NODE PATTERNS

FROM CALLBACKS TO OBSERVER

SLIDES 

Dear Kids,

Here's to the crazy ones. The misfits. The rebels. The troublemakers. The round pegs in the square holes. The ones who see things differently. They're not fond of rules. And they have no respect for the status quo. You can quote them, disagree with them, glorify or vilify them. About the only thing you can't do is ignore them. Because they change things. They push the human race forward. And while some may see them as the crazy ones, we see genius. Because the people who are crazy enough to think they can change the world, are the ones who do.

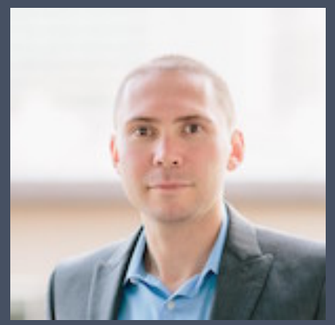
Take care.
John Appleseed

HTTPS://GITHUB.COM/AZAT-CO/NODE-PATTERNS

```
git clone https://github.com/azat-co/node-patterns
```

ABOUT PRESENTER

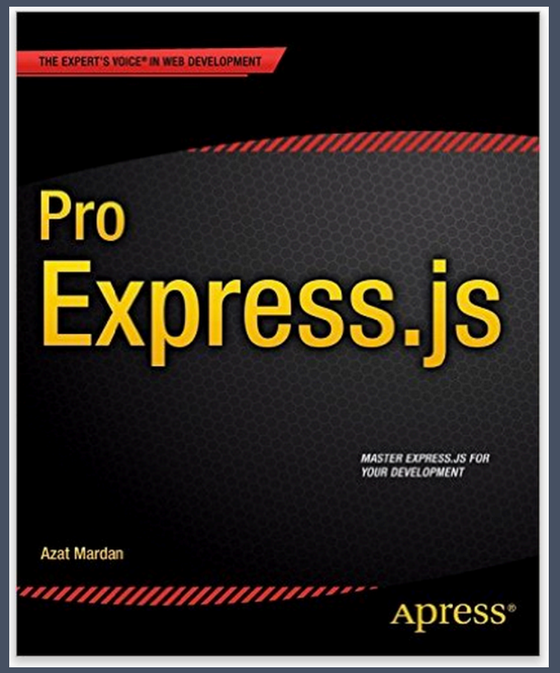
AZAT MARDAN

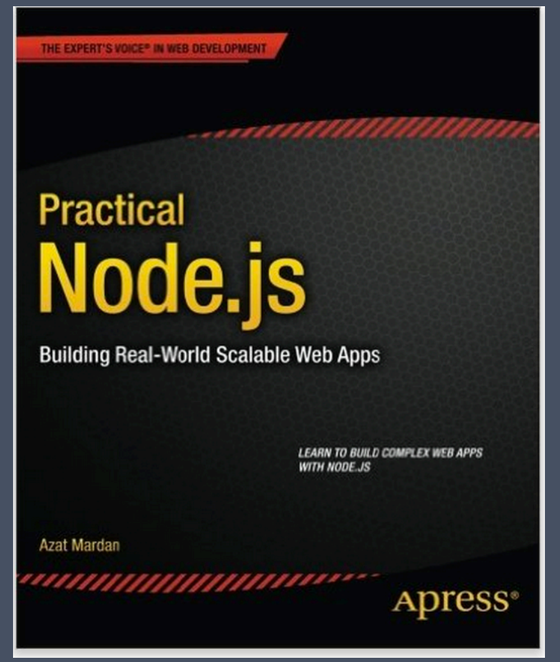


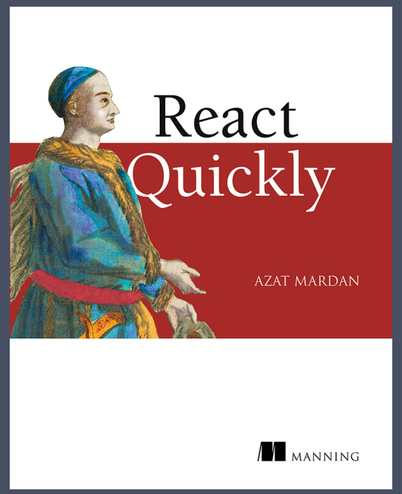
TWITTER: @AZAT_CO
EMAIL: HI@AZAT.CO
BLOG: WEBAPPLOG.COM

