

YOU DON'T KNOW NODE.JS

QUICK GUIDE TO THE BEST FEATURES

SLIDES



[HTTPS://GITHUB.COM/AZAT-CO/YOU-DONT-KNOW-NODE](https://github.com/AZAT-CO/YOU-DONT-KNOW-NODE)

OR

PDF: [HTTP://BIT.LY/1VJWPQK](http://bit.ly/1VJWPQK)

OR

```
$ mkdir node_modules && npm install you-  
dont-know-node
```

KEY TAKEAWAYS

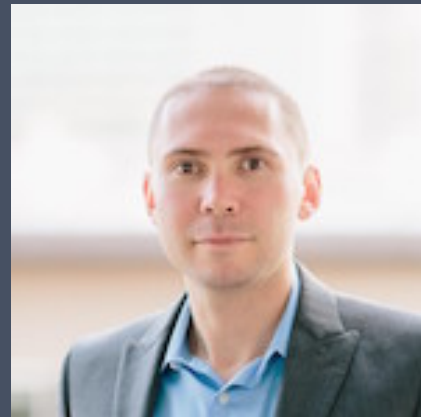
1. EVENT LOOP: BRUSH-UP ON THE CORE CONCEPT WHICH ENABLES THE NON-BLOCKING I/O
2. STREAMS AND BUFFERS: EFFECTIVE WAY TO WORK WITH DATA
3. GLOBAL AND PROCESS: HOW TO ACCESS MORE INFO

MORE KEY TAKEAWAYS

1. EVENT EMITTERS: CRASH COURSE IN THE EVENT-BASED PATTERN
2. CLUSTERS: FORK PROCESSES LIKE A PRO
3. HANDLING ASYNC ERRORS: ASYNCWRAP, DOMAIN AND UNCAUGHTEXCEPTION
4. C++ ADDONS: CONTRIBUTING TO THE CORE AND WRITING YOUR OWN C++ ADDONS

ABOUT PRESENTER

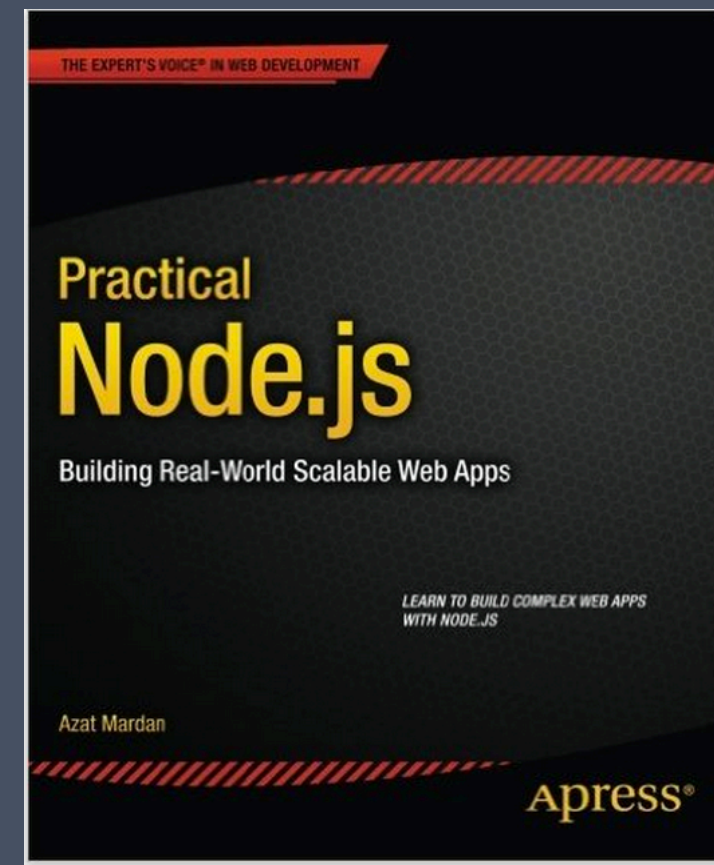
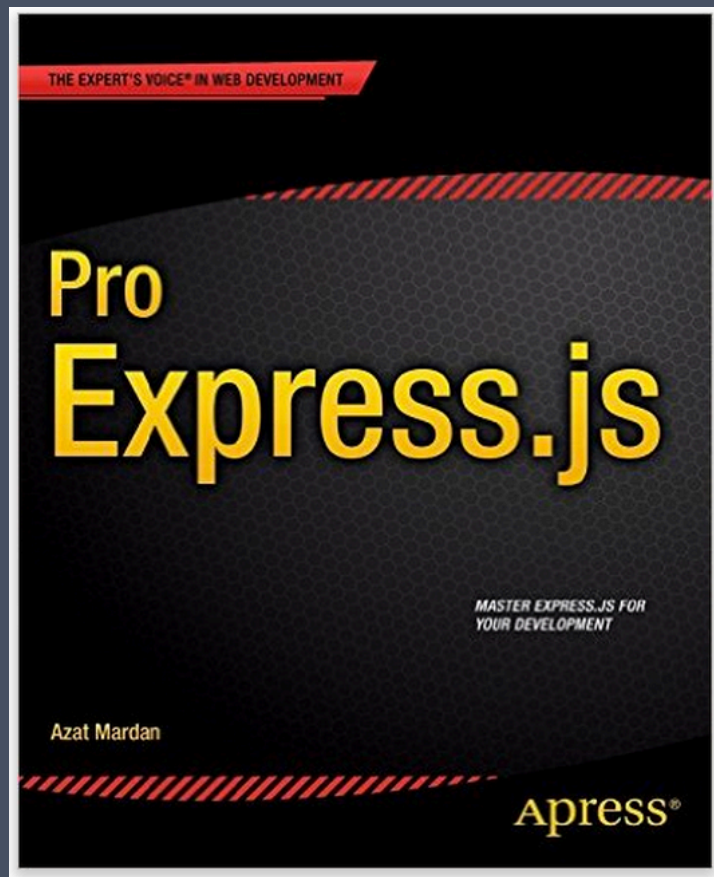
AZAT MARDAN



