# Introduction

What is Node?

A wrapper around V8 with build-in modules providing rich features through easy-to-use asynchronous APIs.

Why Node?

* Wrapper around V8 (execute JS)
* Build-in modules (fs, http, crypto, zip…) https://nodejs.org/api/index.html
* Asynchronous APIs (no threads)
* C++ addons
* Debugger and other utilities
* NPM (Node Package Manager)
* Module dependency manager

Why Node is names Node

In Node we build simple, small, single process building blocks, nodes, that can be organized with good networking protocols to have them communicate with each other and scale up to build large distributed programs. Scaling a Node application is not an afterthought, it's built right in to the name.

What You Get When You Install Node

For MacOS, use Homebrew (<https://brew.sh/>) with the command ‘brew install node’.

For both Mac and Linux, Use nvm (<https://github.com/nvm-sh/nvm>)

# Getting Started with Node

Node’s REPL(Read/Eval/Print/Loop) Mode

> math.random()

Node will:

1. read your line
2. evaluate it
3. print the result
4. and then go back to waiting for further lines.

> .help

.break get our from REPL Mode

.clear Same as .break

.editor Enter editor mode

.exit Exit the repl

.help Print this help message

.load Load JS from a file into the REPL session

.save Save all evaluated commands in this REPL session to a file

Tab and Underscore

Single Tab: Auto complete

Double Tab: See a list of possible things that you can type from whatever partially typed string you have.

Underscore (\_) variable: It stores the value of the last evaluated expression.

Executing Scripts

// hello-world.js

const http = require(‘http’)

const server = http.createServer((req, res) => {

res.end(‘Hello World\n’);

});

server.listen(4242, () => {

console.log(‘Server is running…’);

});

> node hello-world.js

Server is running…

Node is idle status. 4040 port is opened to get a http request.

// print-line.js

console.log(‘Hello Nodejs!’);

> node print-line.js

Hello Nodejs!

>

Node will exit after print ‘Hello Nodejs!’

Working with Timers

// setTimeout-example.js

setTimeout( () => { console.log(‘Hello after 4 seconds!’); }, 4\*1000 );

// setTimeout-with-function.js

const func = () => console.log(‘Hello after 4 seconds!’);

setTimeout( func, 4\*1000 );

// setTimeout-with-function-and-arguments

// func(arg1, arg2, arg3…) => setTimeout(func, delay, arg1, arg2, arg3…)

const func = (name) => console.log(name‘, hello after 4 seconds!’);

setTimeout( func, 4\*1000, ‘Joe’ );

// setInterval-example.js

setInterval( ()=>console.log(‘Hello every 3 seconds!’), 3\*1000 );

// clearing-timers.js

const timerId = setTimeout( () => console.log(‘Not going to print this!’), 0 );

clearTimeout(timerId); // clearInterval, clearImmediate

Node’s Command Line Interface

// Node cli command

$ node –v // version of nodeJs

$node –p “os.cpus().length” // print number of cpus

$node –p “process.versions.v8” // print version of v8

$node –h // help

$ NODE\_DEBUG = ‘http’ node server.js

The “process” Object

$VAL1=10 VAL2=20 node a.js

// equal to above

$export VAL1=100

$export VAL2=200

$node a.js

// or you can use process.argv

$node –p “process.argv” hello 42

// stdin / stdout

process.stdin.on('readable', () => {

const chunk = process.stdin.read();

if (chunk != null)

process.stdout.write(chunk);

});

// exit

setTimeout(() => process.exit(), 1000);

process.on('exit', () => {

console.log('Process will exit now. See you later!');

});

console.log('Hello!');

# Modern JavaScript

Variable and Block Scopes

{

// block scope

{

// nested block scope (let and const are safe to use)