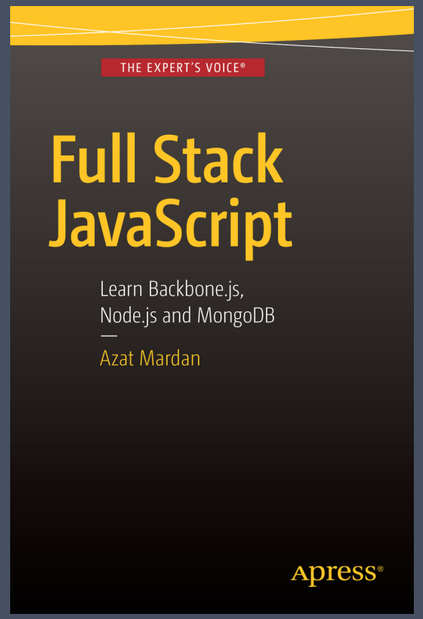
ABOUT PRESENTER

- WORK: TECHNOLOGY FELLOW AT CAPITAL ONE
- EXPERIENCE: FDIC. NIH. DOCUSIGN. HACKREACTOR AND STORIFY
- BOOKS: PRACTICAL NODE.JS. PRO EXPRESS.JS. EXPRESS.JS API AND 8 OTHERS
- TEACH: NODEPROGRAM.COM

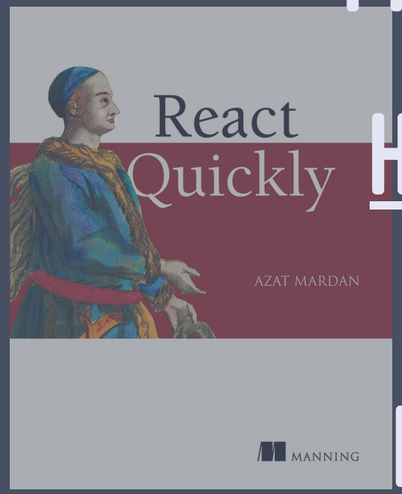

THE EXPERT'S VOICE® IN WEB DEVELOPMENT
Pro
Express.js
MASTER EXPRESS.JS FOR YOUR DEVELOPMENT
Azat Mardan
Apress®


THE EXPERT'S VOICE® IN WEB DEVELOPMENT
Practical
Node.js
Building Real-World Scalable Web Apps
LEARN TO BUILD COMPLEX WEB APPS WITH NODE.JS
Azat Mardan
Apress®


React
Quickly
AZAT MARDAN
MANNING


THE EXPERT'S VOICE®
Full Stack
JavaScript
Learn Backbone.js,
Node.js and MongoDB
—
Azat Mardan
Apress®

FREE: 7+ HOURS OF VIDEOS



HTTP://REACTQUICKLY.CO

AND

HTTP://BIT.LY/1UMNOPC

NODE BASICS

- JAVASCRIPT
- ASYNCHRONOUS + EVENT DRIVEN
- NON-BLOCKING I/O

WHY CARE?

- ASYNC CODE IS HARD
- CODE COMPLEXITY GROWS EXPONENTIALLY
- GOOD CODE ORGANIZATION IS IMPORTANT

JAVASCRIPT?

PROBLEM 1

HOW TO SCHEDULE SOMETHING IN THE FUTURE?

CALLBACKS ALL THE WAY!

FUNCTIONS ARE FIRST-CLASS CITIZENS

```
var t = function(){...}
setTimeout(t, 1000)
```

T IS A CALLBACK

CALLBACK CONVENTION

```
var fs = require('fs')
var callback = function(error, data){...}
fs.readFile('data.csv', 'utf-8', callback)
```

CONVENTIONS

- `error` 1ST ARGUMENT. NULL IF EVERYTHING IS OKAY
  - `data` IS THE SECOND ARGUMENT
  - `callback` IS THE LAST ARGUMENT

NOTE

NAMING DOESN'T MATTER BUT ORDER MATTERS.

NODE.JS WON'T ENFORCE THE ARGUMENTS.

