Practical NodeJS

Azat Satklyčov
azats@seznam.cz,
http://sahet.net,
https://github.com/azatsatklichov/nodejs-app

Agenda

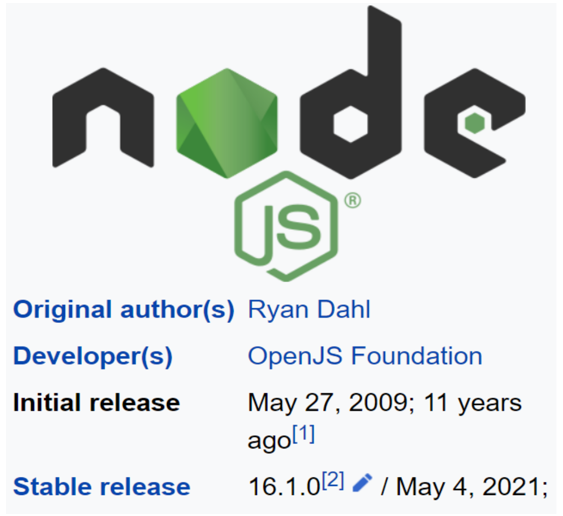
- ✓ Why Node.JS
- ✓ REPL Mode, Node CLI, NPM, Addons, Package
- ✓ Modules (exports, requires)
- ✓ Modern Javascript – templates, scoping,
- ✓ NodeJS Architecture. V8/chakra, libuv, ..
- ✓ Node.js Concurrency – EventLoop
- ✓ Event Driven Architecture – EventEmitters, Streams
- ✓ http /s - Web Frameworks, Templates, Debugger - utilities
- ✓ More build-in modules os, fs, child_process, crypto, dns, zlib, ..

Why Node.JS

- Free, fast, streamed, open-source, cross platform, motivates to build **SSJS** apps & desktop apps.
- Every 6 mo Release. Even[April]/Odd[October], once a new odd version is released the previous even version undergoes transition to Long Term Support (18 mo+12mo)

>node -p process.release.lts >node -p process.versions

- Originally has event-driven (**non-blocking**) architecture capable of async I/O. Aims to optimize throughput and scalability in web apps with many I/O operations. Node.js is not good for CPU-intensive tasks.
- Node.js is a JS runtime env. that runs on the V8 engine (or ~~Chakra~~) and executes JS code outside a web browser. VM (V8) is **Single threaded**.



Original author(s)	Ryan Dahl
Developer(s)	OpenJS Foundation
Initial release	May 27, 2009; 11 years ago[1]
Stable release	16.1.0[2] / May 4, 2021;

- NPM – CLI / Registry, Module dependency manager (CommonJS) – require. Webpack, …
- First class support for C++ addons. See medium article for addons on z/OS
- "JS everywhere" paradigm – single Lang. **Full-stack** for Front-End[JS], Back-End[JS], DB [Mongo, Postgres, .. supports JS syntax] . Rather than real-full-stack. E.g. Java dev. also knows JS, PL-SQL, W3C tech
- Large Ecosystem, SS Frameworks, Desktop apps., Mobile and IoT, More Users, ..

Node REPL Mode

Start REPL (Read Eval Print Loop) session [>node] to test quickly Node.JS & JS commands

> Math.random() 0.5574170173050714	> let x=6 **undefined**	> 23 =='23' true	Multiline? No navigate, no multi expression	Use Node .editor [^D, ^C] + multiline + paste from clipboard

>.help, .break ⬄ .clear, .editor, .exit, .load, .save. Shortcuts [^L], [^D]

TAB (single, double) & Underscore

Node.js has a set of built-in modules, classes, functions which you can use without any further installation.
C:\workspace\nodejs-app>node

> e >events. >events.EventEmiter	> let s = "" > s. > Arrays. Or >let arr=[] >arr. >. //same as .help		> _ //underscrore, $? >new Date() >let d = _

Custom REPL sessions

REPL Module: **repl.start()** – to customize session (e.g. colors, eval, use socket instead stdin,…)

> let x=5; undefined > b=2 2	> node .\10-repl.js > let x=5; > b=2 Thrown: ReferenceError: b is not defined

Control REPL Global Context: **r.context.lodash** = require('lodash');

- lodash.last(['s', 'dd', 'z'])
- node –help | more
- -p (prinat and eval), -c (--check), --v8-options, -r (--require)
- **node -p process.argv.slice(1) 13 "ahoj"**

Node CLI

```
>node -v
>node -p  “os.cpus.length”
>node -p “process.versions.v8”
7.0.276.38-node.19
>
```

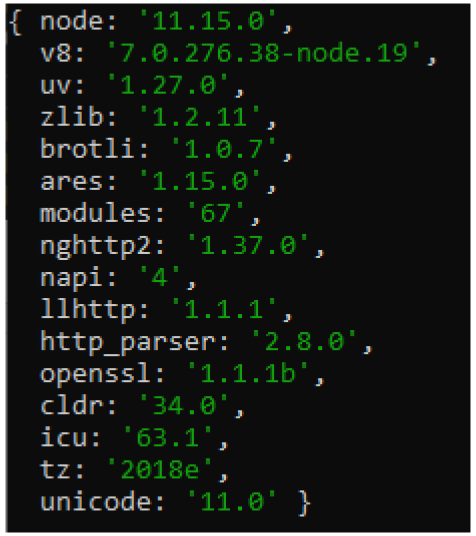
```
>node –h | [less / more]
>node --v8-options  (harmony, trace)
>node --v8-options | more
//--use-strict (enforce strict mode)
```

```
> node --v8-options | [find / grep]
"in progress“

//ENV-VAR
>NODE_DEBUG, NODE_PATH
```

Process obj

To see all globals: > //double tab (or>global.)

