
* Notice that the exports property is no longer empty.
* To import modules, we use the require function. This require function’s parameter is the path of where the module we want to import is located. This require function will return the value associated with the ‘export’s property of the module object of the module we are importing from.
* Also note that when we import a module, we also execute the imported module.
* Ex:
  + So if we have the following two files and I run the app module, the output would be the following:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + In the above example, we run the app module. The first line in the app tells us to import the names module.
  + In the names module, we created grant and peter constants. Then we log out ‘hi’ since recall that when we import a module, we also execute it. Then we add those constants to the ‘exports’ property of the module. Notice that when we add these constants, the key is the constant name (like grant) and the value is the value of that constant (like ‘grant23’). We then log out the { grant: 'grant23', peter: 'peter23' } which is the value of the ‘exports’ property in the module object. Now, we know that the names module exports the object { grant: 'grant23', peter: 'peter23' }.
  + Back in the app module, we imported the names module via the require function. This require function returns the value of module.exports in the imported names module. We store that object in a constant called names and log it. Notice it logs out { grant: 'grant23', peter: 'peter23' } which is the value of module.exports in the imported names module. We then destructure the object and print out the ‘grant23’ and ‘peter23’.
* Note that the module object is specific to each module so there is no name conflict.
* Ex:
  + Even though the other and names modules both have export objects with a key called grant, there is no name conflict.
  + A screenshot of a computer

    Description automatically generated with medium confidence
* Default Exports
  + Graphical user interface, application, Teams

    Description automatically generated
  + Text

    Description automatically generated
  + Notice in the above, we are exporting a string instead of an object.
* Alternative syntax:
  + Graphical user interface, text, application

    Description automatically generated
  + 

**Built In Modules**

* Node has many more built in modules such as the following:
  + Os
  + Path
  + Fs
  + Http
* For more information about the modules, go to: <https://nodejs.org/en/docs/>
* We do not have to install built-in modules.
* To use these modules, we will have to import it via require.
* There are many properties and features for each module, but we will not go over them all.

**OS Module**

* The os module provides properties and methods for interacting with the operating system and the server
* Text

  Description automatically generated
* Text

  Description automatically generated

**Path Module**

* The path module provides properties and methods for interacting with paths
* Text

  Description automatically generated
* Graphical user interface, text

  Description automatically generated with medium confidence

**File System Module**

* The path module provides properties and methods for interacting with files
* This module’s methods come in both a synchronous and asynchronous version
* Synchronous file system methods
  + Reading files
  + To read a file synchronously, we can use the readFileSync. This function takes in two parameters. The first is the relative path to the file we want to read from. The second parameter is optional and is the encoding (generally we will go with utf8).
  + Let’s make two text files as shown below:
  + Graphical user interface

    Description automatically generated
  + Now if we run the following code, we get the following output in the terminal.
  + Text

    Description automatically generated
  + Writing to files
  + To write a file synchronously, we can use the writeFileSync. This function takes in two parameters. The first is a relative path to the file we want to write to. The second parameter is optional and is the value we want to write to the file with.
  + If the file we are writing to does not exist, a new file will be created. If the file we are writing to does exist, its content will be replaced by whatever new content we pass to the writeFileSync.
  + Suppose we have the following code (continuation from the previous example):
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + When we run it, we get the following:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + Notice a new third.txt file was created. Additionally, inside the file is ‘hi from firsthi from second’ since that was the value of first + second in app.js
  + Note that if we run app.js one more time, there is no change to third.txt since the writeFileSync function replaces the existing content, it does not append to the existing content.
  + If we want do not want to replace the existing content, but rather append to the existing content, we pass a third parameter with a value of {flag : ‘a’} (this third parameter is optional).
  + Suppose we have the following code and suppose third.txt is empty:
  + Text

    Description automatically generated
  + When we run the code, we get the following:
  + Graphical user interface, text, application

    Description automatically generated
  + Since third.txt was originally empty, writeFileSync with the {flag : ‘a’} parameter has the same result as if the {flag : ‘a’} parameter was removed.
  + When we run the code again, we get the following:
  + Graphical user interface, text

    Description automatically generated
  + Notice that we appended the new text to the end of the existing text in third.txt and not replace the existing text.
* Asynchronous file system methods
  + It’s better to use asynchronous methods so there is no code-blocking.
  + Reading from a file
  + To read a file asynchronously, we can use the asynchronous readFile method. This function takes in three parameters. The first is the relative path to the file we want to read from. The second is optional and is the encoding (generally we will go with utf8). The third parameter is a callback function that is executed when we have finished reading the file. This callback function takes in two parameters, the first being an error when reading the file and the second being the data that was read from the file. If the file was successfully read, the value of the first parameter (the error) will be null and the value of the second parameter (the data) will be the data that was read from the file. If reading the file was unsuccessful, the value of the first parameter (the error) will be error and the value of the second parameter (the data) will be undefined.
  + Suppose we have the following code:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + Graphical user interface

