# You Don't Know Node

# You Don't Know Node, v2

## Quick Intro to Core Node

# Turn OFF laptops, phones, distractors and LISTEN

# Who am I? ¯_(ツ)_/¯

# Not Kyle Simpson

(Hint: He wrote "You Don't Know JS" series)

# My name is Azat Mardan and I the creator of Node University

# Azat Mardan and numbers

- 📚 14 books (not counting Korean, Chinese, Polish and Russian translations)
- 🎤 20+ conferences talks in 2016-17
- 👨‍💻 200+ blog posts on Webapplog: https://webapplog.com
- ⌨️ 239 top most active GitHub contributor, higher Paul Irish, Todd Motto, TJ Holowaychuk, John Papa, etc. (source)
- 🎓 19 online courses on Node University https://node.university












# Fun Facts

- Read ~270 books in last 4 years
- Check social media only once per week
- Prefer coffee with butter instead of milk
- Live in San Francisco Bay Area

# I know Node

- Storify - startup which we sold
- DocuSign - 50M users
- Capital One - Fortune 500

# I dislike compiled languages

Actually, just one. Java!

# My Mission is to Make a World a Better Place

# My Plan

Software helps people

+

Node is a good tool to build good software

=

I need to teach as many people Node as possible

# My Goal (for this talk)

Spike your interest in core Node features

# Starting with basics: Why Use Node?

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

# Node has non-blocking I/O




# Overview of Blocking vs Non-Blocking

This overview covers the difference between **blocking** and **non-blocking** calls in Node.js. This overview will refer to the event loop and libuv but no prior knowledge of those topics is required. Readers are assumed to have a basic understanding of the JavaScript language and Node.js callback pattern.

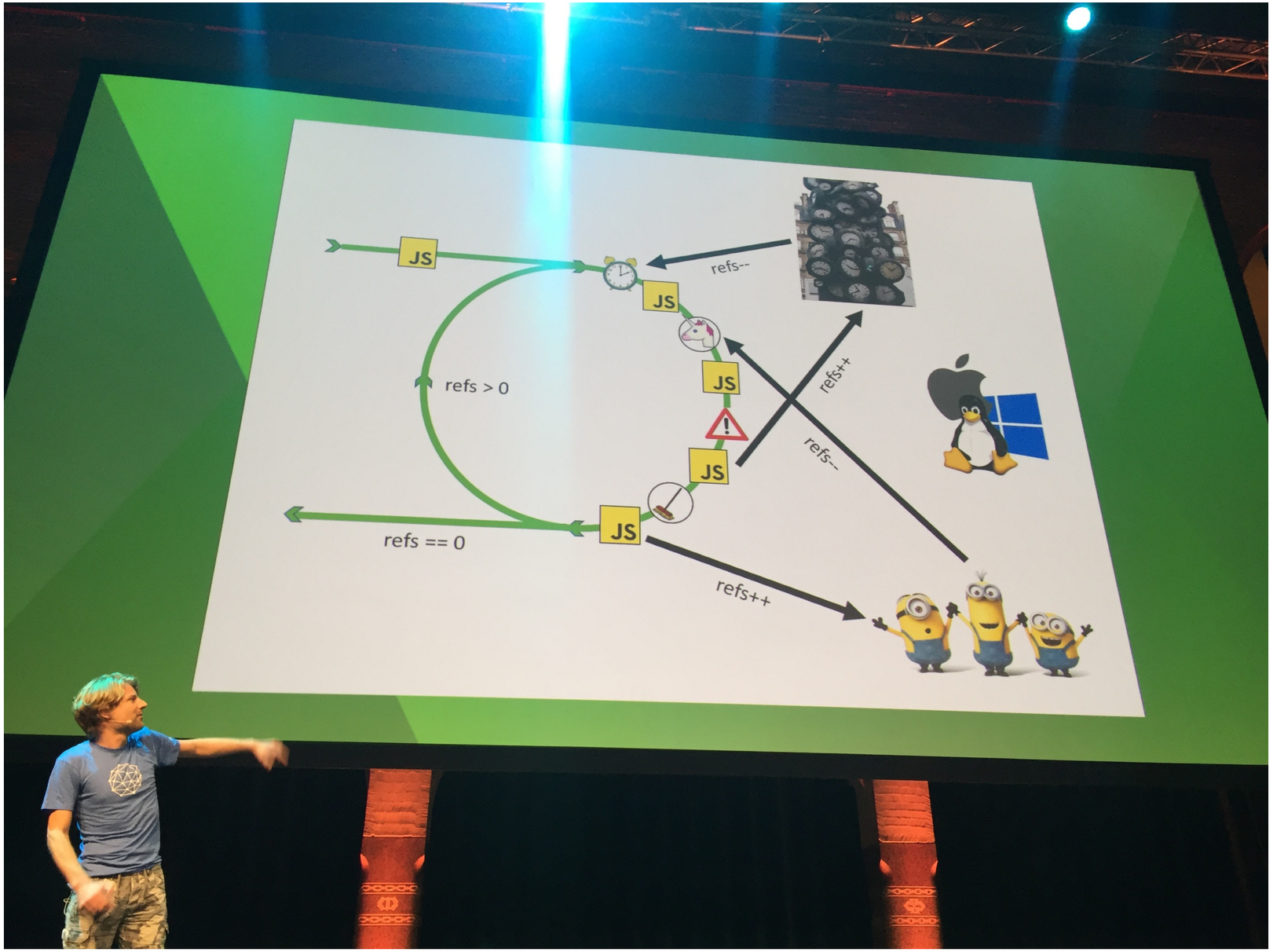
> "I/O" refers primarily to interaction with the system's disk and network supported by libuv.

