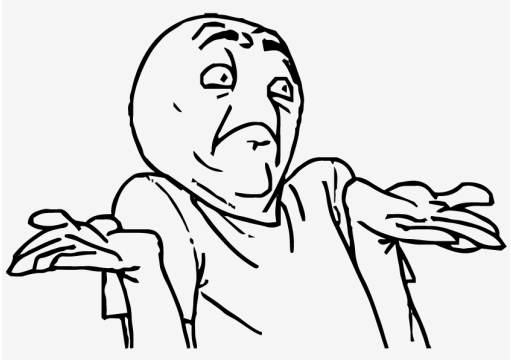
You Don’t know Node.js®

Quick Intro to Core Features



Hengki Sihombing

Co-Founder & CTO at Urbanhire

Agenda

- Why use Node.JS
- How it’s work
- Quick Intro Core Features Node
- Node.js at Urbanhire
- Q & A

Slides & Code

Everything: https://github.com/aredo/tia-pdc-2017

or

Just pdf Slides

What the Hell Node.js is

- JavaScript runtime built on Chrome's V8 JavaScript engine.
- an event-driven
- Asynchronous (libuv)
- Single Threaded but highly Scalable
- Non Buffering

Starting with basics: Why Use Node.js®

Why Node.js

- Speed Development
- Effective single codebase
- It is a good fit for real-time applications
- Node.js app will take up less system RAM
- Because is Asynchronous

Input/output is one of the most expensive type tasks (>CPU)

The cost of I/O ?

- Accessing RAM ~ 250 CPU cycle
- Disk Operations ~ 41 000 000 CPU cycle
  - Read a file
  - Write a File
- Network I/O ~ 240 000 000 CPU cycles
  - Database Query responses
  - Http responses
  - Memcache results

https://goo.gl/aU4TfP

Node has non-blocking I/O

PHP Sleep

```php
<?php

// current time
echo date('h:i:s') . "\n";

// sleep for 10 seconds
sleep(10);

// wake up !
echo date('h:i:s') . "\n";

?>
```

Node.js Sleep

```javascript
console.log('Step: 1')

setTimeout(function () {
  console.log('Step: 3')
}, 1000)

console.log('Step: 2')
```

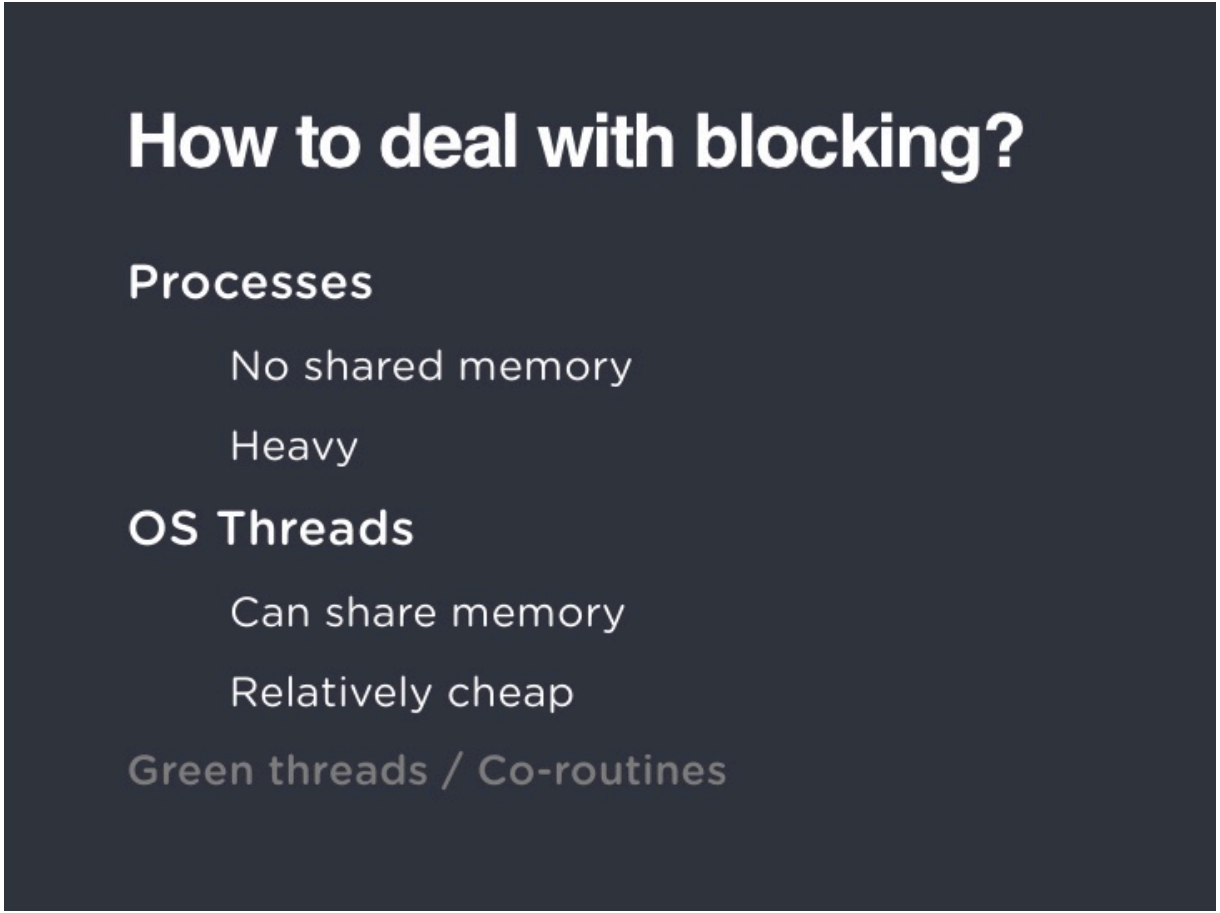
So…, How Node.js works?