TWITTER: @AZAT_CO
EMAIL: HI@AZAT.CO
BLOG: WEBAPPLOG.COM

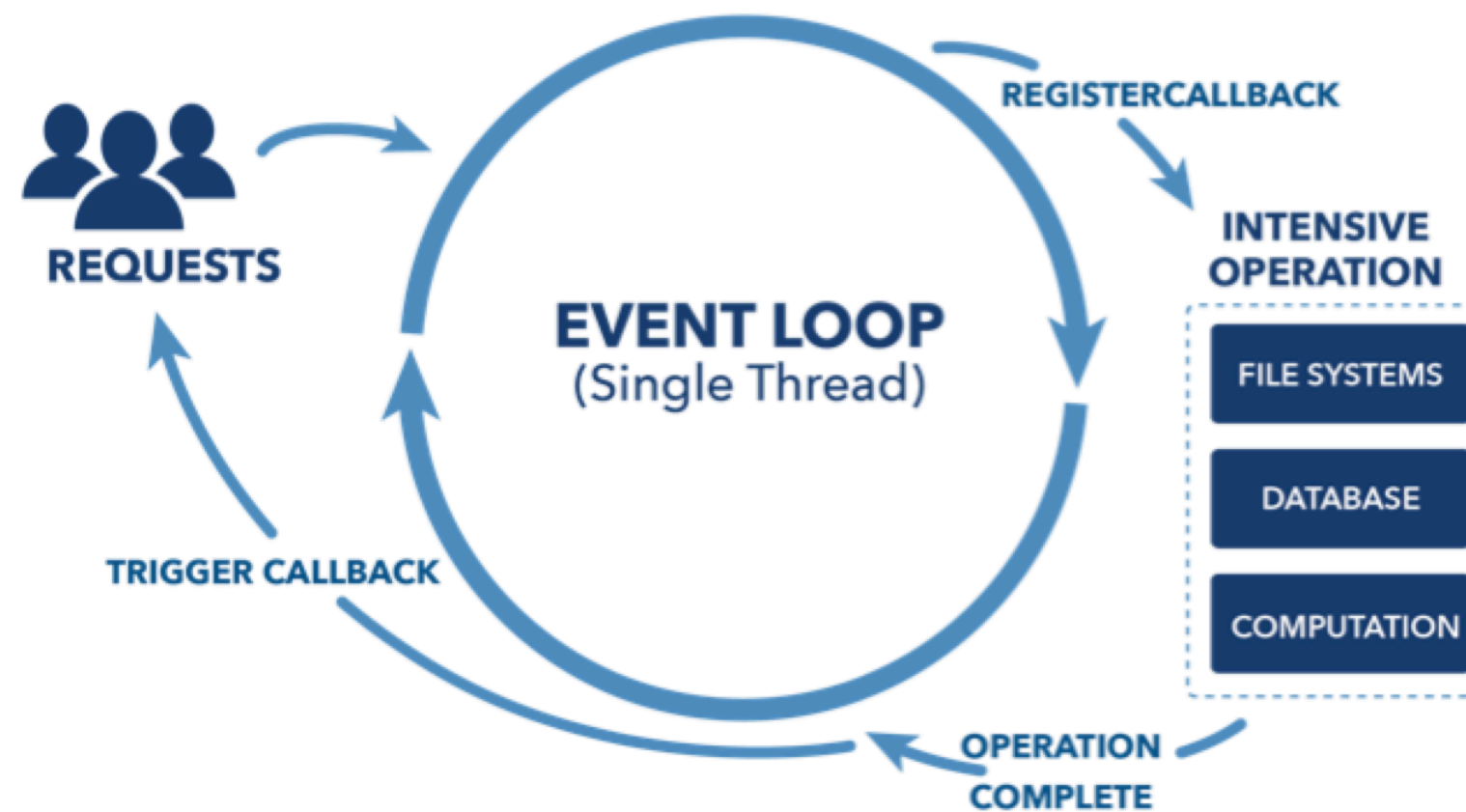
ABOUT PRESENTER

- TECHNOLOGY FELLOW AT CAPITAL ONE
- EXPERIENCE: FDIC, NIH, DOCUSIGN, HACKREACTOR AND STORIFY
- BOOKS: PRACTICAL NODE.JS, PRO EXPRESS.JS AND EXPRESS.JS API



