Intro to Node.js

Springboard

Download Demo Node.js Code

Node.js

- A JavaScript environment that runs server-side
  - It uses the Chrome v8 engine, but doesn't require/use Chrome
- Can be used to build any kind of server-side JS
  - Including building web applications!
  - or as a general-purpose scripting language

Why Node.js?

- The entire stack (frontend/backend) can be JavaScript
- There is an extensive set of add-on libraries via ***npm***
  - Many can also be used in client-side JS
- Widely used

Starting Node

```
$ node myScript.js
Hi there!
```

or, for an interactive REPL:

```
$ node
>
```

(***.exit*** or ***Control-D*** to return to the normal terminal)

NPM

- Massive registry of add-on libraries
- Command line tool, ***npm***, comes with Node
- Easy dependency management for a project
- Register with npm to publish open-source or proprietary packages

Starting a Project with NPM

```
$ cd my-project

$ npm init
```

- Creates ***package.json*** with metadata & dependencies
- In a hurry? Want the defaults? `npm init --yes`

Example *package.json*

```
{
  "name": "node-files",
  "version": "1.0.0",
  "description": "Exercise to create file listing functions.",
  "main": "",
  "scripts": {
    "start": "node index.js"
  },
  "author": "Whiskey the Dog",
  "license": "ISC",
  "dependencies": {
    "axios": "^0.18.0"
  }
}
```

Store your ***package.json*** in Git

Installing Packages

In the root project directory:

```
$ npm install axios
```

- Adds latest version of axios to ***dependencies*** object in ***package.json***
- Installs the package in a local ***node_modules*** folder

node_modules

A directory containing all dependencies in the root directory of your project.

Always add node_modules to .gitignore

(It's just a collection of dependencies that can be reinstalled)

Reinstalling Packages

When you clone an existing project from GitHub or somehow lose your ***node_modules***, here's how you get those dependencies:

```
$ npm install
```

- `npm install` without arguments uses ***dependencies*** object in ***package.json***
- Similar to `pip install -r requirements.txt` in Python

package-lock.json

- ***package.json*** contains metadata & dependency *requirements*
- For `npm install axios`, this lists dependency as `"axios": "^0.18.0"`
  - Meaning "use axios version 0.18.0 or greater" (because of **^** symbol)
- ***package-lock.json*** contains *exact* versions actually installed
- ***npm install*** looks at ***package-lock.json*** first, before ***package.json***

NPM Summary

- Install dependencies to project's with `npm install <package_name>`
  - This downloads & adds to ***node_modules*** directory
- Dependencies are notated in ***package.json*** and ***package-lock.json***
- ***package-lock.json*** locks in the exact versions as installed
- Always commit ***package.json***; never commit ***node_modules***.
- Use `npm install` on a fresh repository to get a new ***node_modules***

Node *process* Library

process

Node provides a global object, ***process***, for managing the current script.

Using ***process***, we can:

- Access environment variables.
- See the command-line arguments passed to the script.
- Kill the script.

Peruse the documentation for a full list of features.

process.env

`process.env.SECRET_KEY`

Get value of environmental variables from shell

process.env is an object; its keys are the names of environment varibles.

```
$ export SECRET_INFO=abc123
$ node
> process.SECRET_INFO
'abc123'
```

process.argv

`process.argv[index]`

process.argv is an array of command-line arguments given to start this program

process.argv Example

demo/basics/showArgs.js

```
const argv = process.argv;

for (let i = 0; i < argv.length; i += 1) {
  console.log(i, argv[i]);
}
```

```
$ node showArgs.js hello world
0 '/path/to/node'
1 '/path/to/showArgs.js'
2 'hello'
3 'world'
```

process.exit

`process.exit(exit_code)`

Exit the program immediately and return an exit code to the shell.

Traditionally, 0 is "no error"; other numbers (1, 2, etc.) are script errors.

The Module System

Modules

Modules are the way to share code across different files in a Node project.

You might hear this system referred to as "CommonJS Modules".

There aren't ***<script>*** tags in the Node ecosystem, so you have to include other files by exporting/importing explicitly.

Importing a Project File

All imports use the ***require*** keyword.

To import a local project file, specify a ***relative path*** to that file:

demo/modules/other.js

```
const usefulStuff = require("./usefulStuff");

const results = usefulStuff.add(2, 3);

console.log(results);
```

- This usually means `./` for current directory or `../` for parent directory
- You don't need to include the file extension for `.js` and `.json` files.

Importing a Library

To import a built-in module or NPM package, simply exclude the relative path.

demo/modules/google.js

```
const axios = require('axios');

axios.get('http://google.com').then(function(resp) {
  console.log(resp.data.slice(0, 80), '...');
});
```

Node will look in its includes core modules and ***node_modules***.