    Description automatically generated
  + When we run the app.js file, we get the following output:
  + 
  + Writing to files
  + To write a file asynchronously, we can use the asynchronous writeFile method. This function takes in four parameters. The first is a relative path to the file we want to write to. The second parameter is the value we want to write to the file with. The third parameter is optional and we can add the {flag : ‘a’} object to indicate to append to the end of the file instead of replacing the existing file. The fourth parameter is a callback function that is executed when we finish writing to the file. This callback function takes in two parameters, the first being an error when reading the file and the second being the data that was read from the file. If the file was successfully written to, the value of the first parameter (the error) will be null and the value of the second parameter (the data) will be undefined. If writing to the file was unsuccessful, the value of the first parameter (the error) will be error and the value of the second parameter (the data) will be undefined.
  + If the file we are writing to does not exist, a new file will be created.
  + Suppose we have the following code (and an empty third.txt file):
  + Text

    Description automatically generated
  + When we run the code, the output will be:
  + Graphical user interface, text

    Description automatically generated
  + When we run the code again, the output will be:
  + Text

    Description automatically generated
  + Notice that the content is getting appended to third.txt
  + Asynchronous Nature
  + Because readFile and writeFile are both asynchronous, their callback functions are pushed into the task queue in the event loop.
  + Ex:
  + Text

    Description automatically generated
  + When we run this code, we get the following output:
  + Text

    Description automatically generated with medium confidence
  + Note that ‘hi’ is immediately printed since logging ‘hi’ is synchronous, then there is a massive pause after (because of the for loop). After the for loop is done executing, either ‘wrote to file undefined read file hi from first’ or ‘read file hi from first wrote to file undefined’ is printed, depending on which of readFile and writeFile finished first.

**Http Module**

* IP addresses and domains
  + IP addresses are a series of numbers that act as an address for any computer connected to the internet
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + Some computers host websites and they will also have an IP address to identify it.
  + If we want to connect to the server on the host computer, we could type the server’s IP address into the browser
  + However, instead of typing the IP address of the server, we type its domain name like ‘gmail’. When we enter a domain name in the browser (the browser is the client), the browser will look up the IP address associated with the domain and use that IP address to connect to the server. The browser will send a request to the server that runs that website. The server looks at that request to determine what to send back to the browser (usually html).
  + The communication between the browser and server is via HTTP (hyper-text transfer protocol) which is a set of instructions that dictate how communication occurs. This is analogous to human communication where two people will both speak English to communicate with each other. Clients and servers will use HTTP to talk to each other.
* The http module allows node to transfer data over the HTTP.
* To create a server, we use the createServer method from the http module. This function returns an instance of the server that we can store in a constant if we need. The createServer takes in a parameter which is a callback function which gets executed every time a request comes into our server. This callback function parameter takes in two arguments. The first is the request object and the second is the response object. The request object contains information about the request. The response object is what we will use to send a response back to the client. We will explore this callback parameter function below in more depth.
* If we just create a server and do nothing else, the server is not actively listening for requests.
* To make the server actively listen for requests, the listen method will be invoked. The listen method is a function that we can access from an instance of a server. This listen method takes in three arguments. The first argument is the port number that the server will listen to (generally port 3000 will be used). The second argument is optional and is the hostname (default value is ‘localhost’). The third argument is optional and is callback function that is executed when the server starts listening.
* Text

  Description automatically generated
* Before we run the code, we will go explain localhost and port numbers.
* Localhost and port numbers
  + Localhost is like a domain name on the web such as ‘facebook’
  + This domain name takes us to a loopback IP address which is 127.0.0.1 which points to our own local computer.
  + When we connect to the localhost domain in our browser, the browser will connect to the computer with an IP address of 127.0.0.1 which is our own computer so the browser will connect back to our computer. Our own computer acts as a host for our website.
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + A port number represents a specific channel gateway/port on our computer that our server should communicate through.
  + Port numbers are like different doors into your computer through which internet communications can be made to different applications.
  + For example, on your computer, you've probably got a lot of different applications that connects to the internet and receives and sends data. For example: Skype, discord, mail, etc. These applications will connect to the internet via different port numbers to keep information separate from one another.
  + Our nodejs server will also need its own port number to communicate through. While our nodejs port number is arbitrary (as long as its not being used by some other app), 3000 is a convention for local web development.
  + Thus, when we use localhost, we also type the port number after it to indicate the port we are using. Ex: localhost:3000
* We can now continue from where we left off (right before the localhost and port numbers section) and run the app.js file.
* Output: A screenshot of a computer

  Description automatically generated with medium confidence
* Since we logged ‘started listening for requests on port 3000’, the server is running and actively listening for requests to localhost:3000
* Also notice that the terminal program does not terminate since the server is ongoing (we haven’t told the server to shut down).
* Now, if we open our browser and go to <http://localhost:3000/>, the webpage doesn’t load anything. The loading icon is spinning since we are waiting for a response from our server, but we never actually sent back a response.
* While the webpage doesn’t load, we do get the following response in the terminal:
* Text