EVENT LOOP

NON-BLOCKING I/O



BASIC EVENT LOOP EXAMPLE

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

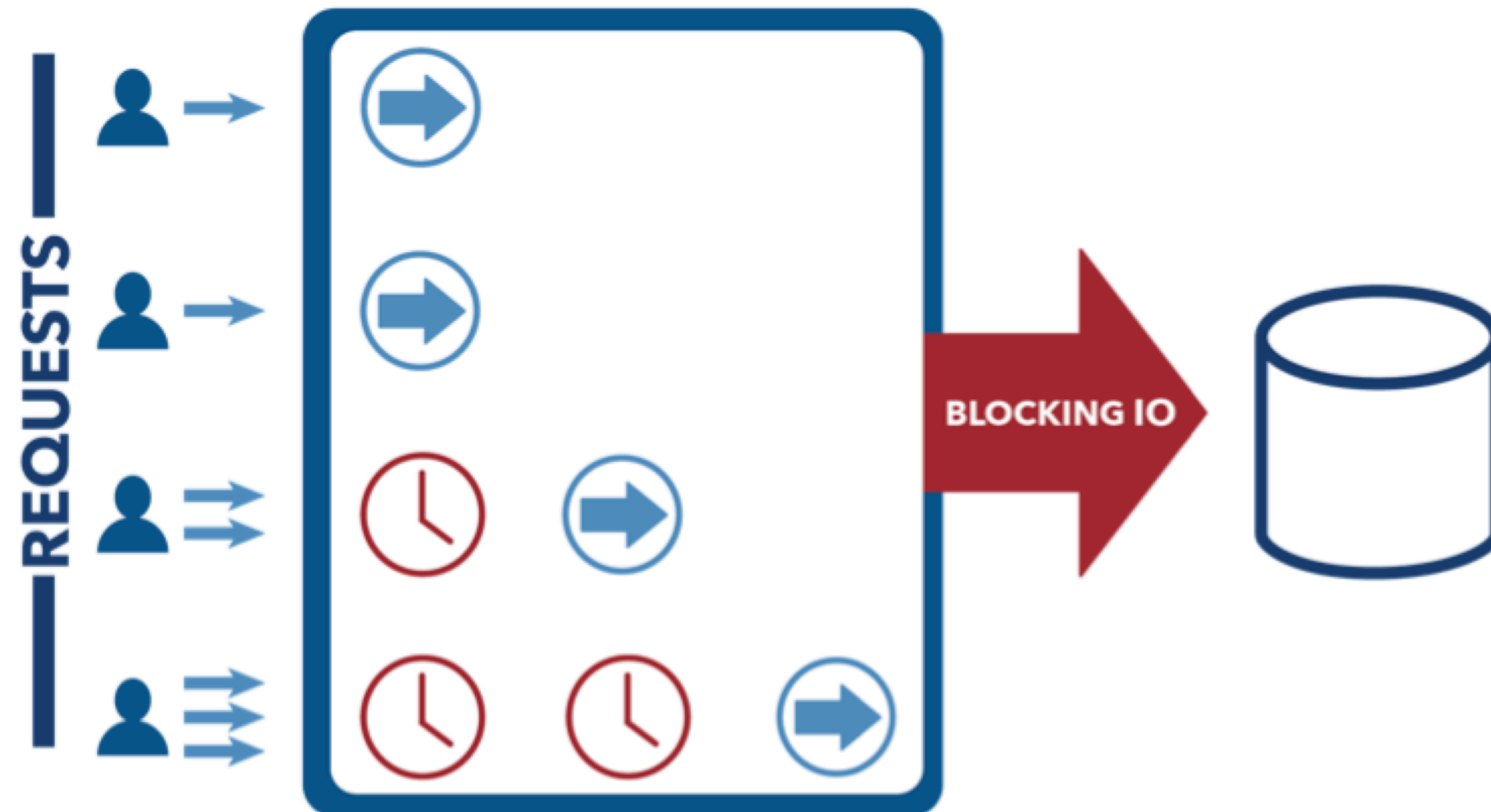
VS.

