**Intro to Node, Events, and Streams**

***Agenda:***

*The what and why of Node.js*

*JS in Node vs in the browser*

*NPM & Node Modules*

*Servers in Node*

*Intro to Postman*

*Streams: What & Why?*

1. When we make a request from the front end, we receive data back from a server. It’s sometimes unclear what exactly we are receiving.
2. Node is an open-source cross-platform runtime environment built by Linux.
   1. It’s a way to run JS in the backend without our browser.
   2. **Runtime environment** is where our code is run.
   3. **“Node”** runs JS for us.
   4. Until now, our browser has been our runtime environment.
      1. Tictactoe, snake game, megamarkets, all have been executing code in our browser.
3. Built on V8, which is an engine for Chrome.
   1. This executes JS code.
   2. Because it’s built on V8, performance benefits from Google’s maintenance of V8 engine is passed onto Node.
4. Node can also make files, delete them, kill processes, shutdown the computer, system calls.
   1. It can do this because it is not **sandboxed** – isolated from the other environments.
   2. Something that is sandboxed only has access to what we pick and choose.
5. Benefits of Node:
   1. It’s high level
   2. It uses the same language as the browser, reducing our cognitive overhead
   3. It’s non-blocking (?)
      1. Our thread of execution is not blocked when it has to process other tasks.
      2. Great for I/O Bound application (input output)
         1. When we might have to send requests to an external database or 3rd party API.
   4. NPM!!!!
6. JS in the browser vs JS in Node
   1. **Big takeaway**  - the underlying implementation is different, but the behavior is relatively the same.
      1. ***JS in the Browser***
      2. **JS Ecosystem is made up of**:
         1. JS Runtime
         2. Event Loop
         3. Queues
      3. We can pass off time-consuming tasks to the Web APIs.
      4. The Web API pushes any delayed functionality onto the callback queue once the long task completes.
      5. The event loop is responsible for clearing the callback queue at the appropriate time.
   2. **JS in Node**
      1. It does **not** have the same event loop or callback.
      2. Instead, it relies on the libuv library to handle the event loop, callback queue, and also for background I/O operations.
      3. It seems to serve as an ersatz web API for JS to pass off lengthy/asynchronous tasks.

|  |  |  |
| --- | --- | --- |
| Application | Node.JS Bindings( Node API) | LibUV  Event queue; event loop; blocking operation; execute callback. |
| V8 |

* 1. Node has **no Web API**



* + 1. No window, no document, navigator,
    2. XMLHTTP Requests, or any other browser-specific object.
  1. Instead, it has its own system-related globals
     1. \_\_dirname
     2. \_\_filename
     3. Process
  2. Node is still built on an event loop, has setTimeout and console, but it’s just a different event loop with a different underlying implementation.
  3. Node modules & NPM
     1. In order to keep things flexible and lightweight, Node relies on the concept of modules for added functionality, depending on the use case.
     2. Separating functionality makes it scalable and easier to debug.
     3. Node modules help us avoid globals and keep our namespace clean
  4. Built in modules:
     1. http, fs (file system), path, events, child\_process, and many more.
     2. There are also modules we write ourselves
        1. this sounds a lot like writing helper function to help us with specific tasks
        2. Except we generalize it even further.
     3. As well as Node modules from the open-source ecosystem, which is why we use npm.
     4. If I want to build a web server, I might import node’s built-in http module.

*Examples:*

const http = require('http')

const fs = require('fs')

* + 1. “fs” module is used for working in the file system. In order to access its functionality, we need to “require” it.
    2. “Require” is kind of like an “import” keyword.
  1. For modules that we write ourselves:

|  |  |
| --- | --- |
| *Counter.js*  let i = 0;  function counter() {  console.log(i)  *return* i+=1;  }  module.exports = counter | *Index.js*  const counter = require('./counter.js')  console.log(counter()) *// 1*  console.log(counter()) *// 2*  console.log(i) *// undefined*  When we pull out the counter function, it forms a closed-over variable environment.  So, we are exporting the **function.**  We can also export multiple things from a file. |

* 1. “Require” –
     1. Node will check if it’s built-in module.
     2. If we are using something we created ourselves,
        1. We need to provide the file path to Node
        2. So that Node can parse through the file and search for the “module.exports” line.
     3. If we
  2. Open Source Modules
     1. To access these node modules, check out npmjs.com
     2. There are TONS of them.
        1. So we want to consider things like:
        2. Is it maintained? Is it appropriate?
        3. To check on these things, check out the repo for the module on github and see if it is being actively maintained.
        4. You should also check out the weekly downloads to see how active the userbase is.
        5. And the last published will show the frequency of updates.
     3. Dependencies
        1. Dependencies are other modules.
        2. Express has 31 dependencies –
           1. It requires 31 OTHER modules to run!
  3. Using a module
     1. We need a “package.json” file that contains the instructions and the dependencies that the project requires.
     2. “npm” looks for the package.json file to install everything.
     3. “npm init” sets up package.json for us.
        1. This takes us through a series of questions to populate the package.json file, which now contains some of the data we provided.
     4. Npm install will not work without first initializing the package.json.
     5. Once npm init is run and the package.json is initialized, we can **then**  run npm install.
        1. To specifiy what modules we need, we simply provide the module name: npm install moment
     6. And this will create a node\_modules folder in our file tree.
     7. And node will recursively go through each module, installing **all dependencies in the dependency tree of each module.**
     8. So even if express has 31 dependencies, it will actually install far more than 31 modules.