Blocking Web Server

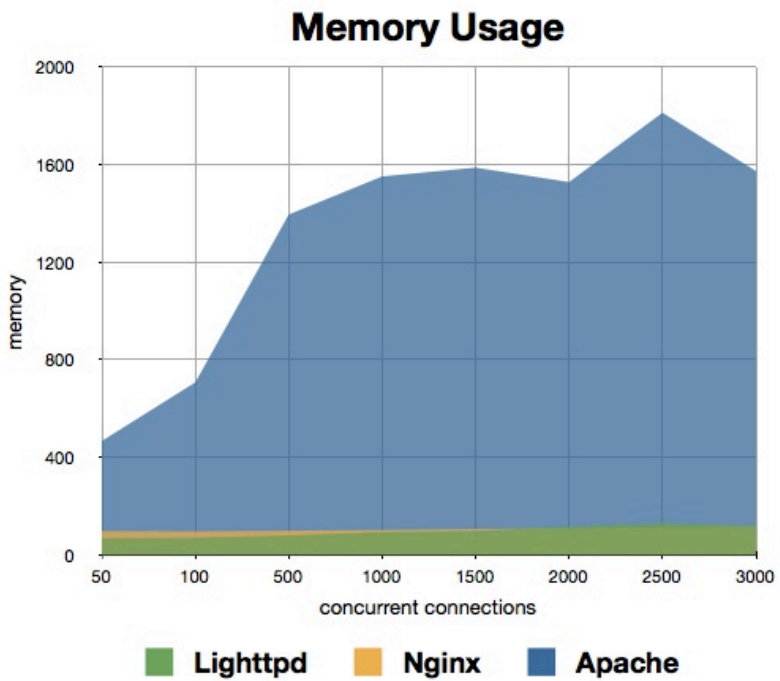

AirAsia
AirAsia

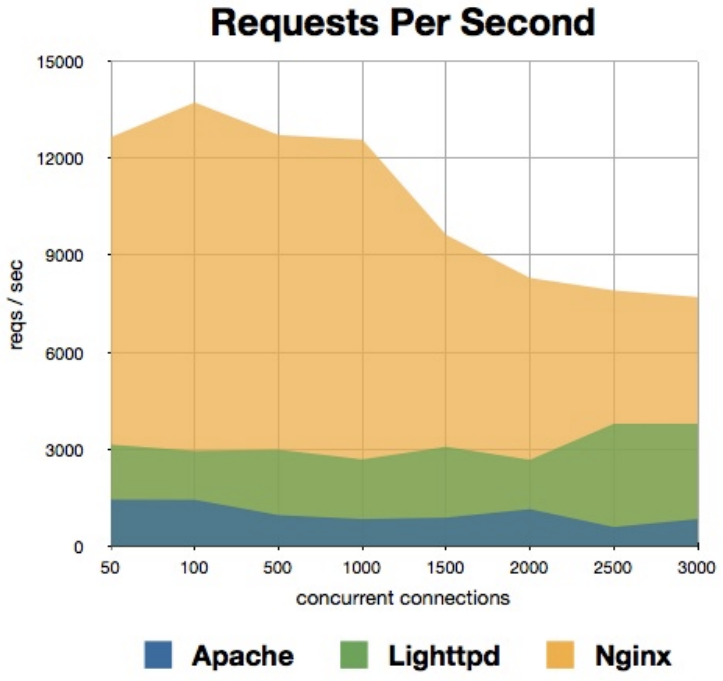
Non-Blocking Web Server

Most web applicantion are I/O Bound not CPU Bound



https://www.slideshare.net/marcusf/nonblocking-io-event-loops-and-nodejs





https://help.dreamhost.com/hc/en-us/articles/215945987-Web-server-performance-comparison

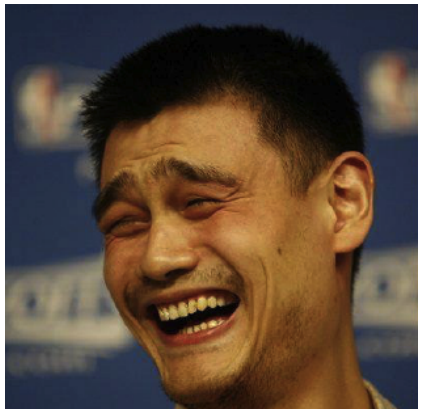
Blocking systems have to be multi-threaded

Thread per connection is Expensive

[Multi-threading] is the software equivalent of a nuclear device because if it is used incorrectly, it can blow up in your face.

https://blog.codinghorror.com/threading-concurrency-and-the-most-powerful-psychokinetic-explosive-in-the-univ

Node is single threaded… and that's good!



Event Loop

JavaScript has a concurrency model based on an "event loop". This model is quite different from models in other languages like C and Java.

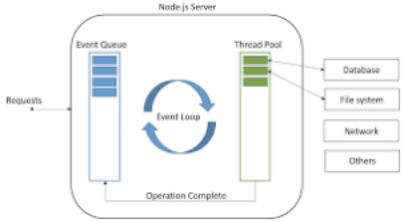
How exactly Event Loop works?

