You Don't Know Node

Quick Intro to 6 Core Features

Better Apps—Better Life

About Presenter

Azat Mardan

Twitter: @azat_co

Email: hi@azat.co

Blog: webapplog.com

About Presenter

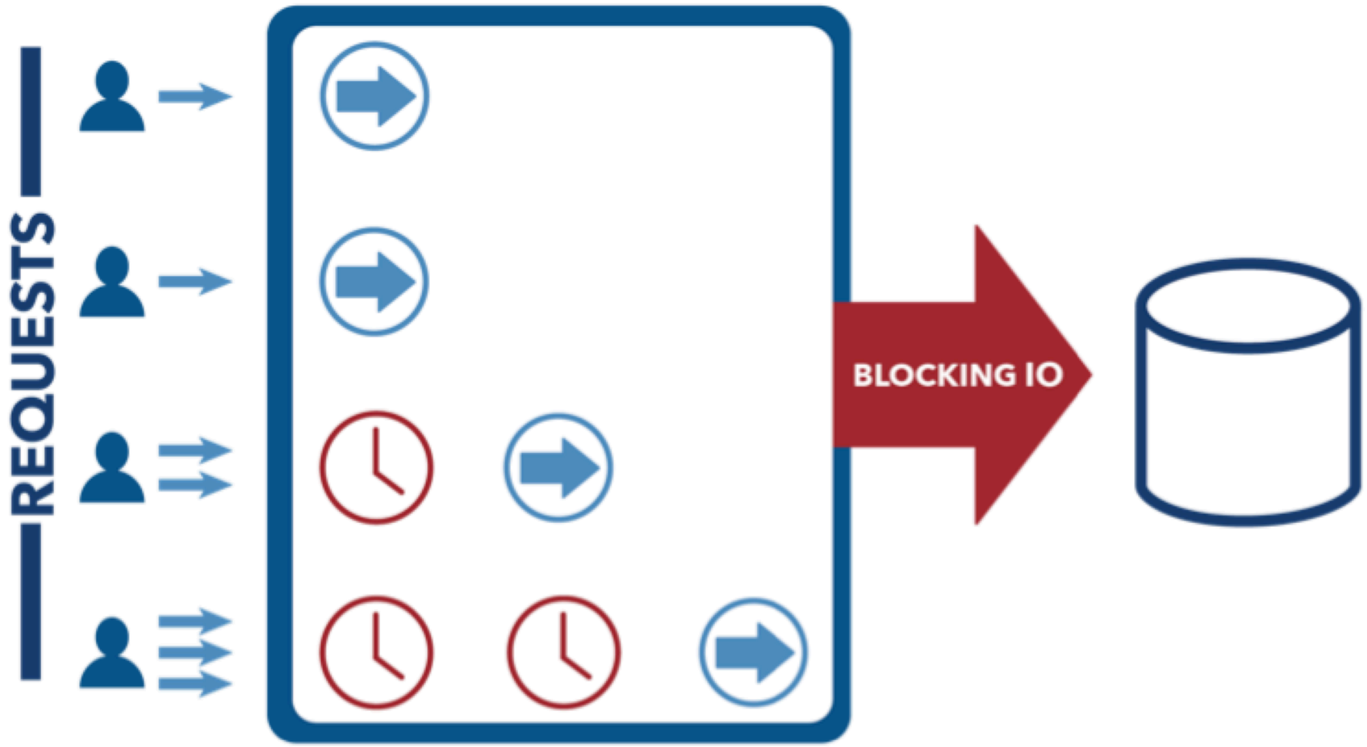
- Work: Technology Fellow at Capital One
- Experience: FDIC, NIH, DocuSign, HackReactor and Storify
- Books: React Quickly, Practical Node.js, Pro Express.js, Express.js API and 8 others
- Teach: NodeProgram.com

Most apps wait for input/output which are the most expensive tasks

Java Sleep

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

TECH | BLOCKING I/O



Node "Sleep"