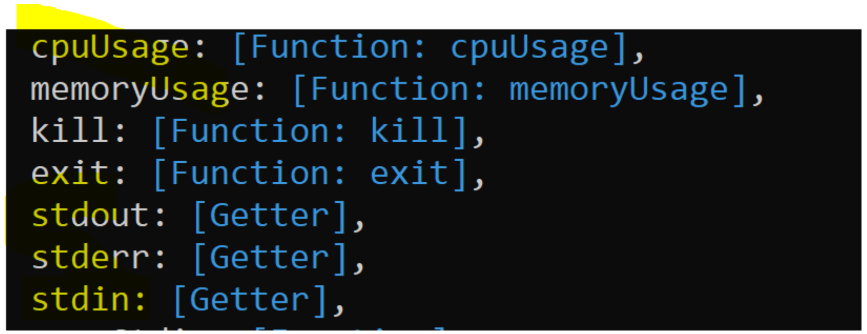
```
>node  -p process.versions
>node  -p process.release.lts
```



```
{ node: '11.15.0',
  v8: '7.0.276.38-node.19',
  uv: '1.27.0',
  zlib: '1.2.11',
  brotli: '1.0.7',
  ares: '1.15.0',
  modules: '67',
  nghttp2: '1.37.0',
  napi: '4',
  llhttp: '1.1.1',
  http_parser: '2.8.0',
  openssl: '1.1.1b',
  cldr: '34.0',
  icu: '63.1',
  tz: '2018e',
  unicode: '11.0' }
```

Can be used for dynamically configured values   (you have name here): e.g. **process.env.val1**

```
>node –v
>process
>process.env    [. Double TAB] [↵]
```



```
cpuUsage: [Function: cpuUsage],
memoryUsage: [Function: memoryUsage],
kill: [Function: kill],
exit: [Function: exit],
stdout: [Getter],
stderr: [Getter],
stdin: [Getter],
```

Other way to pass information to execution context of node

```
>process.argv    [. Double TAB] [↵]
>node -p "process.argv" ahoj 23
>node -p  "process.argv" 3-env-var.js ahoj 23
```

```
> console.log('dd')    // uses stdout stream.
dd
undefined
> process.stdout.write('dd\n')
dd
true
```

STREAMS CAN BE USED WITH PIPES
//process.stdin.pipe(process.stdout)

Modules

- ❑ Node.js **Built-in Modules** (exports, requires), Module.obj – not confuse it with Browser global

```
console.log(arguments);
//in Browser(undefined), in NodeJS ?

//wrapper-function: 5 arguments
//function
 (exports, module, require, __filename, __dirname){
 exports.x - not globally available like Browser
console.log(arguments);
//} (module.exports) //APIs
Caching
```

How require works, require('module')

- ❑ Resolving - finds absolute **paths**
- ❑ Loading
- ❑ Wrapping
- ❑ Evaluating
- ❑ Caching
- ❑ > node .\6-require-module.js

- NOTE: for core-modules RESOLVE return immediate

- Just to resolve(no exec/load) - require.resolve('..')
e.g. to check if optional-package installed or not

- ❑ Node.js global object, Wrapping and Caching Modules
- ❑ GLOBAL obj is like WINDOW in Browser

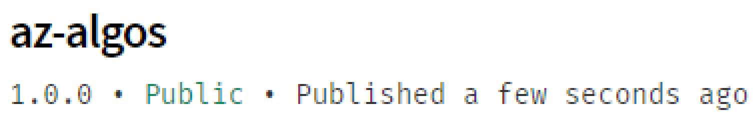
- ❑ CommonJS – Module Dep. Manager.
- ❑ Not like NPM more than ES Module

`process`, `buffer`, `setTimeout -> global.setTimeout`, `global.ans?` **not define global obj**

- ➢ **`7-global.js`**
- ➢ `Buffer.from(), .alloc() [filled], allocUnsafe(not) e.g.`
- ➢ `>Buffer.allocate(23), > Buffer.allocUnsafe(1000).toString()`
- ➢ `Also see REPL //double TAB or global.`

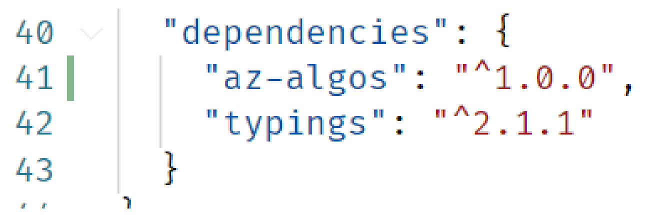
NPM, NPM Command

Why NPM
- npm (see pre-installed) installs from the npm registry [search, meta-info, readme]
- Code Share, Re-use
- Composability
- Versioning
- Team Work
- Anyone can publish anything (name must be unique)
- **Alternatives: yarn, pnpm**



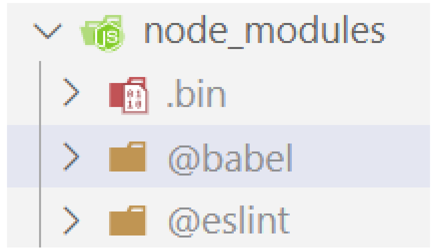
az-algos

1.0.0 • Public • Published a few seconds ago



```
"dependencies": {
  "az-algos": "^1.0.0",
  "typings": "^2.1.1"
}
```