1. Npm install installs all packages listed in the package.json
   1. If a project does not have a package.json file, **create one with npm init.**
   2. **For your projects, be sure to add each package you use to your package.json**
      1. By using npm install <pkg name>
   3. Generate a new project using a package.json file with npm install.
2. Why create a server…?
   1. We often use node to set up a server (npm start)
   2. Some example/usecases for a server
   3. Save/fetch data from our own database
   4. Fetch data from other services (FB/Twit)
   5. Process large quantities/complex data outside the browser.
3. Servers – how do they work?
   1. Servers **listen for requests on a specified ports**
      1. they can also listen for specific **methods** and/or requests to specific **routes** (endpoints).
   2. Depending on the **method and route** specified, a server may or may not process the request in some way.
   3. In the end, the server **will respond** to the request with a **status** and, possibly, some data.
4. How do serers work? Event Emitters
   1. An event emitter is a type of class which enables attaching a specific functionality to a particular event.
   2. Think about events on the browser – like a button click – and attaching a listener to that button with specific functionality to run.

const EventEmitter = require('events');

const myEmitter = new EventEmitter();

myEmitter.on('action-occurred', function() {

console.log('I heard that!')

})

*//somewhere else, later*

function action() {

myEmitter.emit('action-occurred');

}

action()

*the ‘****.emit****’ is what actually dispatches the event.*

1. Servers
   1. Node’s HTTP module, used for creating servers, is an example of an event emitter with a specific api!

const http = require('http');

const server = http.createServer();

boom. Server created.

1. Server functionality

const http = require('http');

const server = http.createServer();

server.on('request', (request, response) => {

console.log('request received');

let body = [];

request.on('data', chunk => body.push(chunk));

request.on('end', () => {

body = Buffer.concat(body).toString();

console.log(`Received body of request: ${body}`)

response.end('here is a status code & data')

});

});

* 1. ‘data’ is waiting for the ‘body’ of the request.
  2. And ‘end’ is waiting for no more chunks to be received?
     1. No – ‘end’ is looking for the end of the body of the request.

And, we need to tell the server to listen on a port!

server.listen('3000', () => {

console.log('listening on port 3000')

})

1. Postman
   1. Testing our backend code!
   2. A robust tool for testing our servers
   3. Test different request methods and routes (endpoints) to ensure that your backend is working as expected (even in error states).
   4. Send the exact data your server is expecting
   5. Enables *decoupling* of our frontend and backend code for easier/faster debugging.
2. Demo…
3. We should all download Postman.
4. Using Postman
   1. Open postman
   2. Create a “new collection”
   3. *Npx kill-port 3000*
   4. *Node index.js -> starts the server*
   5. *Create “new HTTP request”*
   6. Select “get” or whatever method we need.
   7. Tip: when we make changes, we will probably have to restart our server.
   8. Make sure you “end” the response when the “end” of the body is reached.
      1. *Take note that we end this with a “response.end”*
      2. *Vs. request.on(‘end’, cb )*
5. When the data comes in, it comes in “streams” – in a stream of data packets.
   1. A stream is a way of handling a large amount of data that we don’t want to hold in its entirety.
   2. The key goal is to limit the buffering of data to acceptable levels, such that sources and destinations of differing speeds will not overwhelm the available memory.
   3. “Buffer” is essentially a place in memory.
      1. 70% “buffer” means the buffer has been 70% filled…
   4. Our request comes in as a “buffer” data type…?
   5. Every stream broadcasts certain types of events.
      1. Two main types of streams in Node:
         1. Readable Streams
            1. Events: Data, end, close, error
            2. Methods: pause(), resume(), destroy(), pipe()
            3. The Request Object is a readable stream.

This is passed into a handler that implements the readablestream interface.

* + - * 1. It manages the body in chunks.
        2. “Chunk” is written in buffer lingo… hexadecimal representation of the data that was passed in.
      1. Writeable Streams

1. “Process”
   1. Global variable on the back end.
   2. It is an instance of the program and the environment it is running in.
   3. The computer’s OS can manage many processes simultaneously.
   4. In node, we can get info about the process by using the keyword “process”
   5. The below are quite important for command-line interfaces.
   6. These are basic streams.
   7. Stdin (standard in)
      1. Input to a computer “process”
         1. process.stdin
   8. Stdout (standard out)
      1. The output of a computer “process”
         1. process.stdout