  Description automatically generated
* This is because our callback function to the createServer method got executed since a request was made when we went to <http://localhost:3000/>. Executing the callback function logged ‘request made’, confirming a request was made.
* Also note that nothing is logged to the browser console since when we are logging to the server console, not the browser console.
* Request object
  + Recall that the callback parameter function to the createServer method took in two parameters, the first of which was the request object. This request object contains information about the request.
  + We could log out the request object in callback parameter function and it would log a massive object.
  + Some notable methods within this request object include the ‘url’, and ‘method’ methods.
  + To get the url path after the domain name and port number that the client requested for, we can use the ‘url’ method.
  + To get the method of that request (GET, POST, PUT DELETE, ETC), we can use the ‘method’ method.
  + Ex:
  + Suppose we have the following code and we run the app js file and then we enter <http://localhost:3000/about> in google chrome.
  + Text

    Description automatically generated
  + We would get the following output:
  + Text

    Description automatically generated
  + Notice that the request method was a GET request. Notice that the url path after the domain name and port number that the client requested for is ‘/about’.
* Response object
  + Recall that the callback parameter function to the createServer method took in two parameters, the second of which was the response object. This response object is what we use to send a response to the client.
  + We could log out the response object in callback parameter function and it would log a massive object.
  + Some notable methods within this reponse object include the ‘setHeader’, ‘write’, and ‘end’ method.
  + When we provide a response to client, we need to include response headers. Response headers let the client have more information about what kind of response the server is responding with. For example, is the data the server sends back text, HTML, JSON, etc?. We can also use the response headers to set cookies.
  + To include response headers in our response to the client, we can use the setHeader method. There are many different types of response headers. For instance, we could set the Content-Type header to be one of html or text or json, etc.
  + To include data within our response to the client, we can use the write method.
  + To end the response and send it back to the client, we can use the end method.
  + Ex:
    - Suppose we have the following code and then we go to <http://localhost:3000/>:
    - Text

      Description automatically generated
    - The server terminal output would be:
    - A screenshot of a computer

      Description automatically generated with low confidence
    - The webpage output would be:
    - Graphical user interface, text, application, chat or text message

      Description automatically generated
    - We can inspect the webpage and go to the network tab and see the request the client made.
    - Graphical user interface, text, application

      Description automatically generated
    - We can then click on it and under the headers tab and in the response headers tab, we can see the content-type response header that we specified.
    - Graphical user interface, text, application, email

      Description automatically generated
    - Note there were other response headers that are set by default, and in fact there are other requests to our server, but those can be ignored for now.
  + Ex:
    - Suppose we have the following code and we go to <http://localhost:3000/about>:
    - Text

      Description automatically generated
    - The server terminal output would be:
    - A screenshot of a computer

      Description automatically generated with low confidence
    - The webpage output would be:
    - Text

      Description automatically generated with medium confidence Text

      Description automatically generated
    - When we go to <http://localhost:3000/about>, the request path/url is ‘/about’. Thus, it satisfies the condition in line 4. Inside the if statement, we log out ‘GET request made from /about’ since req.method is a GET request mthod and the req.url is ‘/about’.
    - Then, we prepare to send back a response to the client. We use the setHeader method to indicate that the type of data in the response will be html. Then, we use the write method to include html elements as the data within our response. Then, we use the end method to indicate we are done with our response and that we should send the response back to the client. Notice that we can also add data to our response by passing the data as a parameter to the end method (much like the write method).
    - When the client receives the response, the above html page is loaded. Notice that the default html structure and head tags are automatically generated. We could write our own head tag and it would replace the head tags that were automatically generated.
* Returning HTML Pages
  + In the above section, we were able to send back html within the createServer callback function parameter. But writing html like that is not scalable. Instead, our html pages should be created in a separate file and node should read those files and send the html within those files to the client.
  + To read those files, we need to import the file system module.
  + Ex:
  + Suppose we have the following files:
  + Graphical user interface, application, website

    Description automatically generated
  + Text

    Description automatically generated
  + Text

    Description automatically generated
  + When we run this file and we go to <http://localhost:3000/about>, we get the following
  + Terminal output: Text

    Description automatically generated
  + Webpage output: Graphical user interface, text, application

    Description automatically generated
* Routing
  + Suppose we want to return different html pages based on the path the client requested. For example, if they went to ‘/store’, we render the store.html page.
  + The code would look like the following:
  + Text

    Description automatically generated(the html pages are just the default template with the file name in the body)
  + Text

    Description automatically generated
  + When we run the code and go to <http://localhost:3000/>, we get the following:
  + Text

    Description automatically generated
  + Graphical user interface, text, application, email

    Description automatically generated
  + If we then go to <http://localhost:3000/store>, we get the following:
  + Text

    Description automatically generated
  + Graphical user interface, text, application, email