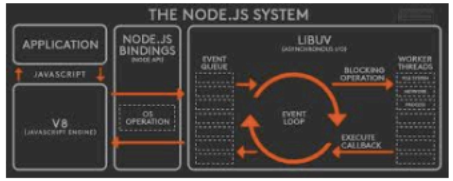
Google

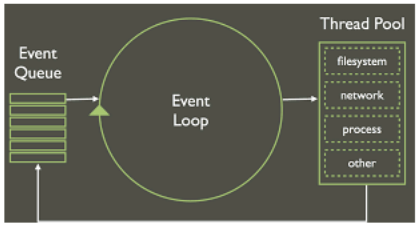
nodejs event loop

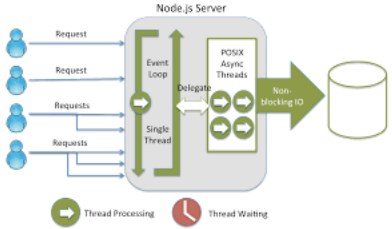
All | Images | Videos | News | Maps | More | Settings | Tools | View saved | SafeSearch

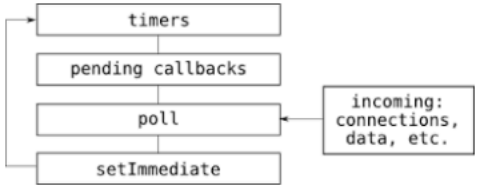
architecture | server | based architecture | nginx | apache | non blocking | javascript | php | python | java | application architecture

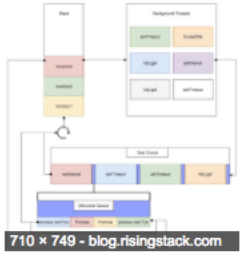


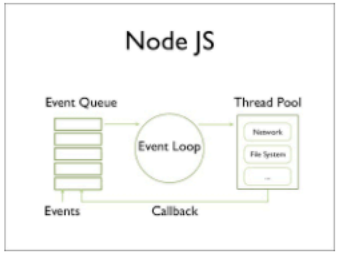


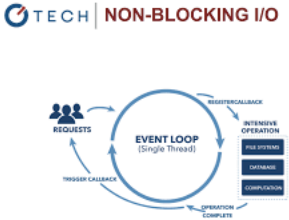


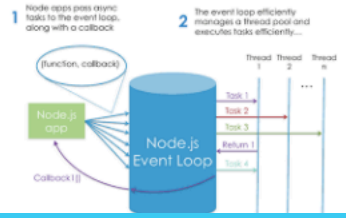


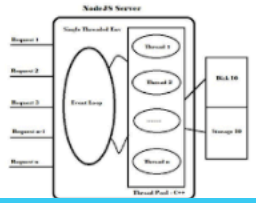


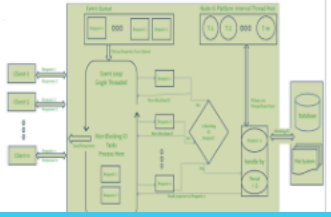


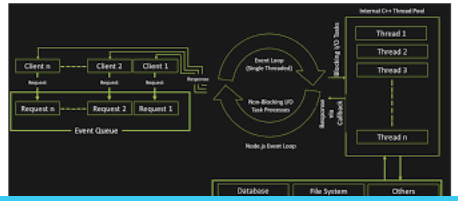


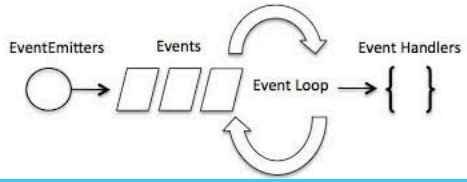












Bert Belder

Libuv & Node.js Core developer

https://www.youtube.com/watch?v=PNa9OMajw9w

Kernel

tcp / udp sockets, servers

unix domain sockets, servers

pipes

tty input

dns.resolveXXXX

Thread Pool

files

fs.*

dns.lookup

pipes (exceptional)

Signal Handler

(posix only)

child processes

signals

Wait Thread

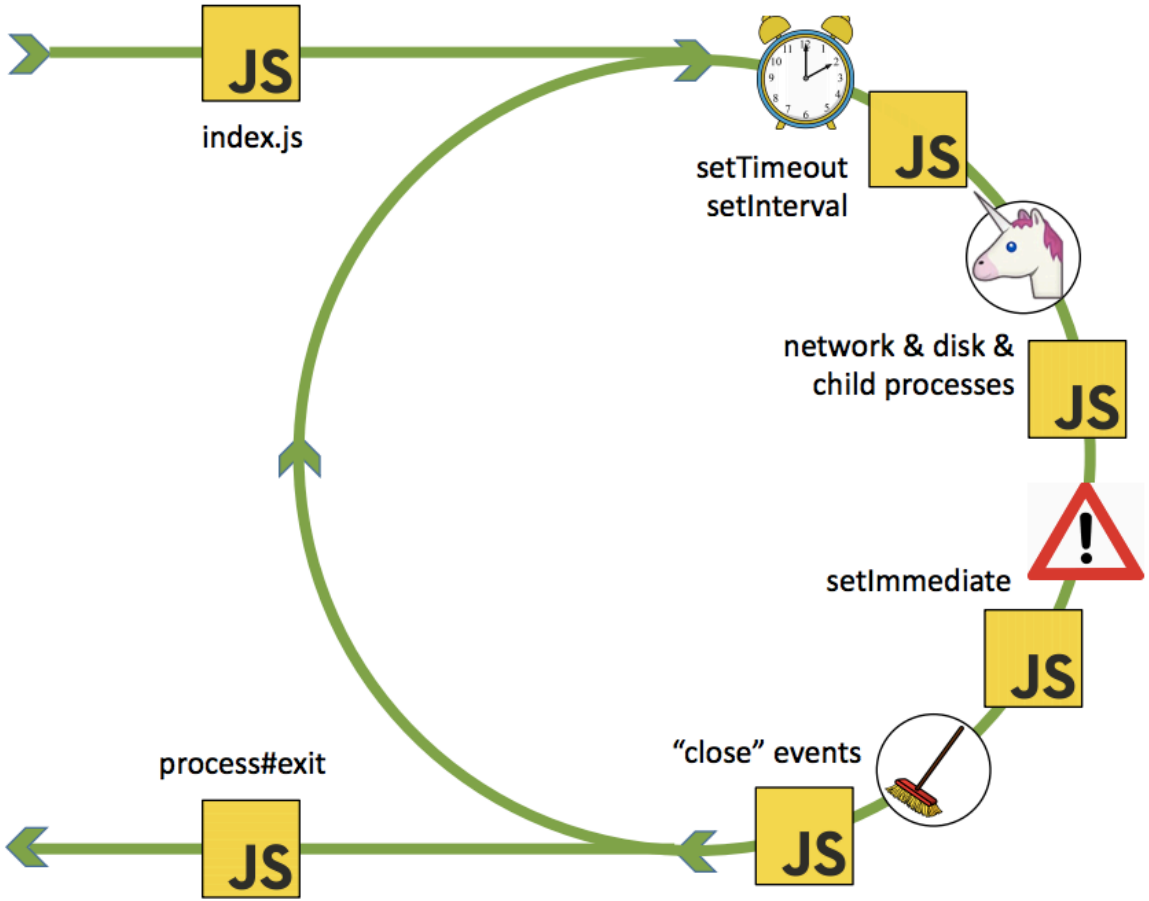
(windows only)