## Blocking

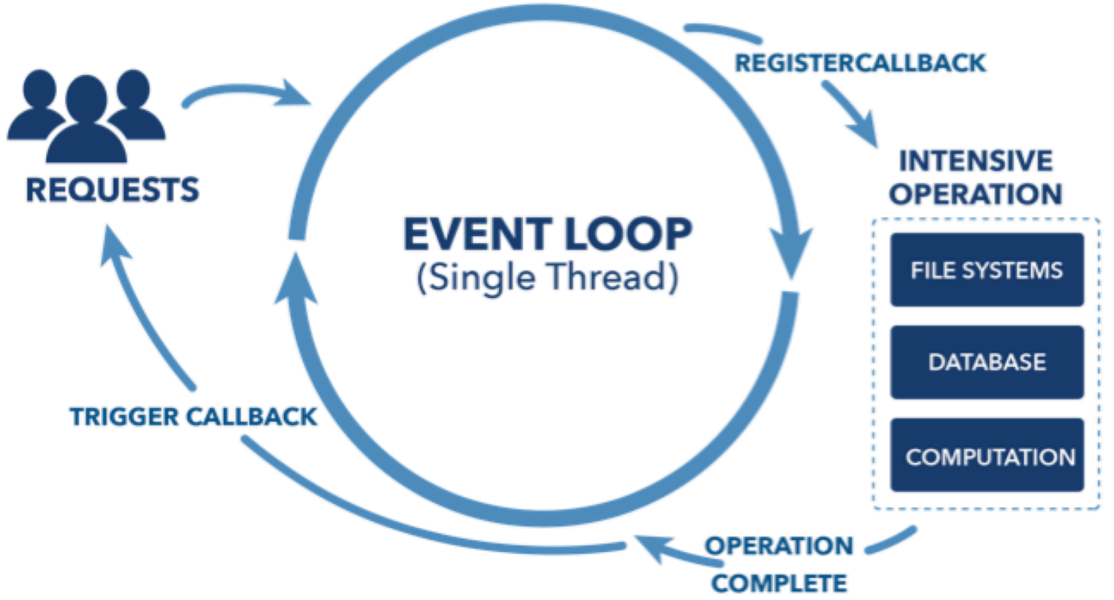
**Blocking** is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes. This happens because the event loop is unable to continue running JavaScript while a **blocking** operation is occurring.

In Node.js, JavaScript that exhibits poor performance due to being CPU intensive rather than waiting on a non-JavaScript operation, such as I/O, isn't typically referred to as **blocking**. Synchronous methods in the Node.js standard library that use libuv are the most commonly used **blocking** operations. Native modules may also have **blocking** methods.

All of the I/O methods in the Node.js standard library provide asynchronous versions, which are **non-blocking**, and accept callback functions. Some methods also have **blocking** counterparts, which have names that end with `Sync`.



# TECH | NON-BLOCKING I/O



# Java Sleep

```
System.out.println("Step: 1");
System.out.println("Step: 2");
Thread.sleep(1000);
System.out.println("Step: 3");
```

# Node "Sleep"

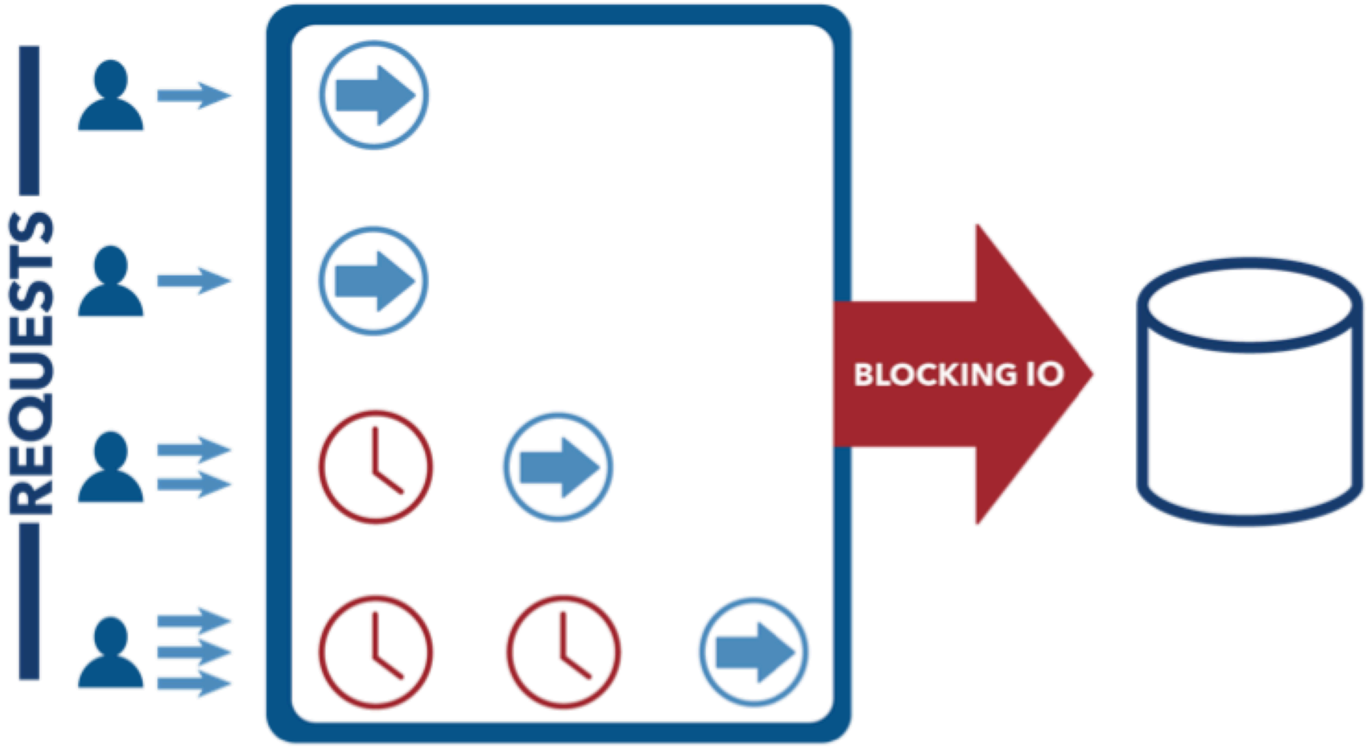
```
console.log('Step: 1')
setTimeout(function () {
  console.log('Step: 3')
}, 1000)
console.log('Step: 2')
```