    Description automatically generated
  + Note that ‘GET request made from /favicon.ico’ is a request that browsers will send by default in order to get the website icon, this can be ignored as its not that important.
* Status Codes
  + Status codes describe the type of response to the browser
  + Some of the common status codes are shown below:
  + Graphical user interface, text, application

    Description automatically generated
  + There are many more status codes as show below:
  + Graphical user interface, text, application

    Description automatically generated
  + To add a status code to our response, we modify the ‘statusCode’ property of the response object.
  + Continuing with the example from the previous section on routing, we can add status codes as shown below inside the switch statement:
  + Text

    Description automatically generated
  + Now, if we run our app.js file and go the <http://localhost:3000/store> for example, we get the following:
  + Table

    Description automatically generated
  + Notice that the status code is 200
  + Now, if we go to some random url like <http://localhost:3000/asd> for example, we get the following:
  + Calendar

    Description automatically generated
  + Notice that status code is 404.
* Redirect
  + Continuing from the example in the above section (status codes), suppose we want to create a new html called store2.html to be the new store. So if anyone goes to ‘/store2’, we respond with the store2.html file. However, we also want it so that when anyone goes to ‘/store’, we redirect them to the new store2 path.
  + To redirect, we want to modify the location header of the response to be the ‘/store’. Recall we modify response headers via setHeader.
  + Thus, the code would look like the following:
  + Text

    Description automatically generated
  + Now, when we run the app.js file and try to go to <http://localhost:3000/store>, we get redirect to <http://localhost:3000/store2>
  + Notice that we should change the status code to 301 which indicates the ‘/store’ page has been moved to ‘/store2’. Note that we didn’t modify the path variable inside the “/store” case. Also notice we had res.end() to end the current response.

**Npm**

* Npm allows us to use pre-built code from other people instead of having to build everything from scratch.
* Stands for node package manager and is the main package manager for node js
* A package contains code that someone created and uploaded to the npm online platform. (package, dependencies, modules are used interchangeably)
* Npm is an online platform and a command line interface (CLI).
* Graphical user interface, website

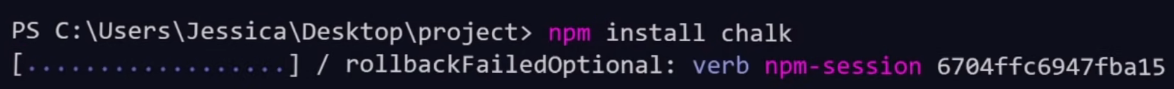
  Description automatically generated
* In the online platform, anyone can publish and share tools written in JS. These tools can be used in the browser, server, or command line.
* In the command line tool, it allows us to interact with the online platform. Via the command line tool, we can:
  + install/uninstall packages
  + use version management. Every package has a version which changes as the package changes. Npm allows us to be up to date with the version, or switch to a specific version.
  + Use dependency management. Many packages are built using other packages. Dependencies are packages that are need for another package. Npm allows us to install a package and all of its dependencies.

Installing npm

* We need to first install node js Graphical user interface, text, application, chat or text message, website

  Description automatically generated
* We can check the node and npm versions using ‘node -v’ and ‘npm -v’ respectively.
* Text

  Description automatically generated
* When we install npm packages, we can install them locally or globally.
* Graphical user interface

  Description automatically generated
* Local packages are installed only in the project folder we want to use them in. To install a package locally, we run in cmd: npm install packageName
* 
* Global packages can be used from any folder. To install a package globally, we run in cmd: npm install -g packageName
* 

Package.json

* If we are using multiple packages in our project, we can keep track of which projects we have installed via the package.json file
* A picture containing text, sign

  Description automatically generated
* This file can also store information about the project such as its name. version, author, etc
* If we want to clone a project and it has a package.json file, we can run npm install and npm will read the package.json file and install all of the packages listed in the package.json file.
* We can create a package.json file by running in cmd: npm init
* When creating this file, npm will ask us some questions about the project.
* Text

  Description automatically generated
* If we don’t want to answer the questions, we can run in cmd: ‘npm init -y’ which sets all the fields to a default value.
* When we install a package, the package.json file gets updated to include the new package we installed as a dependency.
* Graphical user interface, text

  Description automatically generated
* As well, a new folder called node\_modules is created. This node\_modules folder is where the package files are installed. We will see not only the package we installed, but other packages as well since those are the packages that our package depends on. And each of those other packages depend on other packages which are also added to the node\_modules folder
* Text

  Description automatically generated
* We can run npm list in the project folder to display all the installed packages and what versions they are on.
* Text

  Description automatically generated

Semantic Versioning

* Packages can get updated over time to add new features/fix bugs, but this can lead to conflicts between packages that the updated package depends on, or packages that depend on the updated package.
* If we type in cmd: ‘npm view packageName versions’, we get a list of all the versions of the package from the beginning
* Text

  Description automatically generated
* The versions are represented by 3 numbers, separated by a ‘.’
* The first, second, and third numbers are the major, minor, and patch versions of the package
* A picture containing text, clock, first-aid kit