child processes

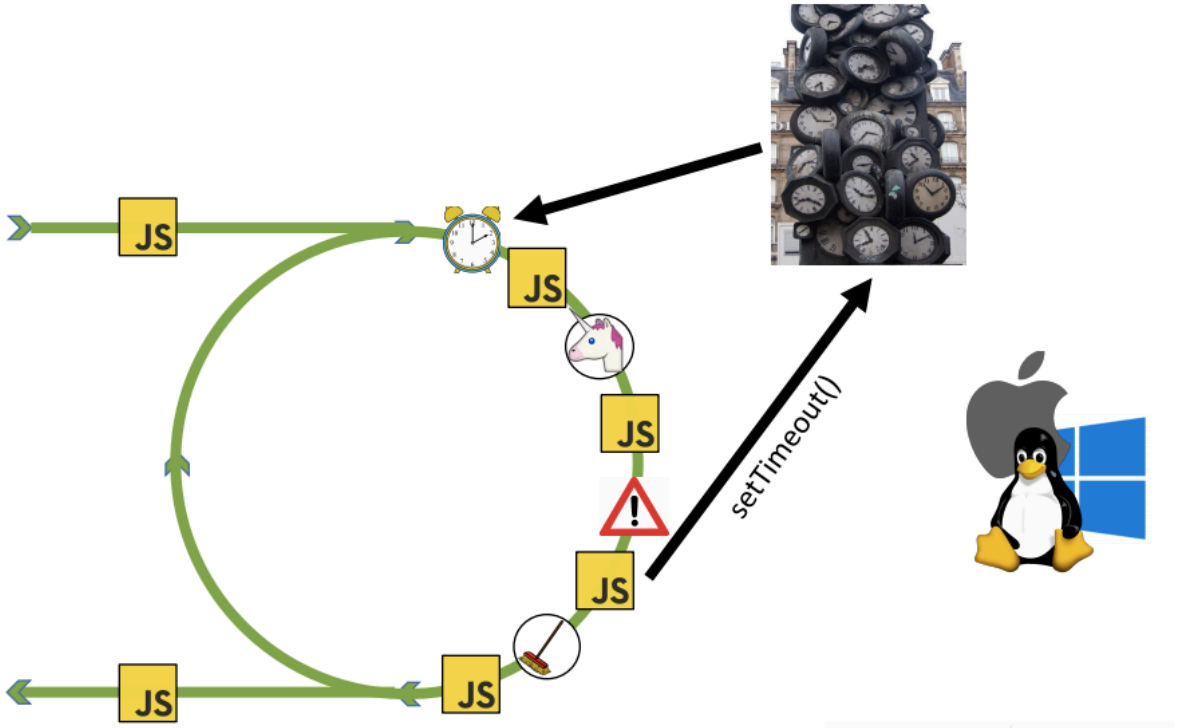
console input

tcp servers (exceptional)