Package (a.k.a MODULE)



```
node_modules
  .bin
  @babel
  @eslint
```

To see all config: > npm config list –l
- npm config list -l | find "init“
- npm config set/delete init-author-name “abc.xyz”
- npm config set save true //npm prune

>npm
>npm i --dry-run \\what will be installed to see before
>>npm ls –g (--depth=0, specific: npm ls lodash)
> npm ls -a –json >npm ls -g
>npm install/i -g npm //self update, -g: GLOBAL (tools,..)

package.json/package-lock.json (i/ci/shrinkwrap)
--save[**-S**] (prod) /--save-dev[**-D**] , **-O** (optinal dep), -g
>npm i -D nodemon
>npm uninstall nodemon
>npm help ci
- **--production** //e.g. nodeman no need in prod

[maven runtime - Weblogic, compile, test]
>npm **search** lodash >npm **home** lodash >npm **repo** lodash
>npm update lodash >npm uninstall lodash

- **Sematic Versioning** (SemVer): (**range** of acceptables): **~, ^**
- A.B.C [Major.Minor(**^** >=B).Patch(**~** >=C)]
- or X notation e.g. 1.x.
- If exact e.g. "express": "4.17.1“ same as “=4.17.1“
- **npm semver calculator** : https://semver.npmjs.com/

- Publishing a package (npm account)

>npm login (asks npm user/pwd/email)
>cd az-algos (create package.json via npm init)
>npm publish (publish package)
See: https://www.npmjs.com/package/az-algos
>npm install az-algos
>node .\5-npm-packages.js
> npm view az-algos //to check published library

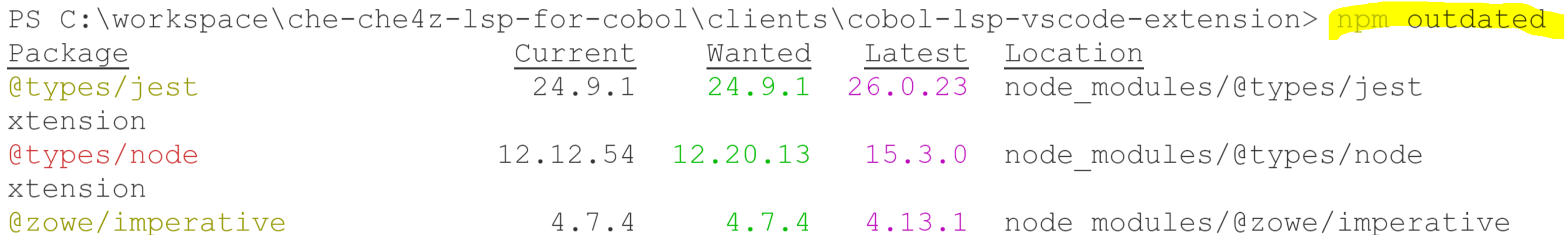
NPM, NPX

\>cobol-lsp-vscode-extension> npm **run** test
\>cobol-lsp-vscode-extension> npm t
\>cobol-lsp-vscode-extension> npm test (finds jest+run)
\>cobol-lsp-vscode-extension> npm jest (unknown)
\>cobol-lsp-vscode-extension> npx jest (npm execute)

\>npm help npm-scripts
EVENTS: 'posttest', 'pretest'

Updating NPM Packages

- npm update (respects all **SemVer,** also may override ~ to ^)
- npm ls –a
- npm i express (adds ^ defaultly) [or @3.4, @latest, @next]
- npm show express (shows info, LATEST, NEXT)
- npm show express version**s** maintainers
- npm outdated (better than manual work)
- npm update



```
PS C:\workspace\che-che4z-lsp-for-cobol\clients\cobol-lsp-vscode-extension> npm outdated
Package                       Current     Wanted    Latest  Location
@types/jest                    24.9.1     24.9.1   26.0.23  node_modules/@types/jest
xtension
@types/node                  12.12.54   12.20.13    15.3.0  node_modules/@types/node
xtension
@zowe/imperative                4.7.4      4.7.4    4.13.1  node_modules/@zowe/imperative
```

We know how require works, require('xyz')

- ❑ xyz.js -> xyz.json -> xyz-binary
- ❑ > node .\6-require-json.js

JSON and C++ Addons

Native bindings

Node.js provides a way to make ***Addons, also*** Read addons for z-OS

- npm install -g node-gyp@latest
- addon\addon-src> node-gyp configure
- > node-gyp build

Modern Javascript

Node.js != JS

https://tc39.github.io/process-document/

The TC39 Process

The Ecma TC39 committee is responsible for evolving the ECMA
discretion to alter the specification as it sees fit. However, the gen

```
const cube = (a) =>   a * a * a;
const html = `
  <div>
    ${Math.random()}
    <br/>
    ${cube(3)};
  </div>
`;
```

- ES TC39, https://github.com/tc39, TC-39 Process, Ecma-262
- V8 Engine will follow implementing TC39
- Yearly Releases since ES2015 [ES6], ES2016, ….
- 5 StagedProcess