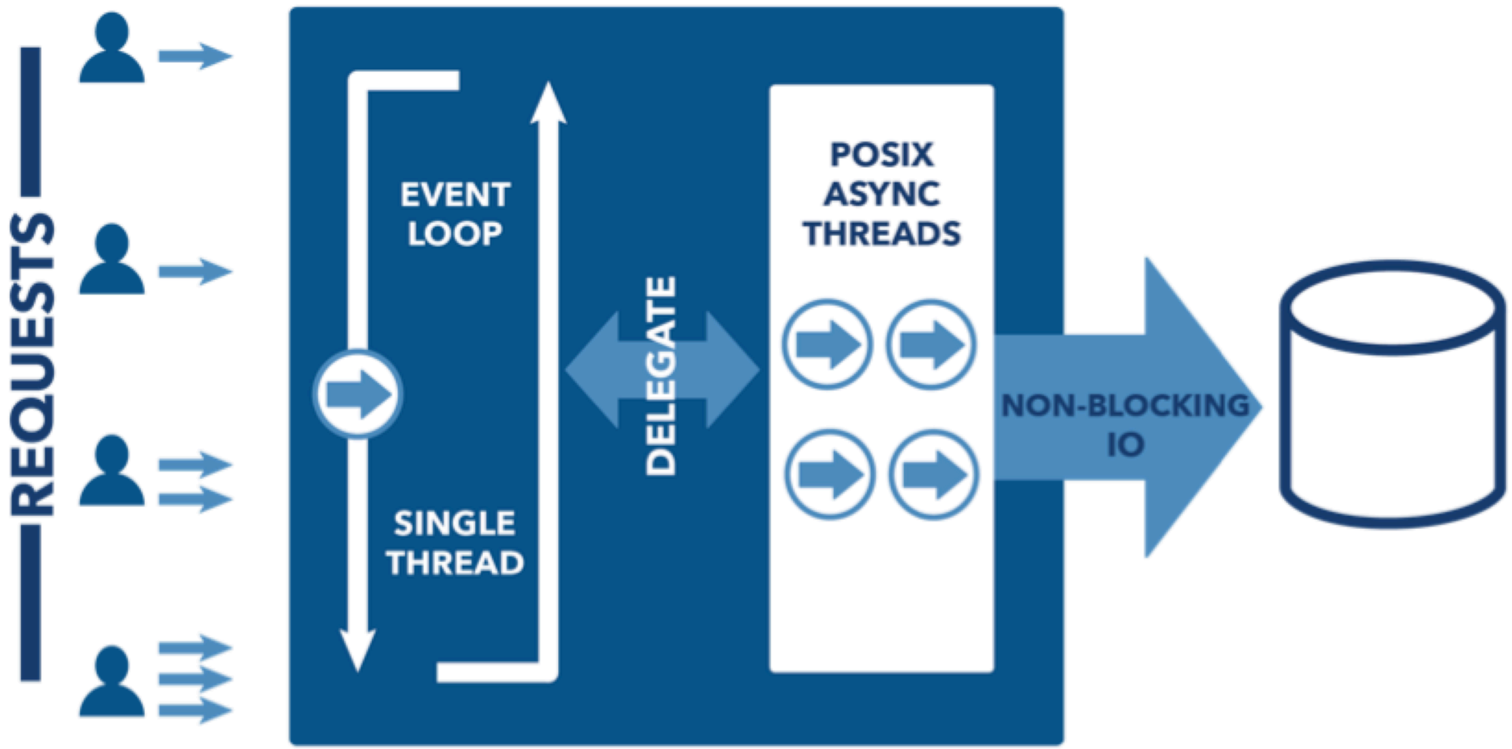
```
console.log('Step: 1')
setTimeout(function () {
  console.log('Step: 3')
}, 1000)
console.log('Step: 2')
```

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')
```

Event Loop

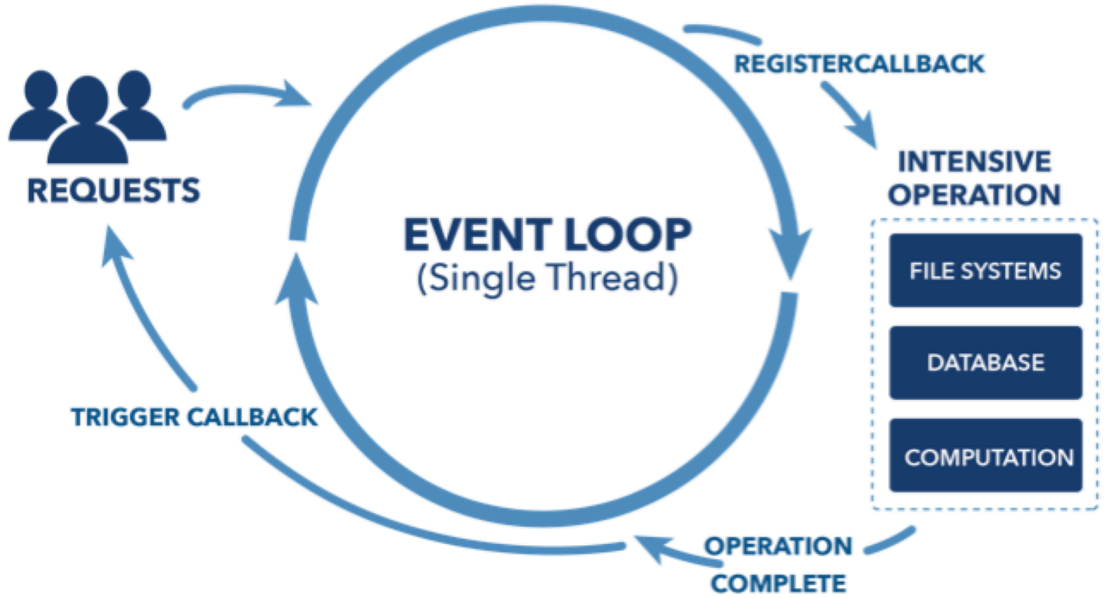
TECH | NON-BLOCKING I/O



Single thread - worry free

Mutli-threading is like a nuclear device

TECH | NON-BLOCKING I/O



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<10000000000; 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!')
//data1->Hello Ruby->data2->Hello NODE!
```