# Process Multiple Tasks

```
console.log('Step: 1')
setTimeout(function () {
  console.log('Step: 3')
  // console.log('Step 5')
}, 1000);
console.log('Step: 2')
// console.log('Step 4')
```

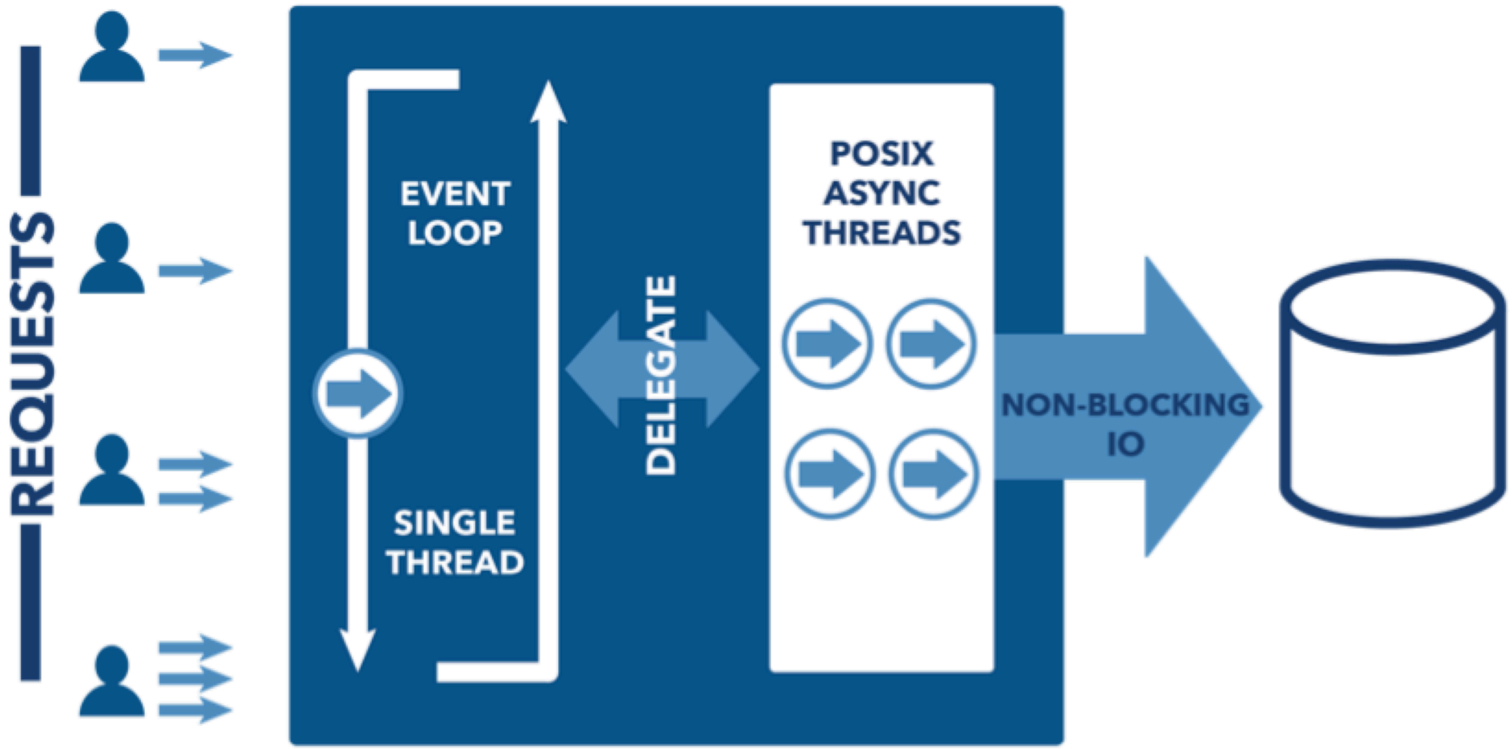
# Blocking Web Server

# BLOCKING I/O



# Non-Blocking Web Server

# TECH | NON-BLOCKING I/O



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

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

# Blocking systems have to be multi-threaded

# Node is single threaded... and that's good! 😄

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

# Blocking Node.js Code

```
// blocking.js
console.log('Step: 1')
for (var i = 1; i<1000000000; i++) {
  // This will take 100-1000ms
}
console.log('Step: 2')
```

# Blocking Node.js Code

```
var fs = require('fs')

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

var contents = fs.readFileSync('ips.txt','utf8')
console.log(contents)
console.log('Hello Node!')
//accounts.txt->Hello Ruby->ips.txt->Hello Node!
```

# Non-Blocking Node.js Code

```
var fs = require('fs')

fs.readFile('accounts.txt','utf8', function(error, contents){
   console.log(contents)
})
console.log('Hello Ruby\n')

fs.readFile('ips.txt','utf8', function(error, contents){
   console.log(contents)
})
console.log('Hello Node!')
//Hello Ruby->Hello Node->... accounts.txt->ips.txt or ips.txt->accounts.txt
```

# Node *typically* is much faster than other platforms

How many of you reach the performance limitations of apps built with blocking I/O systems?