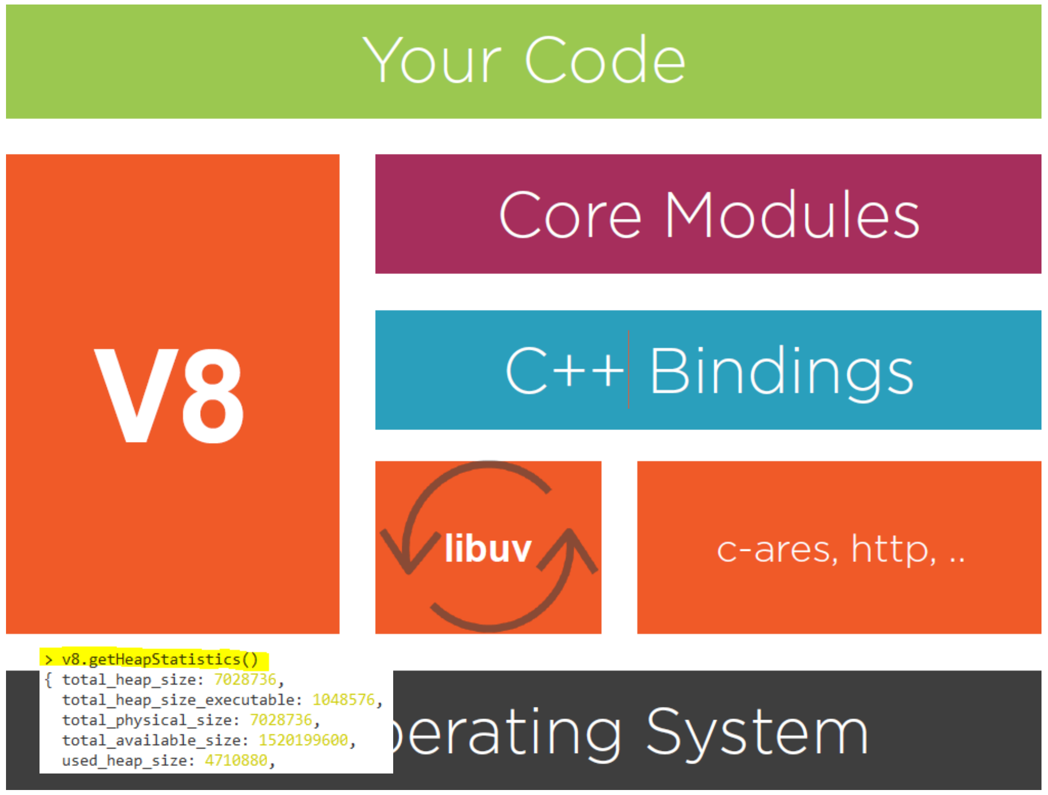
[0-Strawman,1-Proposal,2-Draft,3-Candidate,4-Finished]

- Babel - faster
- Variables and Block Scopes, Object Literals
- Arrow Functions
- Destructing and Rest/Spread
- Inheritance – state and action based (in Java action based)
- Promises and Async/Await

```
const dyno = "dynamo";
const LOG2E = Math.LOG2E;
const obje = {
  a1: 23,
  a2: "oka",
  f1() {},     //function with object literal
  f2: () => {},  //arrow function
  [dyno] :  63, //[] no arr, dynamo : 63
  LOG2E //shorter than LOG2E : LOG2E
};
```

- **Templates-string (interpolation, ` ${dyno-exp}`)** (also like **multi-lines** Java 13)
- **Dynamic properties**

NodeJS Architecture



```
> v8.getHeapStatistics()
{ total_heap_size: 7028736,
  total_heap_size_executable: 1048576,
  total_physical_size: 7028736,
  total_available_size: 1520199600,
  used_heap_size: 4710880,
```

V8 Feature Groups : Shipping, Staged, In Progress

- node -p process.versions.v8
- node --v8-options | find "in progress“
- node –harmony -p "'Node'.padEnd(8, '*')“
- node --v8-options | more
- node --v8-options | find "gc“
- **v8.getHeapSpaceStatistics()** (in repl mode)

Benefits

- **Node.js** - a runtime environment based on Chrome's V8
- **Single threaded architecture** – used as opportunity
- VM agnostic: V8 [>v8] /Chakra **Single threaded -** no race condition, locking issues, ..
- **Non-Blocking I/O -** Not waiting till I/O operation is complete.
- **Asynchronous -** Handle dependent code later once its complete.
- **libuv** (C-lib) – Handles all async events. Used by Node, Rust, Julia.

NodeJS – libuv, Ruby – EventMachine, Python – Twisted.