  Description automatically generated
* Major updates are big and not backwards-compatible which means updating will probably break old code.
* Minor updates are backwards-compatible features and ideally won’t break old code
* Patch updates are small bug fixed that won’t break older builds
* When a new version is released, the numbers on the right reset back to 0. For example, if we are on version 1.9.15 and there is a major update, the new version becomes 2.0.0. If we are on version 1.9.15 and there is a minor update, the new version becomes 1.10.0.
* When we first locally install a package, npm will install the newest public release. When we upgrade, npm has safeguards that help prevent code from being broken.
* In the package.json file, the dependencies object will display the dependency and it’s version such as below: Graphical user interface, application, website

  Description automatically generated
* Notice the ‘^’ symbol which tells npm to update the package to the latest minor and patch version for the currently install major version. So, the above version of 4.1.1 might update to 4.2.1 for example even if version 5.0.0 exists
* We can run in cmd: ‘npm update’ to update all the packages listed to the latest minor and patch version for the currently installed major version (as shown in the bullet point above). Thus, if version 5.0.0 is released and are on version 4.1.1, we can run ‘npm’ install but nothing happens since it is a different major version.
* To override this safeguard and update to the newest version (potentially different major version), we can run in cmd: ‘npm install package\_name@latest’
* 
* To install an older/newer version of a package, we can run in cmd: ‘npm install package\_name@versionNumber’.
* 
* We can also limit updates to patches within the current minor version by running in cmd: ‘npm install [packageName@~major.minor](mailto:packageName@~major.minor)’ . So if we had version 2.2.0 installed and wanted to update to the latest patch within 2.2, we could run in cmd:
* 

Package-lock.json

* When using a package.json file from a different environment, compatibility issues can arise.
* For example, if we copy a github repo that has a package.json file and we run npm install, it is possible that a dependent package has been updated between the time the package.json file was created by the author and when we run npm install. Thus, we could be installing a different minor and patch version that was used in the original repo, which could lead to breaking of code.
* Package-lock.json solves this problem.
* If we run npm install and there is a package-lock.json file, npm will install the exact major and minor patches recorded in the package-lock.json file. This will ensure our package versions match those of the authors.

Devdependency

* A dev dependency is a dependency that the developer uses to make their lives easier when writing code but isn’t needed for the production version of their application. For example, testing packages could also be in dev dependency instead of normal dependency.
* To install packages as a dev dependency, we run ‘npm install packageName -D’.

Uninstall

* To uninstall a package, run ‘npm uninstall packageName’

**Nodemon**

* Nodemon is an npm package that will automatically restart the server when changes in our files are made.
* To use it, we can install it as a dev dependency since we don’t need nodemon in production. When we push our node server to Heroku or whatever, some more serious package will be used to restart the application.
* Thus, run ‘npm i -g nodemon -D’ in the terminal to globally install it. (make sure we already ran npm init)
* 
* To run nodemon, in the cmd run ‘nodemon fileName’.
* If that doesn’t work and we get the following error, go to this location File C:\Users\Dell\AppData\Roaming\npm and delete the nodemon.ps1 file and run the command.
* Text

  Description automatically generated
* The console should now log the following and restart upon changes:
* Text

  Description automatically generated

**Event Loop**

* The node js event loop is similar to the client side js event loop
* Some differences are that node has extra features (such as ‘fs’ module operations and process.nextTick) so node will have to consider these additional features in the event loop.
* The node js event loop uses libuv which is an open source library written in C++ that helps with asynchronous i/o. It gives node access to the underlying computer os, file system, networking, etc. As well, libuv also implements two extremely important features of node, the event loop and the thread pool. The event loop handles easy tasks while the thread pool is for more heavy work like file access.
* Graphical user interface, application

  Description automatically generated
* The event loop is what allows Node.js to perform non-blocking I/O operations – despite the fact that js is single-threaded – by offloading operations to the system kernel whenever possible.
* Any callback functions will be executed in the event loop.
* In order are these callback functions executed? The event loop will explain that.
* Diagram