}

}

if (true) {

// block scope

}

for (let I = 0; I <= 10; i++) {

//block scope

} // var: you can access i from out of block scope, let: only access i within block scope

function sum (a, b) {

var result = a + b; // function scope

} // you cannot access result from out of function scope

// const: reference assign

// const can be use variable/array/object/function

// scalar: immutable

const answer = 42;

const greeting = ‘Hello’;

// array/object: const variable is pointing to same array/object but content can be mutated

const numbers = [1,2,3];

const person = {firstName: ‘John’, lastName: ‘Doe’}

Arrow Functions

// this scope for function vs arrow function

this.id = ‘experts’;

const testerObj = {

// this: testerObj, caller of func1

func1: function() {

console.log(‘func1’, this);

},

// this: global, testerObj’s scope

func2: () => {

console.log(‘func2’, this);

},

};

testerObj. func1();

testerObj. func2();

// standard syntax

const square = (a) => {

return a\*a;

};

// remove {} and return

const square = (a) => a\*a;

// remove ()

const square = a => a\*a;

[1,2,3,4].map(a => a\*a);

Object Literals

const mystery = “answer”;

const obj = {

// regular properties

pl1: 10,

p2: 20,

// functions

f1() {},

f2: () => {},

// dynamic property

[mystery]: 42,

};

console.log(obj.answer); // 42

Destructing and Rest/Spread

// destructing example

// const PI = math.PI; const E = math.E; const SQRT2 = math.SQRT2; is equal to:

const {PI, E, SQRT2} = math;

// with require

const { readFile } = require(‘fs’);

const circle = { label: ‘circleX’, radius: 2 };

// automatically bind radius from circle

// default of precision is 2 and you can omit. ( ={}: make whole second part optional)

const circleArea = ({ radius }, { precision = 2 } = {}) => (PI \* radius \* radius).toFixed(precision);

console.log(circleArea(circle));

console.log(circleArea(circle, {precision: 5}));

const [first, second, , forth] = [10, 20, 30, 40];

// rest & spread example

const [first, ...others] = [10, 20, 30, 40];

console.log("first: " + first + ", others: " + others);

const data = {

tmp1: 10,

tmp2: 20,

firstName: 'John',

lastName: 'Doe',

};

const {tmp1, tmp2, ...person} = data;

console.log("tmp1: " + tmp1 + ", tmp2: " + tmp2);

console.log(person);

const newArray = [...others];

console.log(newArray);

const newObject = {...person};

console.log(newObject);

Template String

let str = `

<div>

${Math.random()}

</div>

`;

console.log(str);

Classes

class Person {

constructor(name) {

this.name = name;

}

greet() {

console.log(`Hello ${this.name}!`);

}

};

class Student extends Person {

constructor(name, level) {

super(name);

this.level = level;

}

greet() {

console.log(`Hello ${this.name}, your level is ${this.level}!`);

}

};

const o1 = new Person("Max");

const o2 = new Student("Tina", "1st Grade");

const o3 = new Student("Mary", "2nd Grade");

o3.greet = () => console.log("I'm special!");

o1.greet();

o2.greet();

o3.greet();

Promises and Async/Await

const https = require('https');

function fetch(url) {

return new Promise((resolve, reject) => {

https.get(url, (res) => {

let data = '';

res.on('data', (rd) => data += rd);

res.on('end', () => resolve(data));

res.on('error', reject);

});

});

}

fetch('https://www.google.com/').then(data => {

console.log(data.length);

});

// much clearer than above

(async function read() {

const data = await fetch('https://www.google.com/');

console.log(data.length);

})();

# Modern JavaScript

The NPM Command

$ npm –v // check npm version

$ npm install –g npm // install npm globally

$ npm i express // install express as ‘dependencies’

$ npm i –D nodemon // install nodemon as ‘devDependencies’

$ npm init // it will create package.json file based on your answers

$ npm init –year // it will create package.json with default

Last command will install express under node\_modules folder and update package.json/package-lock.json

The package.json/package-lock.json Files

package.json: The file contains npm package information

package-lock.json: This file records the version information of all the packages. (e.g. Let's assume between the time that you added the express dependency and the time a team member pulled your code to use it, a new version of express npm package was released. Your team member will not get that new version when they run npm install. They are going to get the exact same version that you used because of package-lock. json.)

