Node JS Intro - C1

Created	@May 28, 2021 6:51 PM
Class	Node Backend
Type	
Materials	
Reviewed	☐

What is NodeJS ?

- Open Source Runtime
- Built on chrome's V8 JS engine
- Created by Ryan Dhal in 2009
- Evolved since creation for devs to do almost anything

What we can create with NodeJS ?

- Tooling
- API's
- CDNs
- Shareable Libraries
- Desktop Applications
- IOT

Browser JS vs Node JS

Aa Property	Node	Browser
Usage	Build server side logic and scripts	Build interactive visual apps for web
DOM	NO, but can virtualize	YES
WINDOW	NO, but has something called as GLOBAL	YES
Modules	YES	Optional, came in ES6
Filesystem Access	YES	NO
Async	YES	YES
Broswer API	NO	YES

How can we get information into and out from node program ?

Node models I/O in the most efficient way.

How to print something on a screen ?

Everyone knows that we can use

```
console.log("Hello");
```

This is but just an exposed API. But how node connects to the environment. It uses POSIX.

POSIX - Wikipedia

The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines the both system- and user-level application programming interfaces (API), along with command line shells and utility interfaces, for

W https://en.wikipedia.org/wiki/POSIX

In a nutshell, POSIX is the way C-style programs integrates with Linux style OS. In C, we read and write form STDIN, STDOUT and we have corresponding file descriptors for them (0,1,2). But how to access those I/O streams in node ?

Here we can use a global `process`.

But first a different question, why we care about this in Node ? Why we didn't dig it deep in browsers. So node is not shipped with any details about I/O, you won't find anything in the specs. JS is i/o agnostic, thats why JS is flexible and gets adjusted to different environment. In browsers we don't have POSIX, instead it is a direct connection to the developer tools.

for Node:

```
process.stdout.write("Hi");
```

This will output without newline character, but `console.log` will give you a trailing new line. But we are getting a notion that console.log is just a wrapper for the above code, but that's not true, it does way more things than expected.

So what happens with the above code is, it just carries a bunch of characters and dumps it to the stdin, but that's not the most efficient way to go forward. A more efficient way will be to use binary stream with buffer.

What about standard error ?

```
console.error("OOPS");
// or
process.stderr.write("Heyy");
```

Both of the above can be used.

```
console.log("Hey");
console.error("OOPS");
```

if we try running `node app.js` both will print but, they are different. Now we will do file descriptor redirection → node app.js 1>/dev/null it will only give "OOPS" and with 2 instead of 1, we get "Hey". It will redirect the logs based on the requirement. We can also do

```
node app.js 2>/dev/null 1>&2
```

Now both of them will be gone and redirected to error logs.

What about standard input ?

We will discuss it later, because it's a bit tricky.

Globals in Node JS

Node gives you a bunch of globals, but just like browsers you shouldn't make them

- **process:** Has information about the environment the program is running in, like machine info

  ```
  console.log(process); console.log(global);
  ```

- **require:** function to find and use module in the current module. JS adopted module pattern and require is the backbone of it.

  ```
  console.log(require);
  ```

- **__dirname:** the current directory path. It is only available with commonJS moduling and not in the es moduling.

  ```
  console.log(__dirname);
  ```

- **module:** has information about the current module, methods for making modules consumable.

```
console.log(module);
```

- **global:** like window, its the "global" object. Almost never use it.
- ... many more

What is a module ?

Mechanisms for splitting JavaScript programs up into separate modules that can be imported when needed.

Creating Modules

All you NodeJS code is a module.

As the author, how and what to expose from your module to other modules.

You do this with the module global object provided by nodejs runtime.

```
const add = (num1, num2) => {
  return num1+num2
}
module.exports = add; // example of default export
```

```
const add = (num1, num2) => {
  return num1 + num2;
}

// the below function will not be available for any other module to import
const notPublic = (num1, num2) => {
  console.log(num1, num2);
}

module.exports = {add, thing() {}, value: 10}; // named export
```

We can also directly write the methods in the module.exports line but we won't be able to refers methods in that case. There is no implicit way of exporting everything, you can make a separate object and then pass it to module.exports. But we should not do it, because it can reduce performance of the build tools because it's not the general way to handle this.

```
const add = (num1, num2) => {
return num1 + num2;
}

// the below function will not be available for any other module to import
const notPublic = (num1, num2) => {
console.log(num1, num2);
}
const myObj = {add, thing() {}, value: 10};  // don't do this
module.exports = myObj; // named export
```

How to require these modules ???

```
const action = require('./index');

console.log(action(1,2));
```

Creating ES Modules

Es modules require a special file extension called .mjs .

```
// utils.js
export const action = () => {

}

export const run = () => {

}
```

```
// app.js

import { action, run } from './utils'
```

Few things happening here. Let's look at the `utils.js` file. Here we're using the `export` keyword before the variable declartion. This creates a named export. With the name being whatever the variable name is. In this case, a function called `action`.