Destructuring Imports

When importing an object, it is possible to destructure into variables:

demo/modules/other2.js

```
const { add, User } = require('./usefulStuff');

const results = add(2, 3);

console.log(results);
```

Exporting from a File

Use built-in ***module.exports*** to explicitly export things from a file.

demo/modules/usefulStuff.js

```
const MY_GLOBAL = 42;

function add(x, y) {
  return x + y;
}

class User {
  constructor(name, username) {
    this.name = name;
    this.username = username;
  }
}

const notNeededElsewhere = 'nope'; // I don't get exported!

// export an object
module.exports = {
  MY_GLOBAL: MY_GLOBAL,
  add: add,
  User: User
};
```

module.exports

Normally ***module.exports*** is an object; this can export multiple things.

But you can actually set it to whatever you want:

in one file

```
module.exports = function() {
  console.log('hello welcome to Node.js');
}
```

in another file

```
const sayHello = require('./example');

sayHello();
// hello
```

Modules Summary

- Export contents of files with ***module.exports*** *(which is usually an object)*
- Import contents of local files using the ***require*** keyword with a relative path
- Import libraries using ***require*** keyword without path—just the library name

Node Callbacks

Many Node library functions utilize asynchronous callbacks by default.

For example, to read a file:

```
fs.readFile('myFile.txt', 'utf8', function(err, data) {
    // process file here...
});
```

Error-First Callbacks

Node.js callbacks usually conform to an **"error-first" pattern.**

- The callback function's first parameter should be listed as ***error***. Node will supply an ***error object*** (if something bad happened), otherwise ***null*** as arguments.
- Then follow the other parameters, if there are any:

```
fs.readFile("myFile.txt", "utf8", function(err, data) {
  if (err) {
    // handle error
  }
  // otherwise we're good
});
```

Note: The Docs

Nice reading on error-first callback pattern from the official docs here.

Handling Errors

In the browser, there are different things to do with errors:

- Show some "an error happened" message in the DOM
- Pop up an alert box
- Log to the console

In Node, you will often do one of these:

- Log the error to the console
- Exit the program with `process.exit(1)` (***process*** is always there)

Note: util.promisify()

Sick of callbacks? For an advanced reading, consider promisifying your callbacks.

Just don't look at ***callbackify***. *We don't talk about callbackify.* 🤐

File System Module

fs

The `fs` module is built-in and provides an interface to your local file system.

- It is commonly used to read and write files.
- To start using it, `const fs = require('fs');` (no installation necessary).

Reading Files

The default method for reading files is asynchronous, using a callback.

`fs.readFile(path, encoding, callback)`

- ***path***: path to file (relative to working directory)
- ***encoding***: how to interpret file
  - for text files, this is almost always "utf8"
  - for binary files (like an image), omit this argument
- ***callback***: function that takes ***error*** and ***data***

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

fs.readFile('myFile.txt', 'utf8', function(err, data) {
  if (err) {
    // handle possible error
    console.error(err);
    // kill the process and tell the shell it errored
    process.exit(1);
  }
  // otherwise success
  console.log(`file contents: ${data}`);
});

console.log('reading file');
// file won't have been read yet at this point
```

Note: Synchronous Version

It's common, when using Node, to use asynchronous functions—but for some things, like reading/writing files, there are synchronous methods you can use:

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

try {
  // store the read file contents
  var contents = fs.readFileSync('myFile.txt', 'utf8');
  console.log(`file contents are "${contents}"`);
} catch (error) {
  // errors thrown by fs will be caught here
  console.error(error);
  // kill the process and tell the shell it errored
  process.exit(1);
}
```

Writing Files

`fs.writeFile(path, data, encoding, callback)`

- ***path***: path to file (relative to working directory)
- ***data***: data to output to file (typically a string)
- ***encoding***: how to write file
  - for text files, this is almost always "utf8"
  - for binary files (like an image), omit this argument
- ***callback***: function that takes ***error***

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

const content = 'THIS WILL GO IN THE FILE!';

fs.writeFile('./files/output.txt', content, "utf8", function(err) {
  if (err) {
    console.error(err);
    process.exit(1);
  }
  console.log('Successfully wrote to file!');
});

console.log('writing file...');
// file won't have been written yet at this point
```

Note: Synchronous Version

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

const content = 'THIS WILL GO IN THE FILE!';

try {
  fs.writeFileSync('./files/output.txt', content);
  console.log('Successfully wrote to file!');
} catch (error) {
  console.error(`File write failed: ${error}`)
  process.exit(1);
}
```

File System Summary

- ***fs*** is a built-in module; import with `require('fs')`.
- The default methods are asynchronous reading and writing
- Stringify file contents when writing with encoding (normally "utf8")

Node vs Browser JS

- Most programmatic behavior is exactly the same (yay v8!)
- The "global object" isn't ***window***, it's ***global***
  - This is where global vars go, where ***setTimeout*** is, etc
- Node doesn't have ***document*** & DOM methods
- Node provides access to filesystem & can start server processes
- Many NPM libraries are "isomorphic" (can be used in web JS or Node)

Looking Ahead

What's Next

- File I/O Exercise
- Writing our first servers with Express.js