Non-Blocking Node.js Code

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

var contents = fs.readFile('accounts.txt','utf8', function(err,contents){
   console.log(contents)
})
console.log('Hello Python\n')

var contents = fs.readFile('ips.txt','utf8', function(err,contents){
   console.log(contents)
})
console.log("Hello Node!")
//Hello Python->Hello Node->data1->data2
```

Node != Browser JavaScript

- Where to store passwords?
- How to create global variables (no `window` in Node)?
- How to access CLI input, OS, platform, memory usage, versions, etc.?

Global

`global.process`

`global.__filename`

global.__dirname

global.module

`global.require()`

```
global.console
global.setInterval()
global.setTimeout()
```

Process

process.pid

process.versions

`process.arch`

process.argv

process.env

```
process.uptime()
```

```
process.memoryUsage()
```

`process.cwd()`

process.exit()

process.on()

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 😞

Event Emitters

Event Emitters

Event emitter is something that triggers an event to which anyone can listen.

Event Emitters

- Event handling in Node uses the observer pattern
- An event, or subject, keeps track of all functions that are associated with it
- These associated functions, known as observers, are executed when the given event is triggered

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)
```

Problems with Large Data

- Speed: Too slow
- Buffer limit: ~1Gb

Streams

Abstactions for continuos 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

`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 ---')
})
```

Notes:

- `data` - input fed into the program. Depending on the size of the input, this event can trigger multiple times
- an `end` event is necessary to signal the conclusion of the input
  stream
- `stdin` is paused by default, and must be resumed before data can
  be read from it

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

`process.stdout`

The standard output streams contain data going out of an application.

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) => {
  var body = ''
  req.setEncoding('utf8')
  req.on('data', (chunk) => {
    body += chunk
  })
  req.on('end', () => {
    var data = JSON.parse(body)
    res.write(typeof data)
    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:

- `new Buffer(size)`
- `new Buffer(array)`
- `new Buffer(buffer)`
- `new Buffer(str[, encoding])`

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

Working with Buffer

```
// buf.js
var buf = new Buffer(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!

Buffer Methods and Properties

- `buf.length`
- `buf.write(string[, offset][, length][, encoding])`
- `buf.toString([encoding][, start][, end])`
- `buf.toJSON()`

Buffer Methods and Properties

- `buf.equals(otherBuffer)`
- `buf.compare(otherBuffer)`
- `buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])`
- `buf.slice([start][, end])`
- `buf.fill(value[, offset][, end])`

Encodings

```
ascii utf8 utf16le
ucs2 base64 binary
char hex
```

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

DevTools

Stream responds faster:

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

Stream Resources

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?

Clusters

```
var cluster = require('cluster')
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)
})
```

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)
}
```

Async Errors

The app crashes! 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

This module is softly deprecated in 4.0 (most likey will be separate from core module), but there's no alternatives in core as of now.

Domain Example

```
var 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:

```
var 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))
});
```

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
```

Capital One

We use Node a lot!

https://www.youtube.com/watch?v=BJPeLJhv1Ic

30-Second Summary

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

Slides & Code 

https://github.com/azat-co/you-dont-know-node

or

PDF: http://bit.ly/1VJWpQK

Q&A ❓ 

Twitter: @azat_co

Email: hi@azat.co