# Probably not many

# My Fav Node Benefit

# JavaScript everywhere. One language to rule 'em all!

# When one language is used everywhere

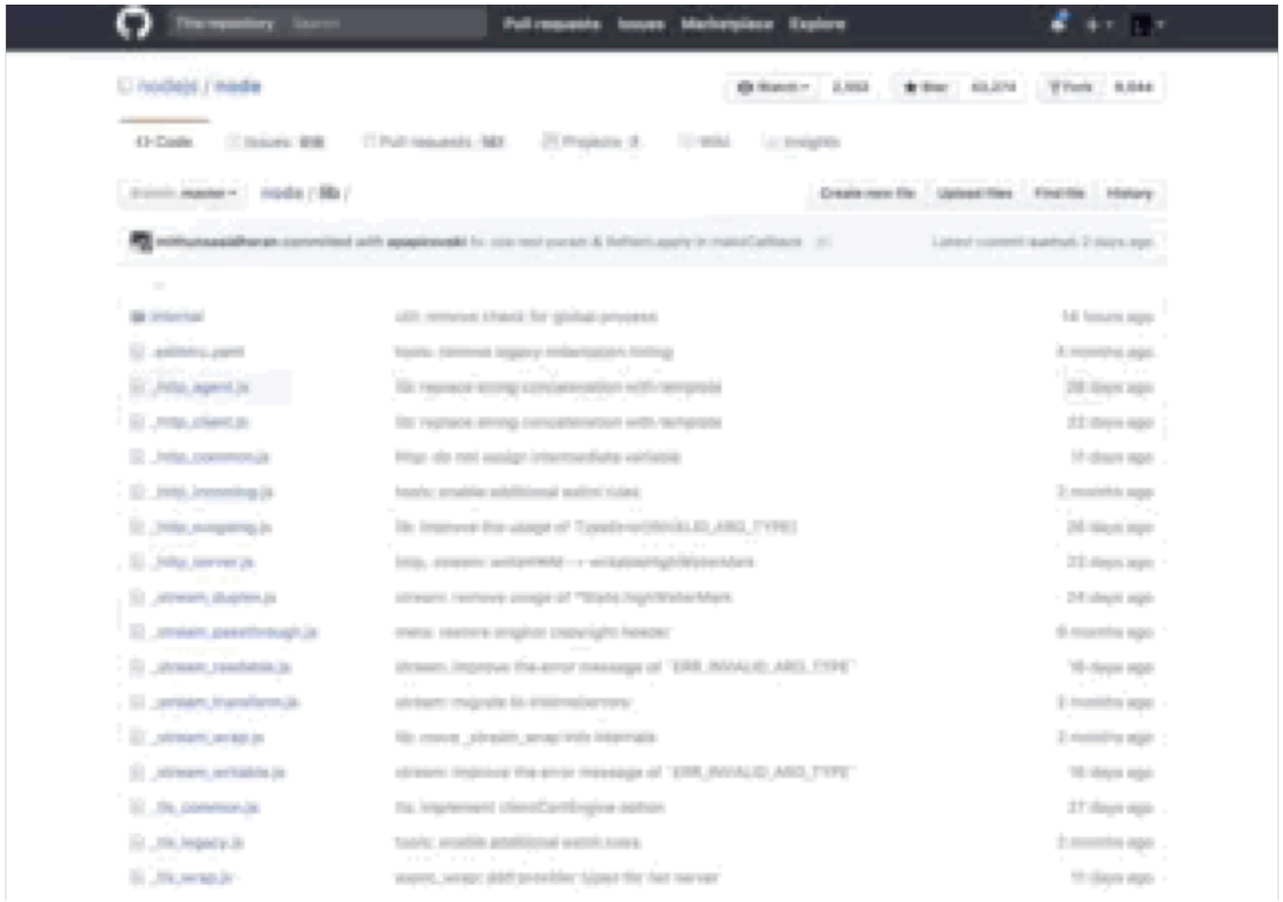
- Think faster
- Reuse code
- Learn quicker

# Most of Node is JavaScript

- Array
- String
- Primitives
- Functions
- Objects

# Node !== Browser JavaScript

# Node Core Modules



# How to create global variables (no `window` in Node), work with modules, get path to my script?

# `global`

Now available everywhere in your code

# It has properties!

```
global.__filename

global.__dirname
```

```
global.module
global.require()
```

# How do I...?

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

`global.process` or `process`

```
process.pid

process.versions

process.arch
```

# process.argv

# process.env

```
process.uptime()

process.memoryUsage()
```

```
process.cwd()
```

```
process.exit()
process.kill()
```

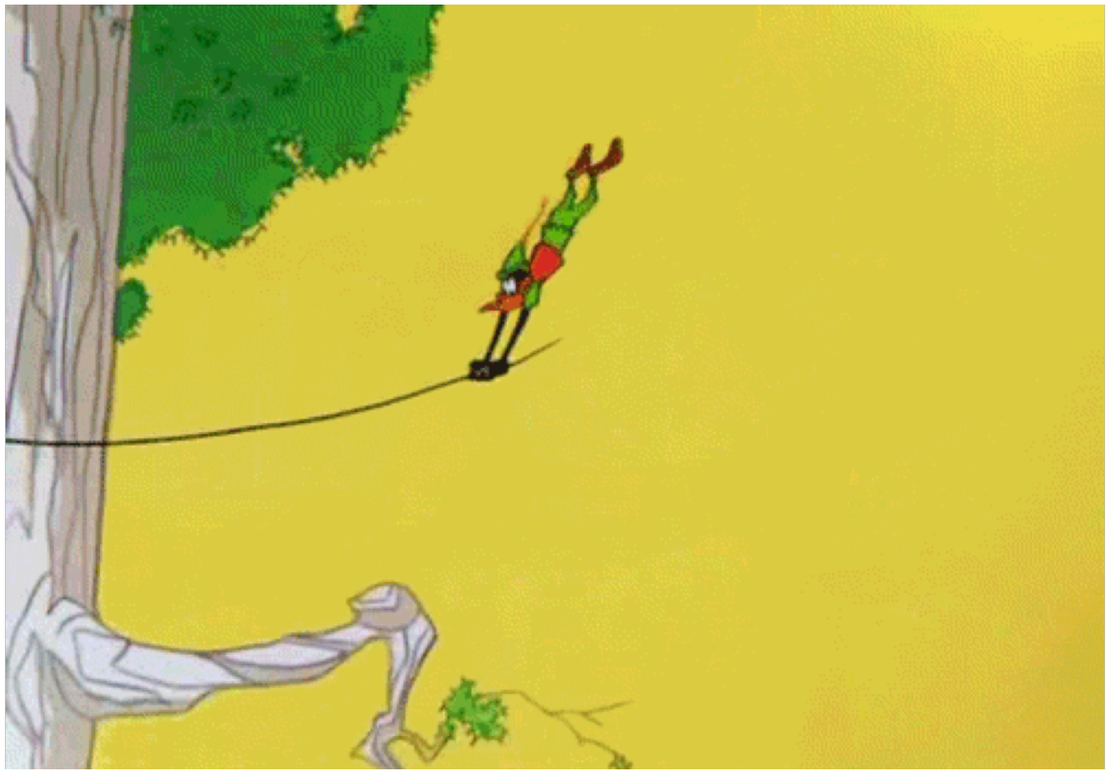
# Who likes and understands callbacks? 🙋‍♀️