- **Robust technology stack** – built-in modules,

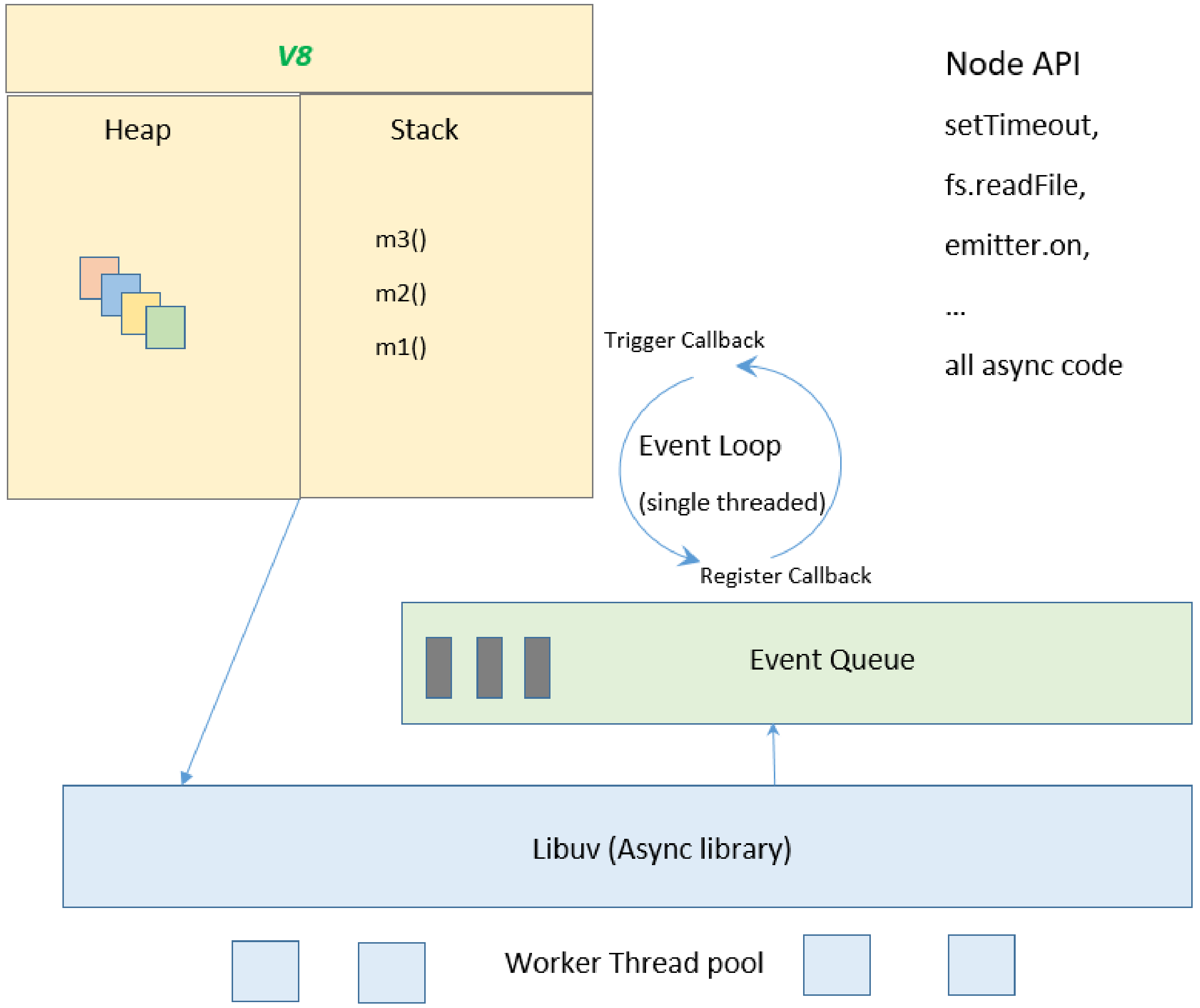
providing rich features via asynchronous APIs

- Dependencies: http-parser, c-ares, OpenSSL, zlib, gtest
- Node.js supports WebAssembly and as of Node 14 has experimental support of WASI

Drawbacks

Low performance on heavy computation - CPU bound tasks.
Callback hell issue - asynchronous nature relies heavily on callbacks
…

Node.js Concurrency – EventLoop. EventEmitters



- **Slow I/O Operations** [disks, network resource, ..] can be handled by one of: sync, **event loop**, thread, fork()
- **Event Loop** (single threaded) is an Event Dispatcher.
- Event Loop adds its own queues to be processed by the **libuv thread pool**.
- If task is written one of async ways (Callbacks | Promises | Async/Await), then it will be handled by Event Loop afford.
- EventEmitter is module that facilitates communication between objects. Emitter object emits(spread) named event that causes listeners to be called

```
>node 8-event-emitter.js
```

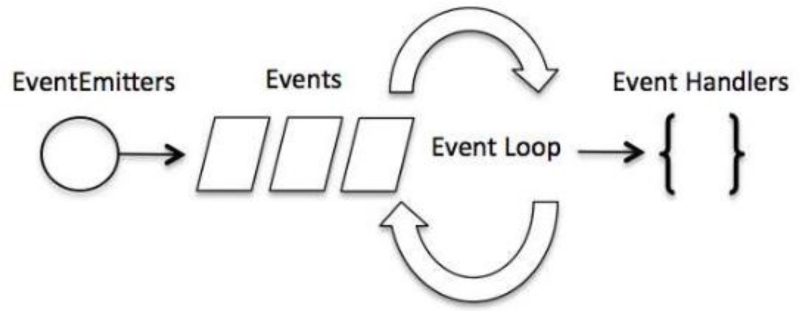
-
  ```
  // Streams are Event Emitters:
  // process.stdin, process.stdout
  ```

Libuv by default creates thread pool with **4** threads max-size is **128**, can be tuned at startup time

Node.js is a single-threaded application, but it can support concurrency via the concept of **event** and **callbacks**. Every API of Node.js is asynchronous and being single-threaded, they use **async function calls** to maintain concurrency.

Event-Driven Programming

In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.



```
// Import events module
var events = require('events');

// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();

// Create an event handler as follows
var connectHandler = function connected() {
   console.log('connection succesful.');

   // Fire the data_received event
   eventEmitter.emit('data_received');
}

// Bind the connection event with the handler
eventEmitter.on('connection', connectHandler);

// Bind the data_received event with the anonymous function
eventEmitter.on('data_received', function() {
   console.log('data received succesfully.');
});

// Fire the connection event
eventEmitter.emit('connection');

console.log("Program Ended.");
```

Although events look quite similar to callbacks, the **difference** lies in the fact that callback functions are called when an asynchronous function returns its result, whereas event handling works on the observer pattern. The functions that listen to events act as **Observers**.