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

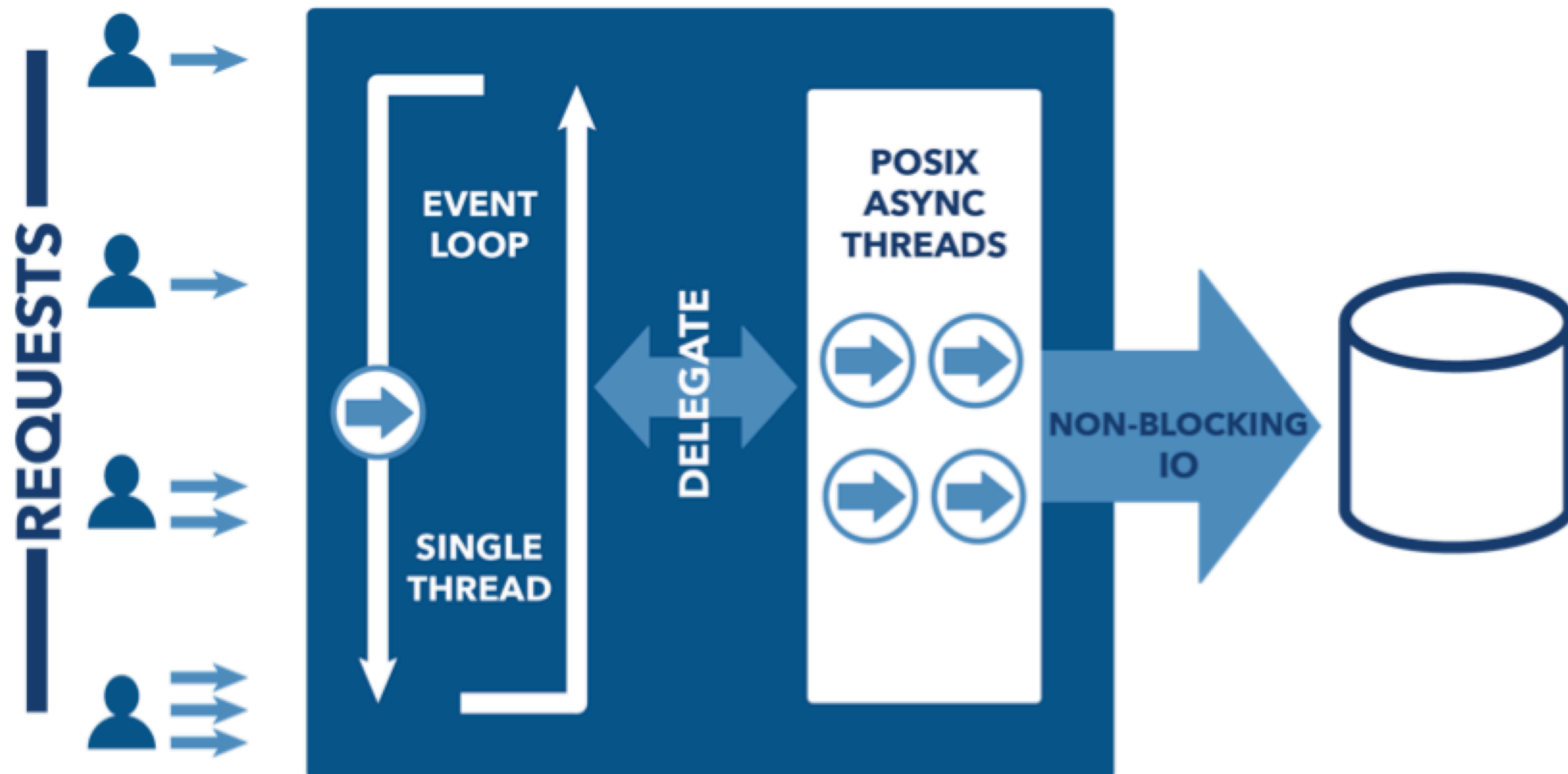
THINKING IN ASYNC CODE

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

TECH | **BLOCKING I/O**



TECH | **NON-BLOCKING I/O**



IT'S STILL POSSIBLE TO
WRITE BLOCKING CODE IN
NODE.JS. 🤪

BLOCKING NODE.JS CODE

```
var fs = require('fs');  
  
var contents = fs.readFileSync('accounts.txt', 'utf8');  
console.log(contents);  
console.log('Hello Capital One\n');  
  
var contents = fs.readFileSync('ips.txt', 'utf8');  
console.log(contents);  
console.log('Hello SECON!');
```

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 Capital One\n');

var contents = fs.readFile('ips.txt', 'utf8', function(err, contents){
    console.log(contents);
});
console.log("Hello SECON!");
```


STREAMS AND BUFFERS

STANDARD STREAMS

STANDARD STREAMS ARE I/O CHANNELS BETWEEN AN APPLICATION AND ITS EXECUTION ENVIRONMENT.

THERE ARE THREE STANDARD STREAMS:

- > STANDARD INPUT - `stdin`
- > STANDARD OUTPUT - `stdout`

stdin

STANDARD INPUT STREAMS CONTAIN DATA GOING INTO APPLICATIONS.

THIS IS ACHIEVED VIA A READ OPERATION.

INPUT TYPICALLY COMES FROM THE KEYBOARD USED TO STARTED THE PROCESS.

TO LISTEN IN ON DATA FROM STDIN, USE THE `data` AND `end` EVENTS:

```
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

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.**

TO WRITE TO `stdout`, USE THE `write` FUNCTION:

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

`stderr`

THE STANDARD ERROR STREAM IS AN OUTPUT STREAM LIKE
`stdout`.

IT IS USED PRIMARILY TO LOG MESSAGES AND ERRORS FOR THE
PURPOSE OF
DEBUGGING.

WRITING TO `stderr` IS DONE SIMILARLY TO `stdout`:

```
process.stderr.write('An error message\n');
```

**NOTE THAT `stdout` AND `stderr` ARE SPECIAL STREAMS IN
NODE AS THEY ARE BLOCKING!**

TELETYPE CONTEXT

TO CHECK IF THE APPLICATION IS BEING RUN IN TTY CONTEXT.