# http://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))
        }
      })
    })
  }
})
```

Callbacks are not very developmental scalable 😞

# Me When Working With Deeply Nested Callbacks



# 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

```
var events = require('events')
var emitter = new events.EventEmitter()
```

# Events

In node.js an event can be described simply as a string with a corresponding callback.

```
emitter.on('done', function(results) {
  console.log('Done: ', results)
})
```

# Using Event Emitters

```
var events = require('events')
var emitter = new events.EventEmitter()

emitter.on('knock', function() {
  console.log('Who\'s there?')
})

emitter.on('knock', function() {
  console.log('Go away!')
})

emitter.emit('knock')
```

# Inheriting from EventEmitter

```
// job.js
var util = require('util')
var Job = function Job() {
  // ...
  this.process = function() {
    // ...
    job.emit('done', { completedOn: new Date() })
  }
}

util.inherits(Job, require('events').EventEmitter)
module.exports = Job
```

# Inheriting from EventEmitter

```
// weekly.js
var Job = require('./job.js')
var job = new Job()

job.on('done', function(details){
  console.log('Job was completed at', details.completedOn)
  job.removeAllListeners()
})

job.process()
```

# Listeners

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

# Resources to Learn Node Patterns

- Node Patterns: From Callbacks to Observer: http://webapplog.com/node-patterns-from-callbacks-to-observer
- https://github.com/azat-co/node-patterns
- Node.js Design Patterns, Second Edition by Mario Casciaro, Luciano Mammino

# Problems with Large Data

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

# Streams

## Abstractions for continuous chunking of data

# No need to wait for the entire resource to load

# Types of Streams

- Readable
- Writable
- Duplex
- Transform

# Streams Inherit from Event Emitter

# Streams are Everywhere!

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

# Readable Stream Example

`process.stdin`

Standard input streams contain data going into applications.

This is achieved via a read operation.

Input typically comes from the keyboard used to start the process.

To listen in on data from stdin, use the `data` and `end` events:

```
// stdin.js
process.stdin.resume()
process.stdin.setEncoding('utf8')

process.stdin.on('data', function (chunk) {
  console.log('chunk: ', chunk)
})

process.stdin.on('end', function () {
  console.log('--- END ---')
})
```

# Demo

```
$ node stdin.js
```

# New Interface `read()`

```
var readable = getReadableStreamSomehow()
readable.on('readable', () => {
  var chunk
  while (null !== (chunk = readable.read())) {
    console.log('got %d bytes of data', chunk.length)
  }
})
```

# Writable Stream Example

`process.stdout`

Standard output streams contain data going out of the applications.

This is done via a write operation.

Data written to standard output is visible on the command line.

# Writable Stream

To write to `stdout`, use the `write` function:

```
process.stdout.write('A simple message\n')
```

# What about HTTP?

```
const http = require('http')
var server = http.createServer( (req, res) => {
  req.setEncoding('utf8')
  req.on('data', (chunk) => {
    transform(chunk) // This functions is defined somewhere else
  })
  req.on('end', () => {
    var data = JSON.parse(body)
    res.end()
  })
})

server.listen(1337)
```

# Pipe

```
var r = fs.createReadStream('file.txt')
var z = zlib.createGzip()
var w = fs.createWriteStream('file.txt.gz')
r.pipe(z).pipe(w)
```

# What data type to use for binary data?

# Buffers

Binary data type, to create:

- `Buffer.alloc(size)`
- `Buffer.from(array)`
- `Buffer.from(buffer)`
- `Buffer.from(str[, encoding])`

Docs: http://bit.ly/1IeAcZ1

# Working with Buffer

```
// buf.js
var buf = Buffer.alloc(26)
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97 // 97 is ASCII a
}
console.log(buf) // <Buffer 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76 77 78 79 7a>
console.log(buf.toString('utf8')) // abcdefghijklmnopqrstuvwxyz
```

# Buffer Convertion

```
buf.toString('ascii') // outputs: abcdefghijklmnopqrstuvwxyz
buf.toString('ascii', 0, 5) // outputs: abcde
buf.toString('utf8', 0, 5) // outputs: abcde
buf.toString(undefined, 0, 5) // encoding defaults to 'utf8', outputs abcde
```

# Remember fs?