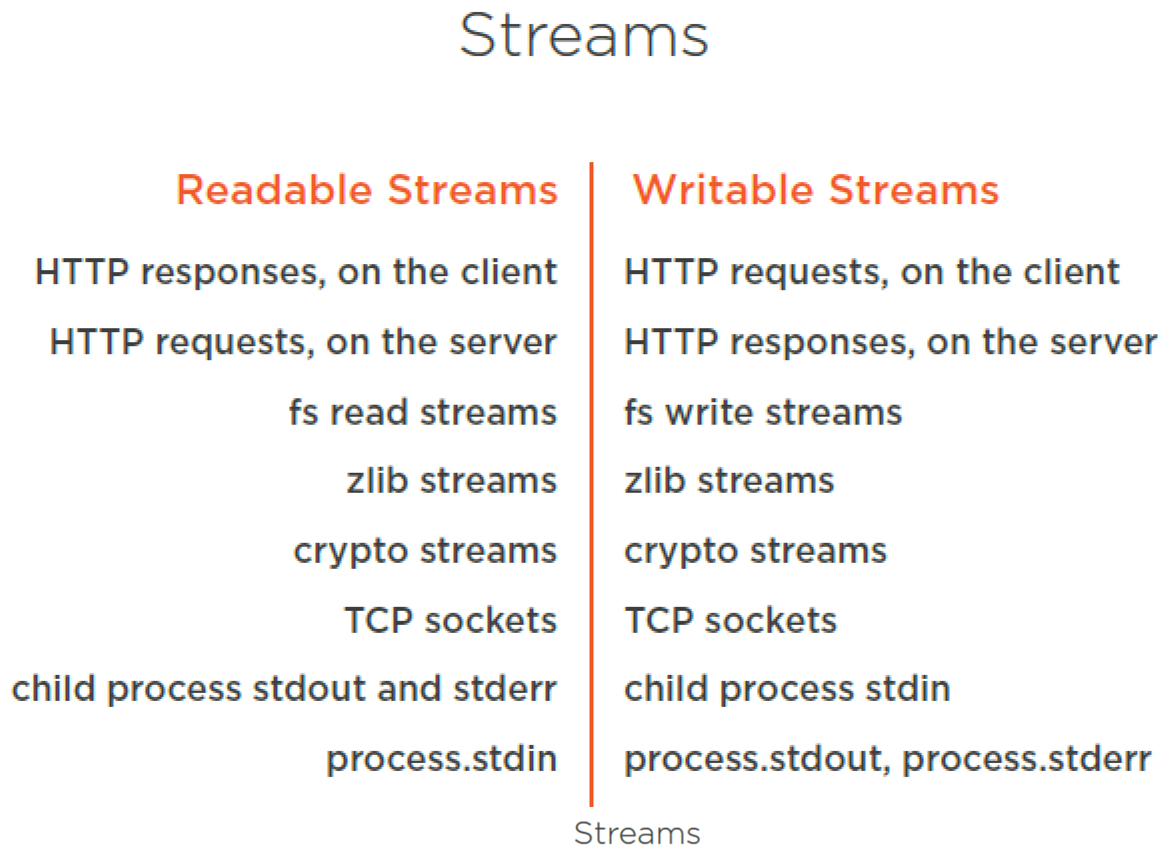
When an **EventEmitter** instance faces any error, it emits an 'error' event. When a new listener is added,
'newListener' event is fired and when a listener is removed, 'removeListener' event is fired.
EventEmitter provides multiple properties like **on** and **emit**. **on** property is used to bind a function with the event and **emit** is used to fire an event.

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

// listener #1
var listner1 = function listner1() {
   console.log('listner1 executed.');
}

// listener #2
var listner2 = function listner2() {
   console.log('listner2 executed.');
}

// Bind the connection event with the listner1 function
eventEmitter.addListener('connection', listner1);

// Bind the connection event with the listner2 function
eventEmitter.on('connection', listner2);

var eventListeners = require('events').EventEmitter.listenerCount
   (eventEmitter,'connection');
console.log(eventListeners + " Listner(s) listening to connection event");

// Fire the connection event
eventEmitter.emit('connection');

// Remove the binding of listner1 function
eventEmitter.removeListener('connection', listner1);
console.log("Listner1 will not listen now.");

// Fire the connection event
eventEmitter.emit('connection');

eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connection');
console.log(eventListeners + " Listner(s) listening to connection event");