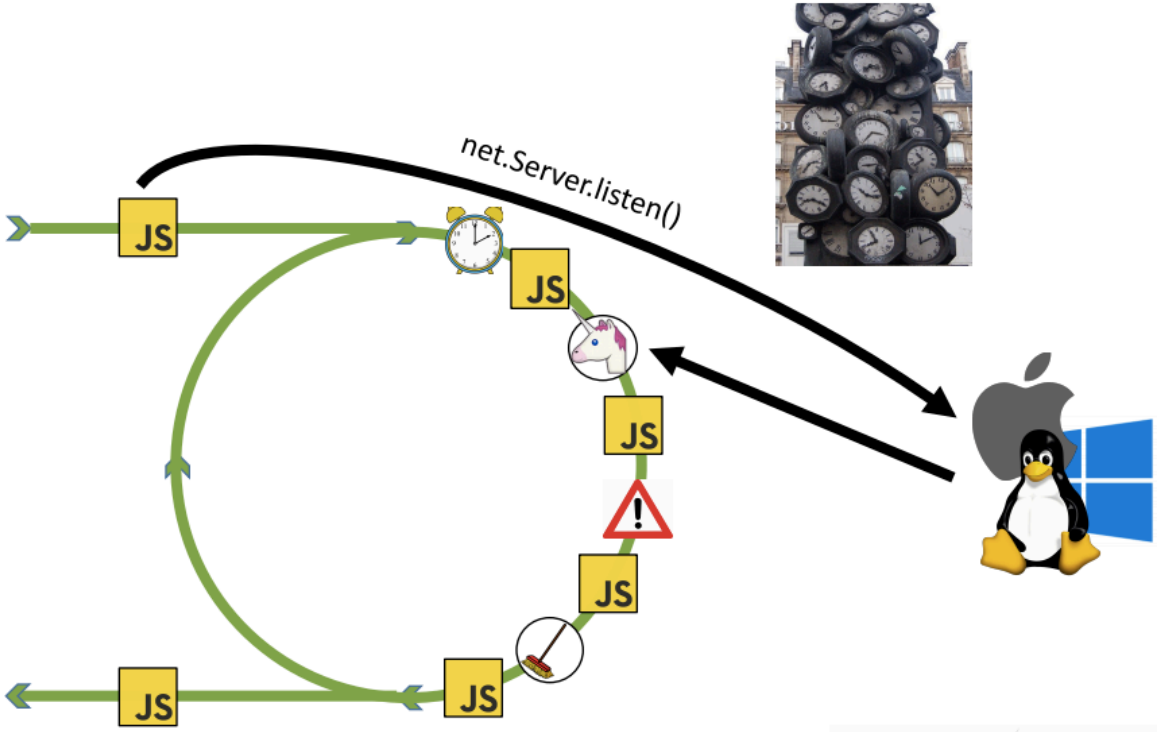
https://goo.gl/aiT9Ga



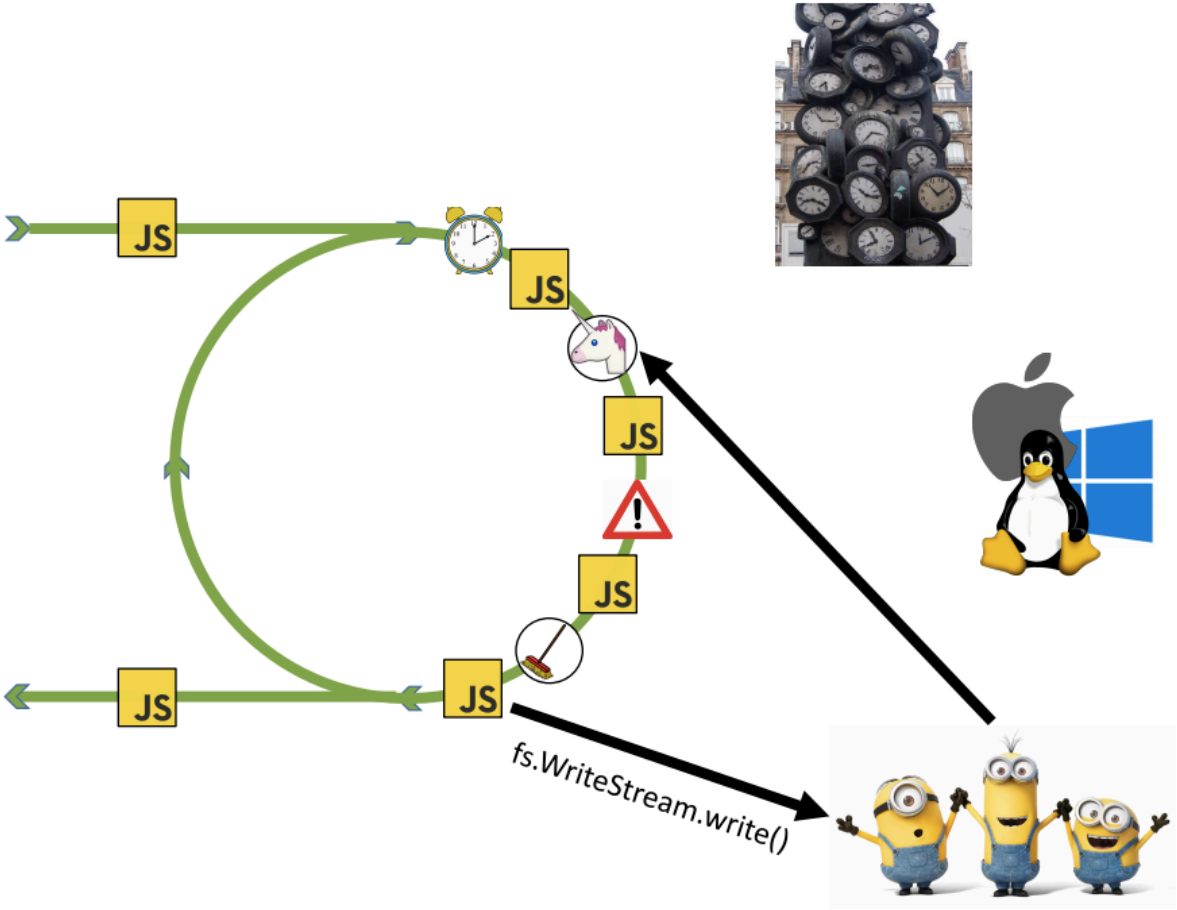
https://www.youtube.com/watch?v=PNa9OMajw9w