```
fs.readFile('/etc/passwd', function (err, data) {
  if (err) return console.error(err)
  console.log(data)
});
```

`data` is buffer!

# Demo

```
$ node server-stream
```

# Streams and Buffer Demo

```
// server-stream.js
app.get('/stream', function(req, res) {
  var stream = fs.createReadStream(largeImagePath)
  stream.pipe(res)
})

$ node server-stream
```

http://localhost:3000/stream
http://localhost:3000/non-stream

# Results in DevTools

`/stream` responds faster!

`X-Response-Time`
`~300ms vs. 3-5s`

# Stream 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

# Clusters

```
var cluster = require('cluster')
var numCPUs = require('os').cpus().length
if (cluster.isMaster) {
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork()
  }
} else if (cluster.isWorker) {
  // your server code
})
```

# Cluster Demo

1. Run `code/cluster.js` with node (`$ node cluster.js`).
2. Install `loadtest` with npm: `$ npm install -g loadtest`
3. Run load testing with: `$ loadtest  http://localhost:3000 -t 20 –c 10`

Press control+c on the server terminal

# Cluster Libraries

- Core cluster: lean and mean
- strong-cluster-control (https://github.com/strongloop/strong-cluster-control), or `$ slc run`: good choice
- pm2 (https://github.com/Unitech/pm2): good choice

# 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

# pm2 Demo: Typical Express Server

```
var express = require('express')
var port = 3000
global.stats = {}
console.log('worker (%s) is now listening to http://localhost:%s',
 process.pid, port)
var app = express()
app.get('*', function(req, res) {
  if (!global.stats[process.pid]) global.stats[process.pid] = 1
  else global.stats[process.pid] += 1;
  var l ='cluser '
    + process.pid
    + ' responded \n';
  console.log(l, global.stats)
  res.status(200).send(l)
})
app.listen(port)
```

# pm2 Demo

Using `server.js`:

```
$ pm2 start server.js -i 0
```

In a new window:

```
$ loadtest  http://localhost:3000 -t 20 -c 10
$ pm2 list
```

# Spawn vs Fork vs Exec

- `require('child_process').spawn()` - large data, stream, no new V8 instance
- `require('child_process').fork()` - new V8 instance, multiple workers
- `require('child_process').exec()` - buffer, async, all the data at once

# Spawn Example

```
fs = require('fs')
process = require('child_process')
var p = process.spawn('node', 'program.js')
p.stdout.on('data', function(data)) {
  console.log('stdout: ' + data)
})
```

# Fork Example

```
fs = require('fs')
process = require('child_process')
var p = process.fork('program.js')
p.stdout.on('data', function(data)) {
  console.log('stdout: ' + data)
})
```

# Exec Example

```
fs = require('fs')
process = require('child_process')
var p = process.exec('node program.js', function (error, stdout, stderr) {
  if (error) console.log(error.code)
})
```

# There are few more methods

- child_process.execFile()
- child_process.execSync()
- child_process.execFileSync()

# 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.

# Synchronous Error in Node

```
try {
  throw new Error('Fail!')
} catch (e) {
  console.log('Custom Error: ' + e.message)
}
```

For sync errors try/catch works fine.

# Async Error Example

```
try {
  setTimeout(function () {
    throw new Error('Fail!')
  }, Math.round(Math.random()*100))
} catch (e) {
  console.log('Custom Error: ' + e.message)
}
```

The app crashes!

# Me When Async Error's Thrown

# Async Errors

How to deal with it?

# 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)
})
```

# on('error') Chained Method Example

```
var http = require(‘http’)
var server = http.createServer(app)
  .on('error', function(e) {
    console.log(‘Failed to create server’)
    console.error(e)
    process.exit(1)
  })
```

# on(‘error’) Named Variable Example

```
var req = http.request(options, function(res) {
  // … processing the response
})

req.on('error', function(e) {
  console.log('problem with request: ' + e.message)
})
```

# 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.

# uncaughtException

Always listen to `uncaughtException`!

```
process.on(‘uncaughtException’, handle)
```

or

```
process.addListener('uncaughtException', handle)
```

# uncaughtException Expanded Examples

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

or

```
process.addListener('uncaughtException', function (err) {
  console.error('uncaughtException: ', err.message)
  console.error(err.stack)
  process.exit(1)
```

# Domain

(Just out of curiosity)

# Domain Example

```
let domain = require('domain').create()
domain.on('error', function(error){
  console.log(error)
})
domain.run(function(){
  throw new Error('Failed!')
})
```

# Domain with Async Error Demo

## domain-async.js:

```
let d = require('domain').create()
d.on('error', function(e) {
    console.log('Custom Error: ' + e)
})
d.run(function() {
  setTimeout(function () {
    throw new Error('Failed!')
  }, Math.round(Math.random()*100))
})
```

# Source code for Domain

https://github.com/nodejs/node/blob/master/lib/domain.js

```
var _domain = [null];
Object.defineProperty(process, 'domain', {
  enumerable: true,
  get: function() {
    return _domain[0];
  },
  set: function(arg) {
    return _domain[0] = arg;
  }
});
```

# Domain overwrites uncaught exception

```
if (process.hasUncaughtExceptionCaptureCallback()) {
  throw new errors.Error('ERR_DOMAIN_CALLBACK_NOT_AVAILABLE');
}

process.on('removeListener', (name, listener) => {
  if (name === 'uncaughtException' &&
      listener !== domainUncaughtExceptionClear) {
    // If the domain listener would be the only remaining one, remove it.
    const listeners = process.listeners('uncaughtException');
    if (listeners.length === 1 && listeners[0] === domainUncaughtExceptionClear)
      process.removeListener(name, domainUncaughtExceptionClear);
  }
});
```

# C++ Addons

# How to Write C/C++ binding for your IoT, hardware, drone, smartdevice, etc.?

# Node and C++

## Create the `hello.cc` file:

```
#include <node.h>

namespace demo {

using v8::FunctionCallbackInfo;
using v8::HandleScope;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;
```

# Node and C++

## Create the `hello.cc` file:

```
void Method(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  args.GetReturnValue().Set(String::NewFromUtf8(isolate, "capital one"));
}

void init(Local<Object> exports) {
  NODE_SET_METHOD(exports, "hello", Method);
}

NODE_MODULE(addon, init)

}  // namespace demo
```