  Description automatically generated
* When we start our node application, the event loop starts running right away
* The event loop has 4 phases:
  + Expired Timer Callbacks
    - Callbacks from functions such as setTimeout and setInterval are pushed into this queue
  + I/O Callbacks
    - Callbacks from async ‘fs’ and ‘http’ functions such as readFile and writeFile are pushed into this queue.
  + Immediate Queue Callbacks
    - Callbacks from the setImmediate function are pushed into this queue.
  + Close Handlers Callbacks
    - Callbacks for close events (such socket.on(‘close’, …)) are pushed into this queue.
  + (note there are actually 1 or 2 more phases that are internally used by node, but aren’t that important)
* Each phase has its own callback queue which gets filled with callbacks that come from the events that the event loop receives. For example, callbacks from functions such as setTimeout and setInterval are pushed into the Expired Timer queue. Callbacks from functions such as readFile and writeFile are pushed into the I/O queue.
* These callback functions are processed one by one. When there are no more callbacks left in the queue, the event loop will enter the next phase.
* If a callback is added to the callback queue in its associated phase, but the event loop is in another phase, then the callback will only get execute when the event loop comes back to its associated phase. For example, if a timer expires while the event loop is in the I/O phase, then the callback for that timer will only get executed when the event loop comes back to the Expired Timer phase.
* There are also two other queues, the process.nextTick callback queue and the microtasks queue. If there are callbacks in one of these two queues, they will be executed right after the current function execution instead of waiting for the entire loop to finish or instead of waiting for the current phase to complete. The process.nextTick callback queue will get executed first then the microtask queue.
* A tick is once cycle in this event loop. When we complete one tick, the event loop needs to decide if the program should exit or go on to the next tick. If there are any pending timers or I/O tasks, the event loop will go on to the next tick. If not, we exit the program.
* process.nextTick()
  + When we pass a function to process.nextTick(), we instruct the engine to invoke this function at the end of the current operation, before the next event loop tick starts:
  + Text

    Description automatically generated
  + When this current phase of the event loop ends, the JS engine runs all the functions passed to nextTick as callbacks
  + It's the way we can tell the JS engine to process a function asynchronously (after the current function), but as soon as possible, not queue it.
* setImmediate()
  + Text

    Description automatically generated
  + It’s callbacks are added to the Immediates callback queue.
* Ex:
  + Text

    Description automatically generated
  + Output: Text

    Description automatically generated
  + The event loop is useful because it enables us to make time consuming functions to be asynchronous. As a result, the synchronous code is not blocked.
  + For example, suppose we have a website with many users currently logged in. If one of the users clicks a button which triggers a long call to the database and we do not make this database call asynchronous, then the call would block the server and all the other users would not be able to use the website. Note that it is not the call to the database that is avoidable (since we still need to load the data for that user), but it is waiting for that database call that is avoidable.
  + A code example of the above could be the following:
  + Text

    Description automatically generated
  + Now suppose we have 2 users, user 1 and user 2. Suppose user1 enters <http://localhost:3000/about> in their browser and right after, user 2 enters <http://localhost:3000/> in their browser. Since user 1’s request was received first, we handle user 1’s request first. User 1’s url path is indeed equal to ‘/about’ which causes some blocking code to occur. This blocking code blocks the entire server for all users, not just user 1. Thus, even though user 2 sends a request right after user 1, the server cannot handle user 2’s request. Thus, while the blocking code is running, both user 1 and user 2 just have the spinning loading icon for their website. When the blocking code is done, the response is sent back to user 1 and ‘about page’ is loaded on their website. Immediately after responding to user 1, the request from user 2 is responded to and now user 2 sees ‘home page’ on their website.
  + TLDR: Blocking code blocks the entire server for all users.

**Using Promises/Asynce Await to refactor asynchronous code**

* Suppose we wanted to read two text files, and write their combined content into a new file. We could write it as below:
* Text

  Description automatically generated
* But this code is hard to read as it has too many nested callbacks.
* To attempt to make it easier to read, we can use promises.
* Text

  Description automatically generated
* In the above code, we used a wrapper getText function that takes in a file path and return a promise. This promise rejects with the error if the path is unreadable. This promise resolves with the content within the file if the path is readable. In doing so, the readFile method is easier to handle. A similar wrapper writeText function is created to make the writeFile method easier to handle.
* By using this getText and writeText function that returns a promise, we can chain .then and .catch methods as shown above, but this code is still hard to read as there is too many nested .then and .catch.
* To attempt to make it easier to read, we can use async await.
* Text

  Description automatically generated
* Notice the use of the readThenWrite async function. Notice how much cleaner to code inside it. This function is now very readable.
* However, the writeText and getText wrapper functions are still unreadable.
* To attempt to make it easier to read, we can use import the utils module with provide us a method called promisify. This promisify method allows us to take a function as an input that follows the common Node. js callback style, i.e., with a (err, value) and returns an equivalent promise version of the callback instead of a callback. Thus, we can pass in the readFile and writeFile methods as the parameters to this promisify method and get new functions as shown below.
* Text

  Description automatically generated
* Notice how much cleaner the code is. The readFilePromise function replaced the getText wrapper function and the writeFilePromise function replaced the writeFilePromise wrapper function.
* In fact, we can make the code even cleaner since we can access the ‘promises’ property of the required ‘fs’ module. This ‘promise’ property is an object that includes many methods. For example it provides readFile and writeFile methods. Note that the readFile and writeFile method that we got from the ‘promises’ property of ‘fs’ are not the same as the readFile and writeFile methods from ‘fs’ directly. The readFile and writeFile method that we got from the ‘promises’ property of ‘fs’ return promises instead of callbacks (much like util.promisify(readFile) and util.promisify(writeFile)).
* Using the ‘promises’ property of ‘fs’, our code can now look like the following:
* Text

  Description automatically generated
* Notice that this code is extremely readable and clean.