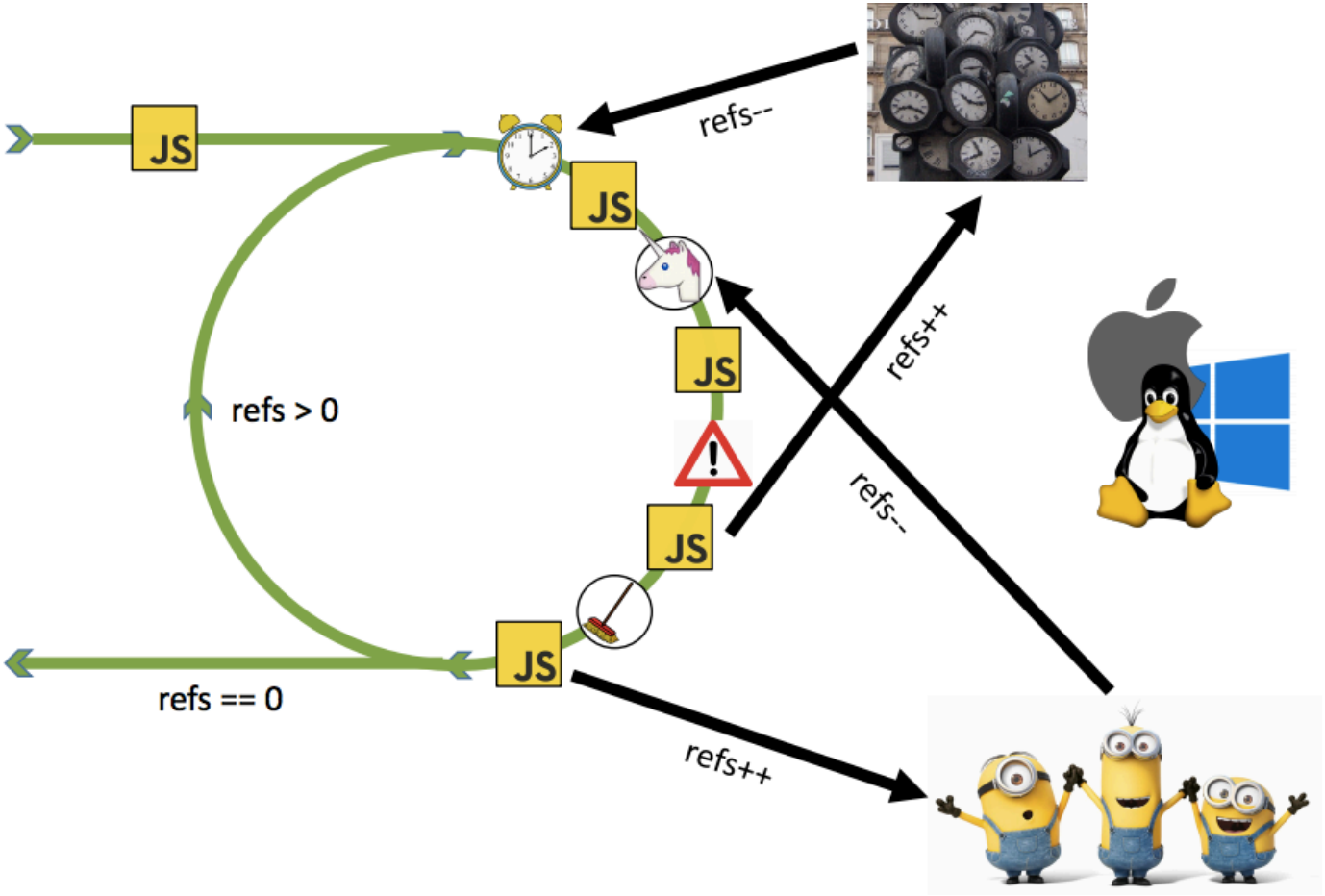
https://www.youtube.com/watch?v=PNa9OMajw9w



https://www.youtube.com/watch?v=PNa9OMajw9w



https://www.youtube.com/watch?v=PNa9OMajw9w



https://www.youtube.com/watch?v=PNa9OMajw9w

Entering Node.js

- Single thread for your code
- … but I/O run in parallel
- Handle thousand of concurrent with single connection
- Need to be careful with CPU-intensive code

But..,
It's still possible to write blocking code in Node.js

Blocking Node.js Code

```
// blocking-code-1.js

const timing = require('./timing')
timing.start(__filename)

console.log('Step: 1')
for (var i = 1; i<1000000000; i++) {
  console.log(i)
  // This will take 100-1000ms
}
console.log('Step: 2')
console.log('Done', timing.finish(__filename).asSeconds() + ' seconds')
```

Blocking Node.js Code

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

var contents = fs.readFileSync('ruby.txt','utf8')
console.log(contents)
console.log('Hello Ruby\n')

var contents = fs.readFileSync('ips.txt','utf8')
console.log(contents)
console.log('Hello Node!')

// ruby.txt -> Hello Ruby -> ips.txt -> Hello Node!
```

Non-Blocking Node.js Code

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

fs.readFile('ruby.txt','utf8', (error, contents) => console.log(contents))
console.log('Hello Ruby\n')

fs.readFile('ips.txt','utf8', (error, contents) => console.log(contents))
console.log('Hello Node!')

// Hello Ruby->Hello Node->... ruby.txt->ips.txt or ips.txt->ruby.txt
```

Most of Node.js is JavaScript

Node.js !== Browser JavaScript

How to create global variables (no window in Node.js) ???

global || GLOBAL

global has properties

global.__filename

global.__dirname

global.module

global.require()

global.process or process

with process, you can do…!!