# Creating `binding.gyp`

Create `binding.gyp`:

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "hello.cc" ]
    }
  ]
}
```

# node-gyp

```
$ npm install -g node-gyp
```

https://github.com/nodejs/node-gyp

# Configuring and Building

```
$ node-gyp configure
$ node-gyp build
```

Check for compiled `.node` files in build/Release/

# C++ Addons Examples

https://github.com/nodejs/node-addon-examples

# Including Addon

Create `hello.js` and include your C++ addon:

```
var addon = require('./build/Release/addon')
console.log(addon.hello()) // 'capital one'
```

Run

```
$ node hello.js
```

# Moaaarr *core* Node

- NAPI
- Crypto
- HTTP/2
- Node modules
- npm commands and scripts

# 30-Second Summary

1. Event Emitters
2. Streams
3. Buffers
4. Clusters
5. C++ Addons
6. Domain

# There are big, big, big problems

- 2 days
- 5 days
- 8 days

# New information

- Repetition
- Immersion
- Practice

# Solution: You Don't Know Node Online Course


https://node.university/p/you-dont-know-node

# Sign up for Node University NOW!


Node.University is the ultimate resource for the best online Node education!

https://node.university 🚀

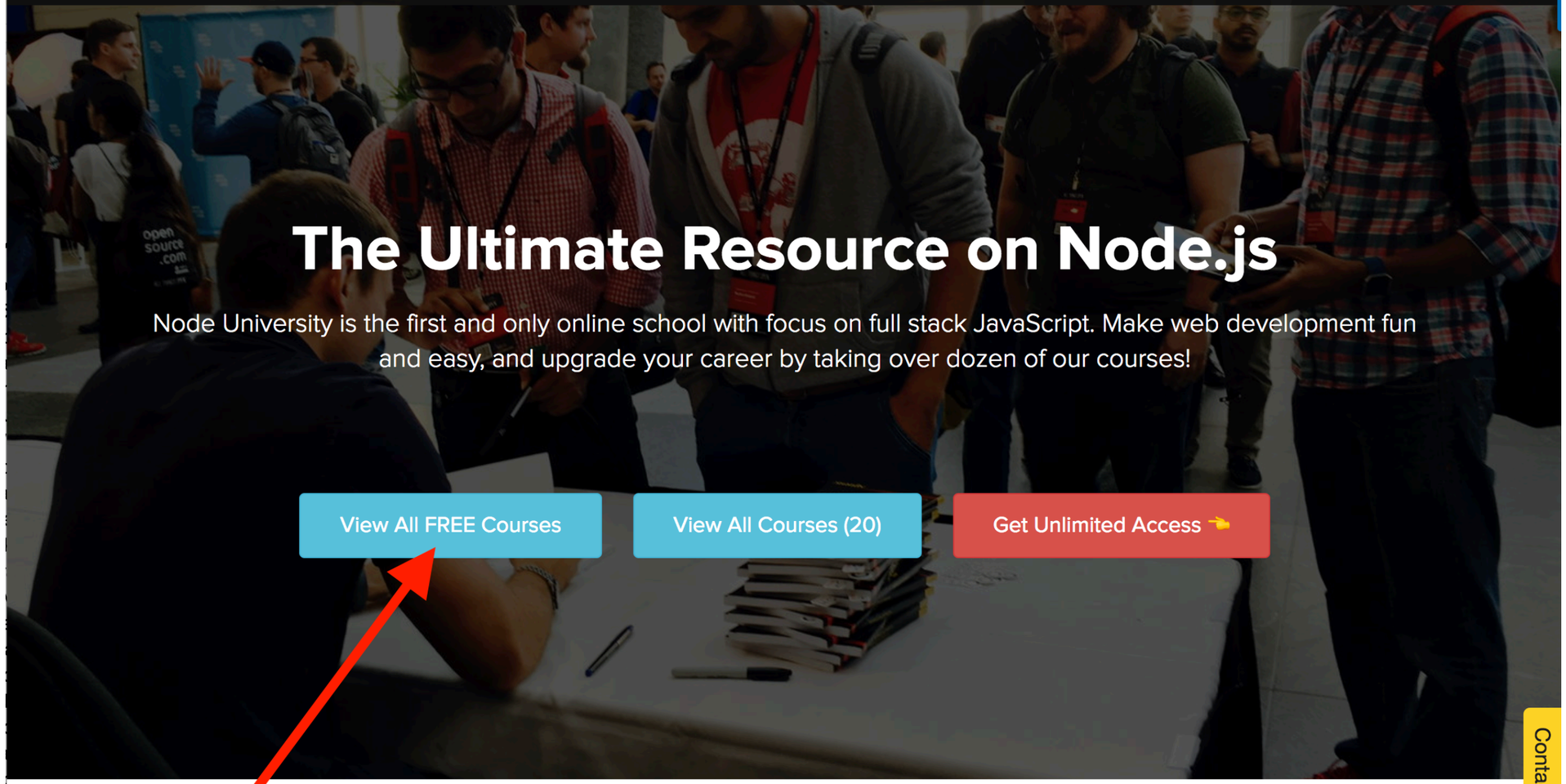
# This is how to do it

# Take out your laptops or phones  

Seriously. Do it now!

# Go to node.university

(https://node.university in Safari, Chrome, Edge)



# The Ultimate Resource on Node.js

Node University is the first and only online school with focus on full stack JavaScript. Make web development fun and easy, and upgrade your career by taking over dozen of our courses!