**Events**

* In browser js, the website listens for ‘onclick’ events when a user clicks a button.
* A similar philosophy appears in node.
* Event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions, sensor outputs, or message passing from other programs or threads
* Event-driven programming is heavily used in node as many built-in modules in node use events under the hood
* To use events, we will require the events module which returns a class that we can store in a constant
* If we want to create custom events, we need to create a new instance of the class that the events module returns.
* This EventEmitter instance contains many methods, but the two main methods are ‘emit’ and ‘on’.
* Emit is a function whose first parameter is an event. The emit function will emit this event. We can add additional parameters after the first parameter which will be supplied to the callback function in ‘on’
* On is a function that has two parameters. The first is an event. The second is a callback function. Whenever an event is emitted, the callback function will be executed. This callback function can take in parameters and these parameters are supplied from the extra parameters in the emit function.
* Ex:
  + suppose we have the following code then we go to <http://localhost:3000/>
  + Text

    Description automatically generated
  + Output: Text

    Description automatically generated
  + Notice in line 2, we require the events module which returns a class that we stored in a constant called EventEmitter.
  + In line 3, we create an instance of the EventEmitter class and store that instance object in an object called customEmitter.
  + In lines 5 to 9, we create a server that will log to the console the request url (line 6), emit a ‘myEvent’ event with the request url as a parameter to .on callback functions (line 7), then send a response back to the client.
  + The server runs before the client is able to enter <http://localhost:3000/> in their browser so no emit is emitted just yet.
  + In lines 11 – 15, we use the ‘on’ function to listen to the ‘myEvent’ event.
  + In lines 17, we use the ‘on’ function to listen to the ‘myevent’ event. Note that ‘myEvent’ and ‘myevent’ are two different events.
  + In line 19, we emit the ‘myEvent’ event. Because of that, the callback functions on line 11, 13, and 15 are executed. These callback functions are executed in the same order that their ‘on’ methods were created. Thus, the callback function on line 11 is executed first, logging ‘on 1’. Then, the callback function on line 13 is executed. This callback function takes in one parameter called text. Since the ‘myEvent’ emitted on line 19 did not include parameters, the value of the text parameter is undefined (that’s the default parameter value). Thus, the text ‘on 2 undefined’ is logged. Then, the callback function on line 15 is executed. Similar to the callback function on line 13, the line 15’s callback function’s parameters of text1 and text2 both have a value of undefined, hence logging ‘on 3 undefined’.
  + Notice that the callback function on like 17 is not executed since the ‘myevent’ event was never emitted, rather the ‘myEvent’ event was emitted. Also notice that the callback function on like 21 is not executed since it hasn’t been reached yet.
  + Then we get to line 21, we use the ‘on’ function to listen to the ‘myEvent’ event.
  + On line 23, we listen to port 3000 on localhost.
  + The above happens pretty quickly so suppose only now does the user go to <http://localhost:3000/>. This sends a request which is received by the server. We log out ‘new request from /’ since the path of the above request is ‘/’. Then we emit the ‘myEvent’ event along with ‘/’ (the value of req.url) as a parameter. Thus, the callback functions on lines 11, 13, 15, and 19 are executed with ‘/’ passed as a parameter to those callback functions. This would result in ‘on 1’, ‘on 2 /’, ‘on 3 undefined’, and ‘on 5’ being logged to the console.
  + When we go to <http://localhost:3000/>, we actually make two requests since the browser also makes a request for favicon.ico by default. This results in the following being logged since we emit the ‘myEvent’ event along with ‘/favicon.ico’ (the value of req.url) as a parameter:
  + Text

    Description automatically generated
* Ex:
  + We can create a wrapper function for our emit function. In our case, when we emit an event, we could enable a parameter to be passed in, that parameter will be logged, and then the parameter will be passed to the ‘on’ method’s callback function.
  + Text

    Description automatically generated
  + Output: 
* Note that instead of passing a callback function to the createServer http module method, we could do the following which is equivalent:
* Text

  Description automatically generated
* The ‘request’ event is a built-in event which gets emitted when a request to our server is made.
* The server now has the ‘on’ method attached to it, making our server listen to the built-in ‘request’ event.