console.log("Program Ended.");
```

Streams

are collections of data that might not be available all at once and don't have to fit in memory. All Streams are EventEmitters.

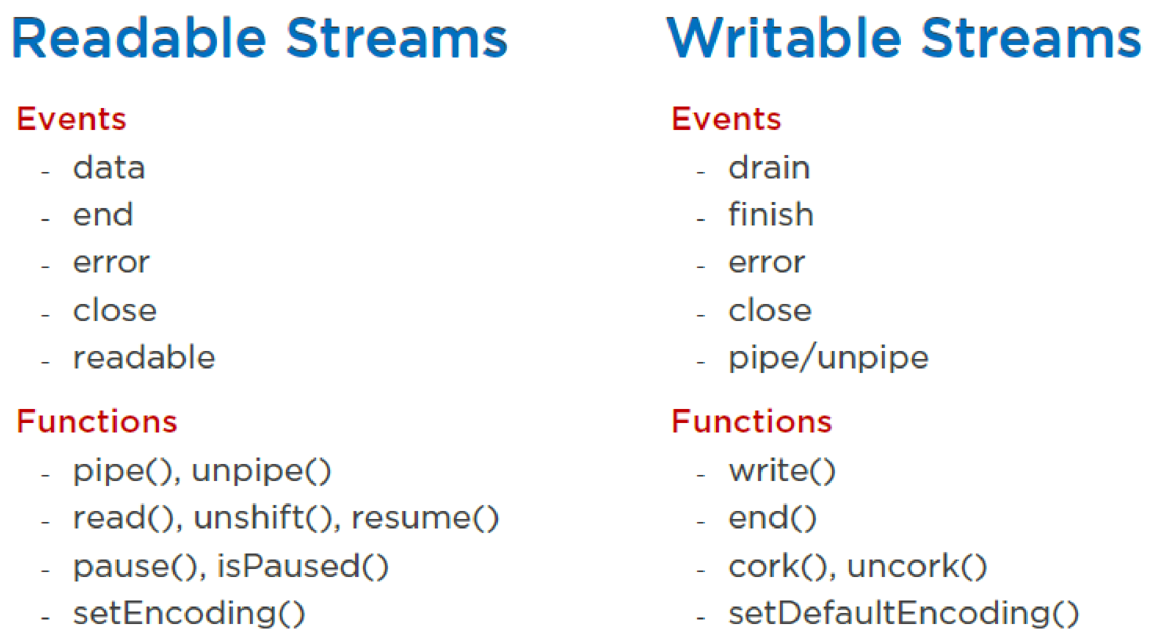
Types of Streams: Readable, Writable, Duplex, Transform



Streams

Readable Streams	Writable Streams
HTTP responses, on the client	HTTP requests, on the client
HTTP requests, on the server	HTTP responses, on the server
fs read streams	fs write streams
zlib streams	zlib streams
crypto streams	crypto streams
TCP sockets	TCP sockets
child process stdout and stderr	child process stdin
process.stdin	process.stdout, process.stderr

Streams



Readable Streams

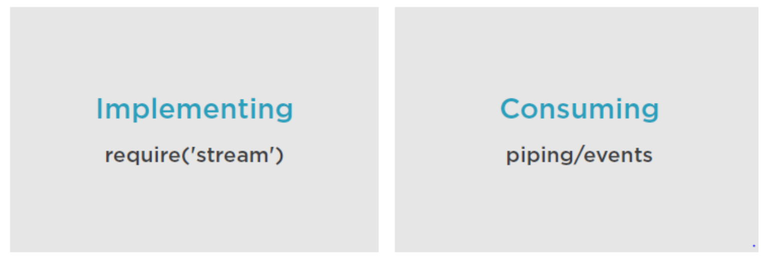
Events
- data
- end
- error
- close
- readable

Functions
- pipe(), unpipe()
- read(), unshift(), resume()
- pause(), isPaused()
- setEncoding()

Writable Streams

Events
- drain
- finish
- error
- close
- pipe/unpipe

Functions
- write()
- end()
- cork(), uncork()
- setDefaultEncoding()



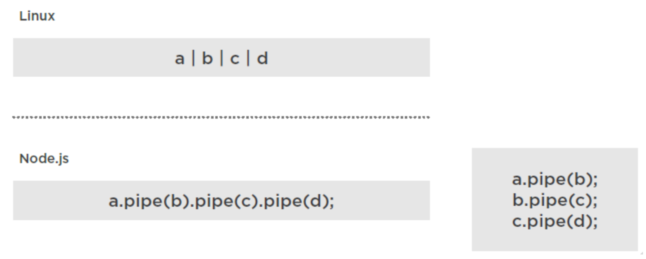
Implementing

require('stream')

Consuming

piping/events

Piping



Linux