View All FREE Courses | View All Courses (20) | Get Unlimited Access 👈

Making thousands of software engineers more productive

CISCO @WalmartLabs CapitalOne STARBUCKS vmware HACK REACTOR GENERAL ASSEMBLY HubSpot ondeck

Contact Node University

Contact Node University

Category: Free Author: All

Find a course

# Free

Get the taste of how amazing it is to work with Node and JavaScript


## Node and MongoDB Foundation

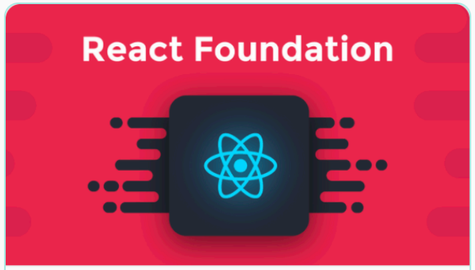
Intro to Node and MongoDB for Absolute Beginners

Azat Mardan FREE


## You Don't Know Node

Quick Intro to 5 Core Node.js Features

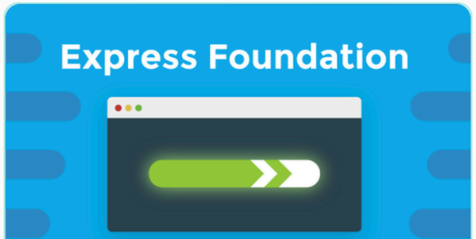
Azat Mardan FREE




## React Foundation

Start Developing with React.js, Babel and Webpack

Azat Mardan FREE

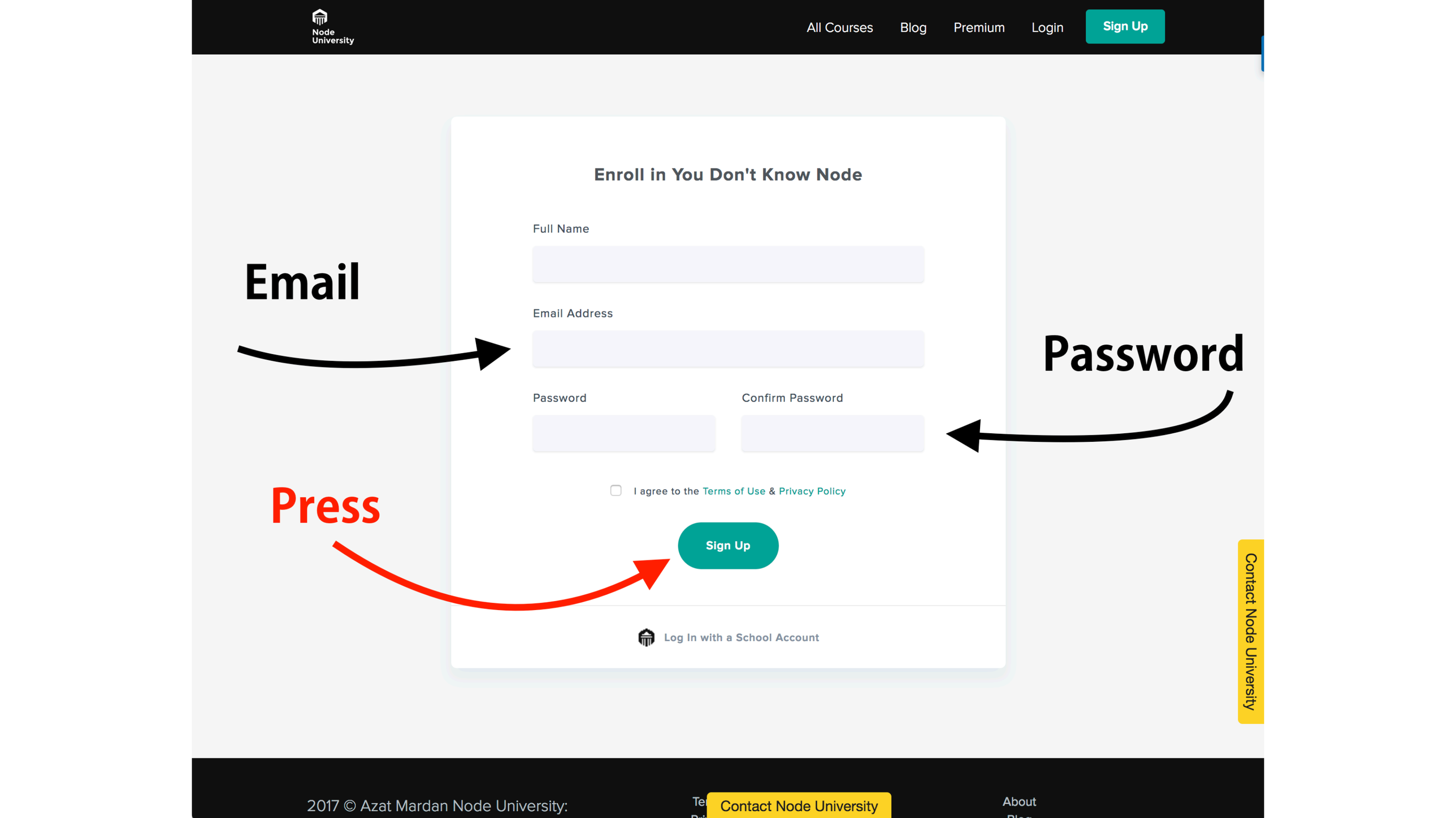

Node University

All Courses Blog Premium Login Sign Up

# You Don't Know Node

## Quick Intro to 5 Core Node.js Features

Enroll in Course for FREE


Node University
All Courses
Blog
Premium
Login
Sign Up
Enroll in You Don't Know Node
Full Name
Email
Email Address
Password
Confirm Password
Password
I agree to the Terms of Use & Privacy Policy
Press
Sign Up
Log In with a School Account
Contact Node University
2017 © Azat Mardan Node University:
Contact Node University
About

# One Last Thing 👉

# CodingHorror.com

**Atwood's Law**

Any application that *can* be written in JavaScript, *will* eventually be written in JavaScript.

# The End