Semantic Versioning (SemVer)

^ ~

4. 2. 3

Major Minor Patch

Breaking Changes Backward Compatible Bug Fixes

^: allow minor changes (e.g. 4.x.y, x > 2)

~: allow patch changes (e.g. 4.2.x, x > 0)

npm semver calculator: https://semver.npmjs.com

Creating and Publishing an NPM Package

// test/index.js

const print = require('frame-print')

print('Hello NPM!');

// frame-print/index.js

module.exports = function print(msg) {

console.log('\*\*\*\*\*');

console.log(msg);

console.log('\*\*\*\*\*');

};

// publish frame-print package to npmjs

frame-print$ npm init --yes // create package.json file under frame-print folder

frame-print$ npm login // login to npmjs

frame-print$ npm publish // publish frame-print package to npmjs

// install frame-print on test folder

test$ npm i frame-print

NPX and the NPM Run Script

NPX: npm package runner

// package.json

"scripts": {

"start": "node server.js",

"test": "jest",

"check": "eslint server.js"

},

$ npm start // same as “node server.js”

$ jest // error: command not found. Jest exists under node\_modules you can access it

$ npx jest or $ npm test // will run jest

$ npm run check // check is custom script so “run” needed after npm

$ eslint // error: command not found

$ npx eslint –init // create eslint configuration

$npm help npm-scripts // show all the npm script description

// package.json

"scripts": {

"start": "node server.js",

"test": "jest",

“posttest”: “node –e ‘console.log(\“Done!\”)’”

"check": "eslint server.js"

},

$ npm test // will run jest and print ’Done!’

Updating NPM Packages

$ npm ls // show all the packages version

$ npm update // update npm packages

$ npm uninstall express // uninstall package

$ npm show <package\_name> versions(or version) // show package versions (or latest version)

$ npm outdated // show current/wanted/latest version information for all packages

$ npm I lodash@4.1.2 (or lodash@latest) // install specific version or latest version of package

# Modules and Concurrency

Defining and Using Node Modules

Module (In Node.js): A file or a folder that contains code, it wraps file with the function.

// will print arguments of dynamicArgsFunction. E.g. [Arguments] { '0': 1, '1': 2, '2': 3, '3': 4 }

function dynamicArgsFunction() {

console.log(arguments);

}

dynamicArgsFunction(1,2,3,4);

/\*

it outputs 5 arguments in node. Module is function not file!

exports, module, require is not global objects, there are arguments!

wrapper.js

\*/

// function (exports, module, require, \_\_filename, \_\_dirname) {

let g = 1; // variable inside function

console.log(arguments);

exports.a = 42; // same as module.expots.a = 42; expors are not global object, it is arguments

// return module.exports;

// }

/\*

test.js: will display {a: 42} because wrapper.js returns module.exports

\*/

const wrapper = require(‘./wrapper’);

console.log(wrapper);

Examples of Module APIs

// Top-level API is an object, object.js

exports.language = 'English';

exports.direction = 'RTL';

exports.encoding = 'UTF-8';

// result.js

const obj = require('./object’);

console.log(obj.language, obj.direction, obj.encoding); // English RTL UTF-8

// Top-level API is an array, array.js

module.exports = [1,2,3,4];

// result.js

console.log(require('./array')); // [1,2,3,4]

// Top-level API is a string, string.js

module.exports = `

<div>

Hello World!

</div>

`;

// result.js

console.log(require('./string')); // <div>Hello World!</div>

// Top-level API is a function, function.js

module.exports = title => `

<div>

${title}

</div>

`;

// result.js

const func = require('./function');

console.log(func('Hello World, haha!')); // <div>Hello World, haha!</div>

Node’s global Object

global keyword in node is equal to window keyword for js.