- Access CLI input?
- Get system info: OS, platform, memory usage, versions, etc.?
- Read *env vars* (passwords!)?

process.env

process.argv

process.pid

process.cwd()

process.exit()

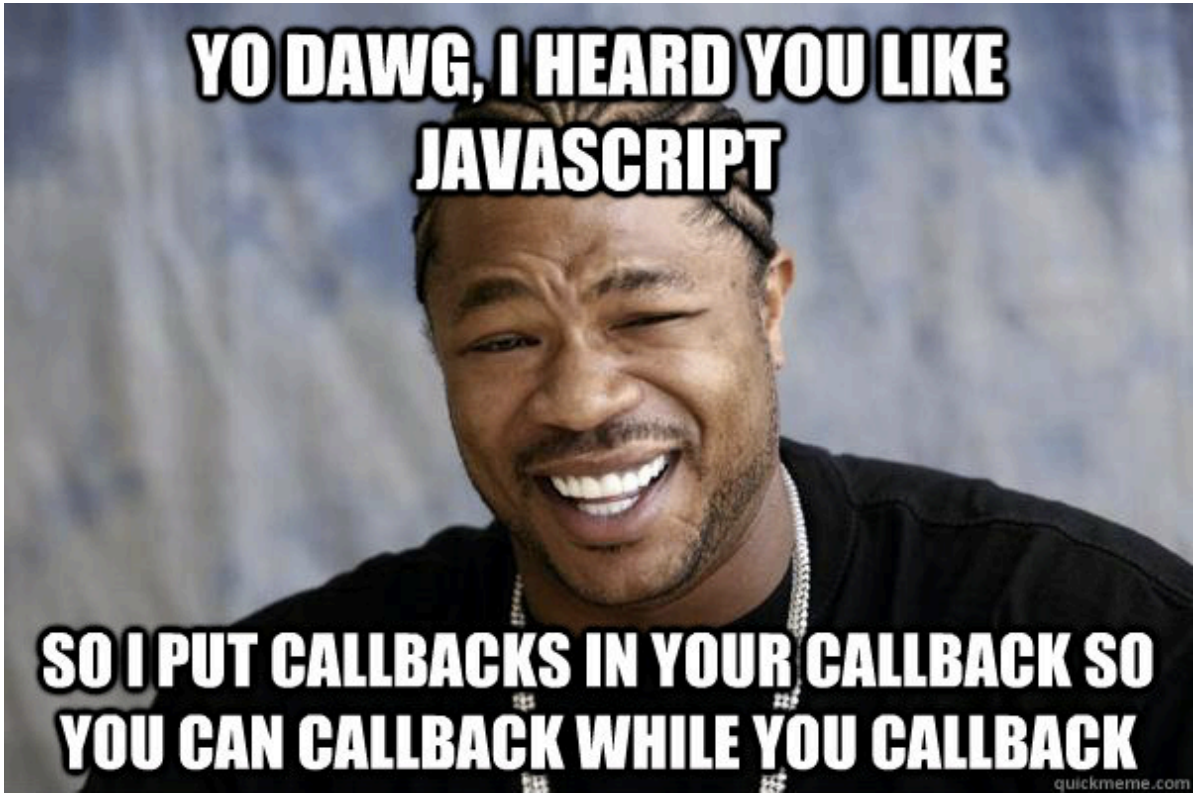
process.kill()

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

Who likes and understands callbacks?

callbackhell.com

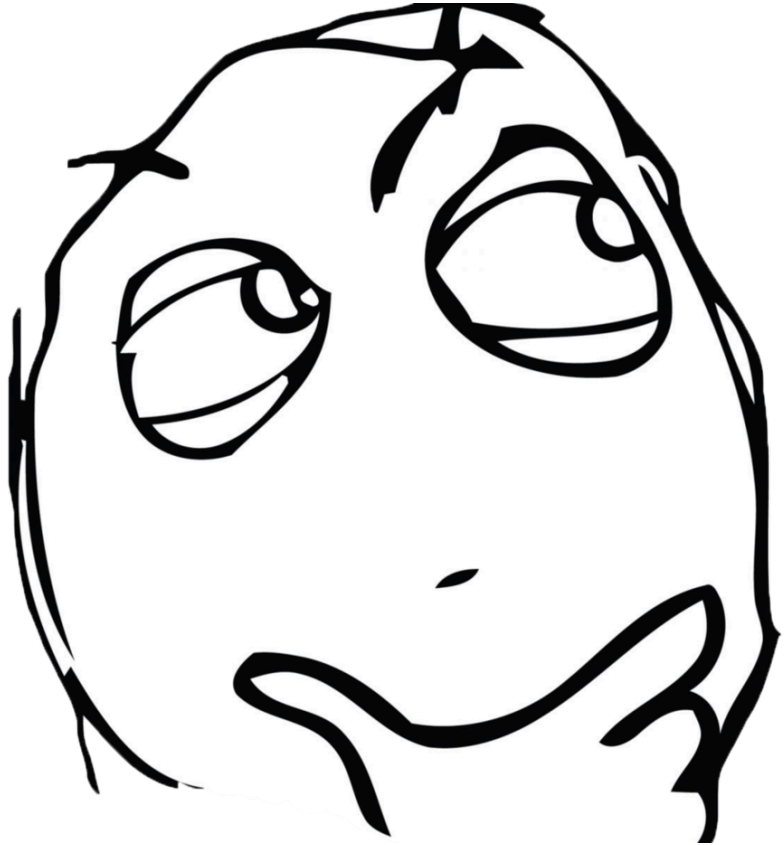
```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```


YO DAWG, I HEARD YOU LIKE JAVASCRIPT
SO I PUT CALLBACKS IN YOUR CALLBACK SO YOU CAN CALLBACK WHILE YOU CALLBACK
quickmeme.com

Handling Callback Hell

- Async.js
- Promises (Native, Bluebird, Q)
- Generatos / co module
- async/wait (Node v7 and v8)

How to handle async errors?



Handling Async Errors

- Event Loop: Async errors are harder to handle/debug, because system loses context of the error. Then, application crashes.
- Try/catch is not good enough, but for sync errors try/catch works fine

Best Practices for Async Errors?

- Listen to all “on error” events
- Listen to uncaughtException
- Use domain (soft deprecated) or AsyncWrap
- Log, log, log & Trace
- Notify (optional)
- Exit & Restart the process

on(‘error’)

Anything that inherits from or creates an instance of the above: Express, LoopBack, Sails, Hapi, etc.