**Streams and Buffers**

* If we try to read a large file, we have to wait for the file to completely finish being read before we can do something with the data inside the file.
* To overcome this problem of waiting for the file to completely finish being read, we can use streams
* Streams allow us to start using the data within a file before the file has fully been read.
* Analogy:
  + Suppose we have a water source. Suppose we also have an empty. We want to fill the pool with water from the water source.
  + Method 1: On way to fill up the pool would be the get a tank that is filled with the full amount of water from the water source. Then, we deliver the tank to the pool and empty all of the water from the tank at once to instantly fill the pool. This method forces us to wait a long time since we have to wait for the tank to be fully filled. This is similar to reading a file using readFile or readFileSync.
  + Method 2: Another way to fill up the pool would be to have a stream (like the water stream) that delivers water from the water source to the pool. This way, the pool is filled up a little bit at a time, but we can start using the pool with a bit of water in it almost straight away. This is similar to reading a file using streams.
* Thus, if we have a massive file we need to read, we could wait until all of it's been read and then do something with it. But that could take a long time, or we could pass the data a bit at a time through a stream and this way, small chunks of data are packaged up into what's known as a buffer and then sent down the stream every time the buffer fills up. Thus, every time we get a new chunk of data from the buffer we can start using it.
* A picture containing diagram

  Description automatically generated
* Streams are useful when loading videos since we don’t have to load the entire video, we just need to load the start and the user can start watching.
* There are 4 different types of streams: readable, writable, duplex, and transform. We will only deal with readable and writable streams.
* Readable Streams
  + Readable streams allow us to read data
  + To create a readable stream, we have to import the createReadStream method from the ‘fs’ module.
  + Then, we have to use the createReadStream method. This method takes two parameters. The first parameter is the path of a file we want to read. The second parameter is optional and is an object that allows us to modify the stream. By default, the size of the buffers is 64 KB. To change the size, we add the ‘highWaterMark’ property whose corresponding value is the number of bytes in a buffer. We can also add the ‘encoding’ property whose value will usually be ‘utf8’.
  + This createReadStream returns us a readable stream that we can store in a constant. This createReadStream also starts reading the file and starts filling the buffers with the read data.
  + Every time a buffer is filled with data, a ‘data’ event is emitted.
  + We then use the readable stream’s ‘.on’ function to listen to the ‘data’ event. Then .on method takes in a callback parameter function. This callback parameter function takes in one parameter which represents the data within the buffer that was just filled up.
  + We can also use the readable stream’s ‘.on’ function to listen to the ‘error’ event. Then .on method takes in a callback parameter function. This callback parameter function takes in one parameter which represents the error when reading the file (such as if the file path is invalid).
  + Ex:
  + Text

    Description automatically generated
  + Output: Background pattern

    Description automatically generated
  + Notice in the above output, we logged many ‘NEW CHUNK’s which represents each time a buffer was filled.
  + Also notice how the size of each buffer was around 64000 bytes or 64KB. This is because by default, the buffer sizes are 64 KB. Notice how the last buffer’s size is different from the other buffer sizes since that is the remainder.
  + However, the data within the buffer is not that useful as of now since we are reading text.
  + To solve this, we set the encoding in the createReadStream method to be ‘utf-8’
  + Ex:
  + Text

    Description automatically generated
  + Part of the output: A picture containing text

    Description automatically generated
  + Note that I didn’t print the entire output since it was too big.
  + We can also change the buffer sizes from the default of 64 KB to 500 000KB for demonstration purposes:
  + Ex:
  + Text

    Description automatically generated
  + Output:
  + Text

    Description automatically generated
  + Also notice that when we add up the sizes of all the buffers, it should be roughly equal to the size of the file we write to. The size of all the buffers is 499.95KB + 88.84KB = 588.79KB which is close to the file size of 576KB. 
* Writeable streams
  + Writeable streams allow us to write data
  + To create a writeable stream, we have to import the createWriteStream method from the ‘fs’ module.
  + Then, we have to use the createWriteStream method. This method takes 1 parameter which is the path of a file we want to write to (note that if the file does not exist, it will be created). This createWriteStream returns us a writeable stream that we can store in a constant.
  + To write to that file, we have to use the .write function (it is a method within the writeable stream). When we call the ‘write’ function, the original data within the file is removed. Then, every time we write to the file, the content we wrote is appended.
  + Ex: Suppose we have the following .js file and the following newfile.txt:
  + Text

    Description automatically generated
  + Text

    Description automatically generated
  + Then, when we run the .js file, we get the following:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + In the above code, we read a file using createReadStream and every time we get the data within a fully loaded buffer, we write that data to newfile.txt
  + Notice how the ‘existing content’ text was replaced.
* Pipe
  + A pipe is used when we want to pass data directly from a readable to a writeable stream.
  + Basically, it is a much shorter way of writing the below:
  + Text

    Description automatically generated
  + Instead of writing the above, we can now write the below which is equivalent:
  + Text

    Description automatically generated
  + Suppose we have a server and when a request is made, we read from a large text file, then send a response with the contents within the text file
  + We could write the below:
  + Text

    Description automatically generated
  + Output:
  + Graphical user interface

    Description automatically generated with medium confidence
  + Console network output:
  + Table

    Description automatically generated
  + Graphical user interface, text, application

    Description automatically generated
  + Notice how we probably shouldn’t send an entire 2.0MB response in one go.
  + A better way would be to write the above would be below:
  + Text

    Description automatically generated
  + Output:
  + Table

    Description automatically generated
  + Graphical user interface, application, Word

    Description automatically generated
  + Notice how this time, we send the 2.0MB in chunks as opposed to one go.