USE THE `isTTY`
PROPERTY:

```
$ node teletype.js  
// process.stdin.isTTY === true  
// process.stdout.isTTY === true
```

```
$ echo "hello world" | node teletype.js  
// process.stdin.isTTY === false
```

```
$ node teletype.js | cat  
// process.stdout.isTTY === false
```

BUFFERS

BINARY DATA TYPE. TO CREATE:

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

DOCS: [HTTP://BIT.LY/1EACZ1](http://bit.ly/1EACZ1)

```
buf = new Buffer(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97; // 97 is ASCII a
}
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>
```

BUFFER CONVERSION:

```
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) throw 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` - FOR 7 BIT ASCII DATA ONLY. VERY FAST. STRIPS THE HIGH BIT IF SET.
- > `utf8` - MULTIBYTE ENCODED UNICODE CHARACTERS. STANDARD FOR THE WEB.
- > `utf16le` - 2 OR 4 BYTES. LITTLE ENDIAN ENCODED UNICODE CHARS.

ENCODINGS

- `ucs2` - ALIAS OF 'UTF16LE'.
- `base64` - BASE64 STRING ENCODING.
- `binary` - DEPRECATED RAW BINARY DATA INTO STRINGS BY USING ONLY THE FIRST 8 BITS OF EACH CHAR.
- `hex` - ENCODE EACH BYTE AS TWO HEXADECIMAL CHARACTERS.

STREAMS AND BUFFER DEMO

SERVER-STREAM.JS:

```
app.get('/stream1', function(req, res) {  
  var stream = fs.createReadStream(largeImagePath)  
  stream.pipe(res)  
})
```

```
$ node server-stream
```

HTTP://LOCALHOST:3000/STREAM1
HTTP://LOCALHOST:3000/NON-STREAM

STREAM RESOURCES

[HTTPS://GITHUB.COM/SUBSTACK/STREAM-ADVENTURE](https://github.com/substack/stream-adventure)

```
$ sudo npm install -g stream-adventure  
$ stream-adventure
```

[HTTPS://GITHUB.COM/SUBSTACK/STREAM-HANDBOOK](https://github.com/substack/stream-handbook)

GLOBAL AND PROCESS

GLOBAL

- `global.process`
- `global.__filename`
- `global.__dirname`
 - `global.module`
- `global.require`

PROCESS

- › `process.pid`
- › `process.versions`
- › `process.arch`
- › `process.argv`
- › `process.env`

MORE PROCESS

- `process.uptime()`
- `process.memoryUsage()`
 - `process.cwd()`
 - `process.exit`
 - `process.on()`

EVENT EMITTERS

EVENT EMITTERS

EVENT EMITTER IS SOMETHING THAT TRIGGERS AN EVENT TO WHICH ANYONE CAN LISTEN.

[HTTPS://NODEJS.ORG/API/EVENTS.HTML](https://nodejs.org/api/events.html)

IN NODE.JS AN EVENT CAN BE DESCRIBED SIMPLY AS A STRING WITH A CORRESPONDING CALLBACK.

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

```
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

```
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);
```

CLUSTERS

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
- > STRONG-CLUSTER-CONTROL ([HTTPS://GITHUB.COM/STRONGLOOP/STRONG-CLUSTER-CONTROL](https://github.com/strongloop/strong-cluster-control)), OR `$ slc run`
- > PM2 ([HTTPS://GITHUB.COM/UNITECH/PM2](https://github.com/unitech/pm2))

PM2

[HTTPS://GITHUB.COM/UNITECH/PM2](https://github.com/unitech/pm2)

[HTTP://PM2.KEYMETRICS.IO](http://pm2.keymetrics.io)

ADVANTAGES:

- > LOAD-BALANCER AND OTHER FEATURES
- > OS RELOAD DOWN-TIME, I.E., FOREVER ALIVE

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);  
});
```

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.

ASYNCHRONOUS 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 `ASYNCTHROW`
 - 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 A 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

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](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](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'
```

Q&A 👍

SEND QUESTIONS TO

[HTTPS://GITHUB.COM/AZAT-CO/YOU-DONT-KNOW-NODE/
ISSUES](https://github.com/AZAT-CO/you-dont-know-node/issues)

TWITTER: @AZAT_CO

EMAIL: HI@AZAT.CO