We should NOT use this global object!

console.dir(global, {depth: 0}); // print global tree structure only first depth

console.log(setTimeout);

console.log(global.setTimeout); // setTimeout is under global

// sample.js

global.thisIsGlobalVariable = 47;

// result.js

const sample = require('./sample');

console.log(thisIsGlobalVariable); // will print 47. Because we set global variable

The Event Loop

What Node users to process asynchronous actions and interface them for you so that you don’t have to deal with threads.

/\*

the Event Loop will monitor the setInterval timer, and every 5 seconds it'll take the interval's callback, which is the first argument to setInterval, the arrow function here, and it'll send this arrow function to V8, and V8 will execute what's inside that function. Because this is an every 5-second kind of thing, the Node process is going to continue to run forever, and it will not exit.

\*/

setInterval( ()=>console.log('who needs this event loop!'), 5000);

EVERY Node process starts this infinite loop that we call the Event Loop, but when the process has no asynchronous operations to perform, the Event Loop will exit and the Operating System will terminate that Node process.

Errors vs. Exceptions

Error is problem (cannot handle by program) and Exception is condition (can handle by program).

const path = require('path');

const fs = require('fs');

const files = ['test1.txt', 'test3.txt', 'test2.txt']; // test3.txt is not exist

files.forEach(file => {

try {

const filePath = path.resolve(process.env.HOME, file);

const data = fs.readFileSync(filePath, 'utf-24'); // wrong encoding

console.log('file data is: ', data);

} catch (err) {

if (err.code == 'ENOENT') {

console.log('file not found!'); // if file not exist

} else {

throw err; // otherwise

}

}

});

Node’s Asynchronous Patterns

// read file synchronous, so print file data and then ‘TEST’

const fs = require('fs');

const data = fs.readFileSync(\_\_filename);

console.log(data);

console.log('TEST');

// read file asynchronous using callback so print ‘TEST’ then file data

const fs = require('fs');

fs.readFile(\_\_filename, function cb(err, data) {

console.log(data);

});

console.log('TEST');

// if there are jobs after readFile then callback will be nested

const fs = require('fs');

fs.readFile(\_\_filename, function cb(err, data) {

fs.writeFile(\_\_filename + '.copy', data, function cb2(err) {

// nest more callback here...

});

});

console.log('TEST');

// use promise to eliminate nested callback issue

const fs = require('fs');

const util = require('util');

const readFile = util.promisify(fs.readFile);

async function main() {

const data = await readFile(\_\_filename);

console.log(data);

}

main();

console.log('TEST');

OR

const {readFile} = require('fs').promises;

async function main() {

const data = await readFile(\_\_filename);

console.log(data);

}

main();

console.log('TEST');

Event Emitter

// Nothing print, emit before subscribe

const EventEmitter = require('events');

const myEmitter = new EventEmitter();

myEmitter.emit('TEST\_EVENT'));

myEmitter.on('TEST\_EVENT', () => console.log('TEST\_EVENT was fired!'));

myEmitter.on('TEST\_EVENT', () => console.log('TEST\_EVENT was fired!'));

myEmitter.on('TEST\_EVENT', () => console.log('TEST\_EVENT was fired!'));

// To fix this emit after or use setImmediate()

const EventEmitter = require('events');

const myEmitter = new EventEmitter();

// setImmediate: run in next iteration of event loop

setImmediate(() => myEmitter.emit('TEST\_EVENT'));

myEmitter.on('TEST\_EVENT', () => console.log('TEST\_EVENT was fired!'));

myEmitter.on('TEST\_EVENT', () => console.log('TEST\_EVENT was fired!'));

myEmitter.on('TEST\_EVENT', () => console.log('TEST\_EVENT was fired!'));

// myEmitter.emit('TEST\_EVENT');