Now in `app.js`, we `import` the action module from the `utils` file. The path to the file is relative to the file you're importing from. You also have to prefix your path with a `'./'`. If you don't, Node will think you're trying to import a built in module or npm module. Because this wa a named export, we have to import with brackets `{ action, run }` with the exact name of the modules exported. We don't have to import every module that is exported.

Usually if you only have to expose one bit of code, you should use the `default` keyword. This allows you to import the module with whatever name you want.

```
// utils.js
default export function () {
  console.log('did action')
}
```

```
// app.js
import whateverIWant from './utils'
```

Modules: Packages | Node.js v16.5.0 Documentation

Node.js will treat as CommonJS all other forms of input, such as .js files where the nearest parent package.json file contains no top-level "type" field, or string input without the flag--input-type. This behavior is to preserve backward compatibility. However, now that Node.js supports both CommonJS and ES modules, it is best to be

https://nodejs.org/api/packages.html

Internal Modules

Node.js comes with some great internal modules. You can think of them as like the phenonminal global APIs in the browser. Here are some of the most useful ones:

- `fs` - useful for interacting with the file system.
- `path` - lib to assit with manipulating file paths and all their nuiances.
- `child_process` - spawn subprocesses in the background.
- `http` - interact with OS level networking. Useful for creating servers.

Command Line Scripts

We can make shell script of node using shbang called as hash bang (#!) at the top of the nodejs file.

```
// test.js
#!/usr/bin/env node

console.log("hello world");
```

Now we need to give executable rights to this file using the command `chmod u+x test.js` and then execute it using `./test.js`

How to pass command line arguments?

we can use `process.argv` . It gives us an array of arguments for both shell and normal execution

```
#!/usr/bin/env node

console.log("hello world");

console.log(process.argv);
```

the output will be:

```
[
  '/opt/homebrew/Cellar/node/16.0.0/bin/node',
  '/Users/sanketsingh/Documents/DevZone/NodeCourse/CLI/test.js',
  '--hello=world'
]
```

Now we don't need the first two arguments so we can splice them.

```
#!/usr/bin/env node

console.log("hello world");

console.log(process.argv.splice(2));
```

Now there are a bunch of conventions that exist for command line arguments, so we can't write every rule of input on our own. So we have an inbuilt package called as `minimist` .

minimist

parse argument options This module is the guts of optimist's argument parser without all the fanciful decoration.

https://www.npmjs.com/package/minimist

```
var args = require("minimist")(process.argv.splice(2))
console.log(args);
```

we can add configuration to minimist.

```
var args = require("minimist")(process.argv.splice(2), {
    boolean: ["help"],
    string: ["file"]
})
console.log(args);
```

`./ex1.js –help=foobar`

File System

Node provides us `fs` module to manipulate file system.

```
// template.html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
  <h1>{title}</h1>
  <p>{body}</p>
```

```
</body>
</html>
```

```
// index.mjs
import { readFile } from 'fs/promises'

let template = await readFile(new URL('./template.html', import.meta.url), 'utf-8')
```

The fs module has import for promise based methods. We'll opt to use those as they have a cleaner API and using async + non-blocking methods are preferred. More on that later. Because we're using .mjs files, we don't have access to __dirname or __filename which is what is normally used in combination with the path module to form an appropiate file path for fs. So we have to use the URL global that takes a relative path and a base path and will create a URL object that is accepted by readFile. If you were to log template, you'd see that its just a string.

Write a file

Writing a file is similar to reading a file, except you need some content to place in the file. Let's interpolate the variables inside our template string and write the final html string back to disk.

```
import { readFile, writeFile } from 'fs/promises'

let template = await readFile(new URL('./test.html', import.meta.url), 'utf-8')

const data = {
  title: 'My new file',
  body: 'I wrote this file to disk using node'
}

for (const [key, val] of Object.entries(data)) {
  template = template.replace(`{${key}}`, val)
}

await writeFile(new URL('./index.html', import.meta.url), template)
```

You should now have a index.html file that is the same as the template.html file but with the h1 and body text substituted with the data object properties. This is some powerful stuff 🔥! Open it in a browser and see it work. At their core, static analysis tools like

TypeScript, Babel, Webpack, and Rollup do just this. Also, please don't use my hacky templating "engine" in a real app! 🤣