```
server.on('error', function (err) {
  console.error(err)
})
```

uncaughtException

uncaughtException is a very crude mechanism for exception handling. An unhandled exception means your application - and by extension Node.js itself - is in an undefined state. Blindly resuming means anything could happen.

How to deal with **uncaughtException**

- You application shouldn’t have uncaughtexceptions. This is clearly insane.
- You should let your application crash, find uncaught exceptions and fix them. This is clearly insane.
- You should swallow errors silently. This is what lots of people do and it is bad.
- You should let your application crash, log errors and restart your process with something like **pm2**, **forever, upstart** or **monit**. This is pragmatic.
- Testing… Testing.. Testing...

uncaughtException Examples

```
process.on('uncaughtException', function (err) {
  console.error(new Date + ' uncaughtException:', err.message)
  console.error(err.stack)
  process.exit(1)
})
```

Events

Events are part of core and supported by most of the core modules while more advanced patterns such as promises, generators, async/await are not.

Events == Node Observer Pattern

- Subject
- Observers (event listeners) on a subject
- Event triggers

Events in Node.js

```
const events = require('events')

// Create an eventEmitter instance
const eventEmitter = new events.EventEmitter()

// Create an event handler as follows
const connectToTheHandler = function connected() {
   console.log('Test connection was successful.')
   // Fire the data_received_success event
   eventEmitter.emit('data_received_success', { connect: 'success' })
}

// Bind the connection_success event with the handler
eventEmitter.on('connection_success', connectToTheHandler)

// Bind the data_received_success event with the anonymous function
eventEmitter.on('data_received_success', (msg) => {
   console.log('Confirmed that the data has been received successfully.', msg)
})

// Fire the connection_success event
eventEmitter.emit('connection_success')
```

Events in Node.js

```
const request = require('request')
let options = {
  method: 'GET',
  uri: 'https://api.ipify.org'
}

request(options, (error, response, body) => {
  // body is the decompressed response body
  let encoding = response.headers['content-encoding'] || 'identity'
  console.log('server encoded the data as: ' + encoding)
  console.log('the decoded data is: ' + body)
}).on('data', (data) => {
  // decompressed data as it is received
  console.log('decoded chunk: ' + data)
}).on('response', (response) => {
  // unmodified http.IncomingMessage object
  response.on('data', (data) => {
    // compressed data as it is received
    console.log('received ' + data.length + ' bytes of compressed data')
  })
})
```

Listeners

```
emitter.listeners(eventName)
emitter.on(eventName, listener)
emitter.once(eventName, listener)
emitter.removeListener(eventName, listener)
```

Problems with Large Data

- Speed: Too slow because has to load all
- Buffer limit: ~1Gb
- Overhyped

- Every action on a computer is an event. Like when a connection is made or a file is opened
- In node.js an event can be described simply as a string with a corresponding callback.

Stream

Abstractions for continuous chunking of data

a stream is a sequence of data elements made available over time.

https://en.wikipedia.org/wiki/Stream_(computing)

No need to wait for the entire resource to load

Streams Inherit from Event Emitter

Streams are Everywhere!

- HTTP requests and responses
- Standard input/output (stdin&stdout)
- File reads and writes

List native Node.js objects implement the streaming interface

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

Read & Write Steam

```
// writeReadPipeStream.js

const fs = require('fs')
const readableStream = fs.createReadStream('./img/output.jpg')
const writableStream = fs.createWriteStream('./img/output2.jpg')

readableStream.pipe(writableStream)
```

Streams and Buffer Demo

```
// express-stream.js

const largeImagePath = path.join(__dirname,'img/output.jpg')

app.get('/stream', (req, res) => {
  var stream = fs.createReadStream(largeImagePath)
  stream.pipe(res)
})

app.get('/non-stream', (req, res) => {
  fs.readFile(largeImagePath, (error, data) => res.end(data))
})
```

Results

```
~$ http http://localhost:3000/stream
HTTP/1.1 200 OK
Connection: keep-alive
Date: Tue, 08 Aug 2017 18:07:16 GMT
Transfer-Encoding: chunked
X-Powered-By: Express
X-Response-Time: 2.698ms
```

```
~$ http http://localhost:3000/non-stream
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 17875538
Date: Tue, 08 Aug 2017 18:07:02 GMT
X-Powered-By: Express
X-Response-Time: 27.838ms
```

Streams Resources

Stream automated workshop: https://github.com/substack/stream-adventure

$ sudo npm install -g stream-adventure

$ stream-adventure

https://github.com/substack/stream-handbook

How to scale a single threaded system?

Cluster Usage

- Master: starts workers
- Worker: do the job, e.g., HTTP server
- Number of processes = number of CPUs

Cluster

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length

if (cluster.isMaster) {
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
} else {
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('process ' + process.pid + ' says hello!')
  }).listen(8000)
}
```

pm2

https://github.com/Unitech/pm2
http://pm2.keymetrics.io

Advantages:

- **Load-balancer and other features**
- **0s reload down-time, i.e., forever alive**
- **Good test coverage**

Node.js at Urbanhire

99.5% Urbanhire code write with Node.js

Use Case

- Express REST API (Microservices)
- Scraping engine
- Realtime Notification
- Streaming data from MongoDB to Elasticserch
- Queue message with NSQ
- Cronjob

One Last Thing

Codinghorror.com

Atwood's Law

Any application that can be written in JavaScript,
will eventually be writen in JavaScript

Thank You

Be a part our Team

https://www.urbanhire.com/careers