```
a | b | c | d
```

Node.js

```
a.pipe(b).pipe(c).pipe(d);
```

```
a.pipe(b);
b.pipe(c);
c.pipe(d);
```

What are Streams?

Streams are objects that let you read data from a source or write data to a destination in continuous fashion. In Node.js, there are four types of streams –

- **Readable** – Stream which is used for read operation.
- **Writable** – Stream which is used for write operation.
- **Duplex** – Stream which can be used for both read and write operation.
- **Transform** – A type of duplex stream where the output is computed based on input.

Each type of Stream is an **EventEmitter** instance and throws several events at different instance of times. For example, some of the commonly used events are –

- **data** – This event is fired when there is data is available to read.
- **end** – This event is fired when there is no more data to read.
- **error** – This event is fired when there is any error receiving or writing data.
- **finish** – This event is fired when all the data has been flushed to underlying system.

Piping the Streams

Piping is a mechanism where we provide the output of one stream as the input to another stream. It is normally used to get data from one stream and to pass the output of that stream to another stream.

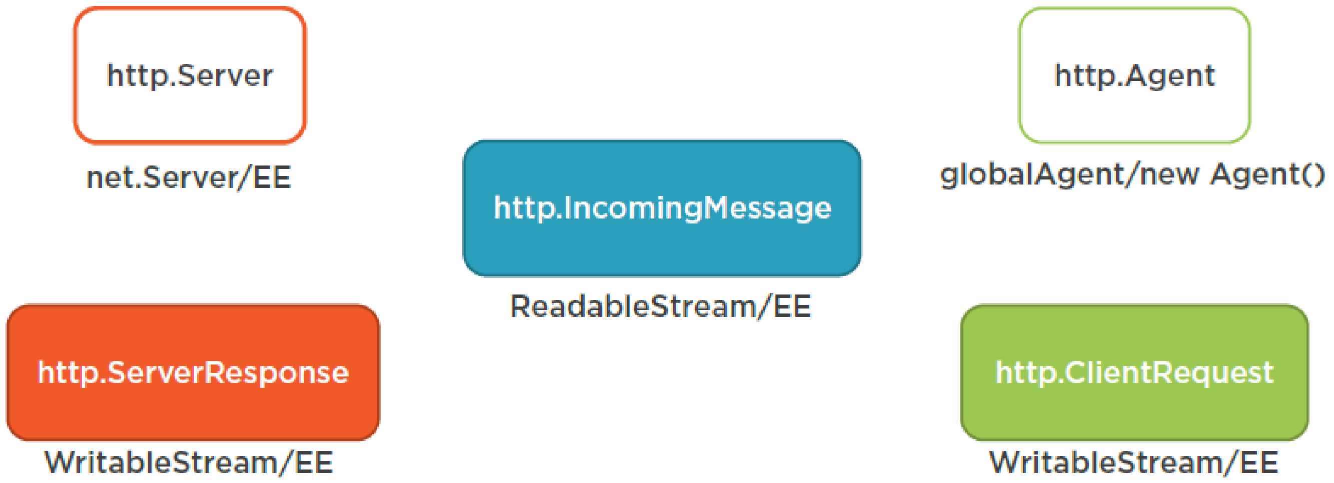
Chaining the Streams

Chaining is a mechanism to connect the output of one stream to another stream and create a chain of multiple stream operations. It is normally used with piping operations.

Working with http/s

- ❑ Streaming Ready HTTP Server
- ❑ Requesting HTTP/HTTPS Data
- ❑ HTTP/HTTPS objects
- ❑ Working with Routes
- ❑ Parsing URLs and Query Strings
- ❑ url/querystring modules
- ❑ req/res both Streams & EE
- ❑ req=Readable, res=Writable

- ❑ Nodemn
- ❑ >npm I –g nodemon
- ❑ >npm server.js (NOOP)
- ❑ >nodemon server.js //monitors, like Spring dev-tool



Web Frameworks, Templates

Web Frameworks:
- Express, koa, sails (inspired by Rails), METEOR, ..

Templates:
- PUG (Jade), handlebars, <% EJS %>, React[JSX]
- //e.g. Java (tiles, velocity, freemarker, thymeleaf, ..)

```
const server = http.createServer((req, res) => {
  //console.log(req);//big REQUEST obj.  IncomingMessage  (2x?)
  console.dir(req, {depth: 0});
   console.dir(req, {depth: 0}); ServerResponse   //status code, headers, body
  res.end("Hello Nodemon !\n");
});
```

Debugger and more utilities

- ❑ Debugger (Node built-in) and more utilities
- ❑ Debug with Chrome dev tools

>node –inspect-brk buggy-file.js
>chrome://inspect   (in Chrome)
Click “inspect” and Ctrl+P

```
// https://nodejs.org/api/http.html#http_class_http_incomingmessage
//no confuse with: https://nodejs.org/api/http.html#http_class_http_clientrequest
```

Node.js Concurrency - Worker Threads

- Before Worker Threads:
- All time consuming tasks are not considered I/O (CPU intensive)
- CPU intensive operations blocks main thread
- Never execute anything from event queue of any pending I/O tasks
- Used child_process | cluster | Napa.js for CPU intensive tasks

- Worker threads introduced in 2018, v12LTS – has stable "worker_thread"
- new Worker(..) - represents an independent JS execution thread
- Each worker owns instance of V8 and EventLoop by V8 isolate.
- Unlike child process or cluster workers share memory
- Creating Worker instance inside of other Worker is possible
- Two-way communication like in like WebWorkers, cluster
- Two ways using workers - (new threads for each incoming task or parent keeps worker live (worker pool))

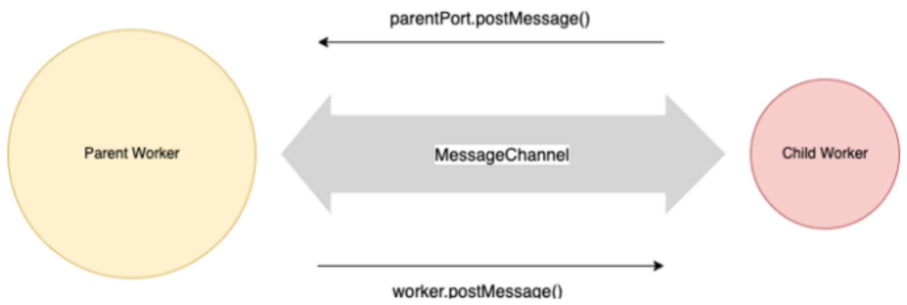


Diagram 1: Message Channel between the parent and the child workers

- Mailer
- Download/upload
- Cypher
- Tabular
- Chalk
- Stringify, …
- REST – like IntelliJ

- packaging
- Cron
- Readline
- V8 context
- Web-stack: mern-mean-mevn
- Jxcore

THANK YOU

References

https://nodejs.org/dist/latest-v16.x/docs/api/
https://www.w3schools.com/nodejs/ref_modules.asp
https://github.com/jscomplete/advanced-nodejs
https://www.npmjs.com/package/az-algos
https://app.pluralsight.com/library/courses/nodejs-advanced/table-of-contents