CONVENTION IS NOT A GUARANTEE. IT'S JUST A STYLE. - READ DOCUMENTATION OR SOURCE CODE.

PROBLEM 2

HOW TO ENSURE THE RIGHT SEQUENCE? CONTROL FLOW 😕

EXAMPLE

HTTP REQUESTS TO:

1. GET AN AUTH TOKEN
2. FETCH DATA
3. PUT AN UPDATE

THEY MUST BE EXECUTED IN A CERTAIN ORDER.

```
... // callback is defined, callOne, callTwo, and callThree are defined
callOne({...}, function(error, data1) {
    if (error) return callback(error, null)
    // work to parse data1 to get auth token
    // fetch the data from the API
    callTwo(data1, function(error, data2) {
        if (error) return callback(error, null)
        // data2 is the response, transform it and make PUT call
        callThree(data2, function(error, data3) {
            //
            if (error) return callback(error, null)
            // parse the response
            callback(null, data3)
        })
    })
})
```

WELCOME TO CALLBACK HELL

CALLBACK HELL

- HARD TO READ
- HARD TO MODIFY/MAINTAIN/ENHANCE
- EASY FOR DEVS TO MAKE BUGS
- CLOSING PARENS - 

CALLBACKHELL.COM

SOLUTIONS

- ABSTRACT INTO NAMED FUNCTIONS (HOISTED OR VARIABLES)
- USE OBVERVERS
- USE ADVANCED LIBRARIES AND TECHNIQUES

NAMED FUNCTIONS

```
callOne({...}, processResponse1)

function processResponse1(error, data1) {
  callTwo(data1, processResponse2)
}

function processResponse2(error, data2) {
  callThere(data2, processResponse3)
}

function processResponse3(error, data1) {
  ...
}
```

MODULAR FUNCTIONS

```
var processResponse1 = require('./response1.js')
callOne({...}, processResponse1)

// response1.js
var processResponse2 = require('./response2.js')
module.exports = function processResponse1(error, data1) {
  callTwo(data1, processResponse2)
}
```

```
// response2.js
var processResponse3 = require('./response3.js')
module.exports = function processResponse2(error, data2) {
  callThere(data2, processResponse3)
}

// response3.js
module.exports = function processResponse3(error, data3) {
  ...
}
```

PROBLEM 3

HOW TO MODULARIZE CODE PROPERLY?

```
› module.exports = {...}
› module.exports.obj = {...}
› exports.obj = {...}
```

NOTE: `exports = {...}` IS ANTI-PATTERN.

PROBLEM 4

HOW TO MODULARIZE DYNAMIC CODE OR WHERE TO INITIALIZE?

SOLUTION

- `module.exports = function(options) {...}`
- `module.exports.func = function(options) {...}`
- `exports.func = function(options) {...}`

IMPORT

```
// code A
module.exports = function(options){
   // code B
}
```

WHEN YOU `require`. CODE A IS RUN AND CODE B IS NOT. CODE A IS RUN ONLY ONCE. NO MATTER HOW MANY TIMES YOU `require`.
YOU NEED TO INVOKE THE OBJECT TO RUN CODE B.

DEMO

```
node import-main
```

IMPORTING FOLDERS / PLUGIN PATTERN

```
// main.js
var routes = require('./routes')

// routes/index.js
module.exports = {
  users: require('./users.js'),
  accounts: require('./accounts.js')
  ...
}
```

SINGLETONS

- `require`: MODULES ARE CACHED

```
// module.js
var a = 1 // Private
module.exports = {
  b: 2 // Public
}
```

```
// program.js
var m = require('./module')
console.log(m.a) // undefined
console.log(m.b) // 2
m.b ++
require('./main')
```

```
// main.js
var m = require('./module')
console.log(m.b) // 3
```

DEMO

```
node main.js
node program.js
```

PROBLEM 5

MODULES ARE CACHED ON BASED ON THEIR RESOLVED FILENAME.

FILENAME WILL BREAK THE CACHING

```
var m = require('./MODULE')
var m = require('./module')
```

OR DIFFERENT PATHS

SOLUTION

```
global
```

`global.name`

OR

`GLOBAL.name`

```
_log = global.console.log
global.console.log = function(){
  var args = arguments
  args[0] = '\033[31m' +args[0] + '\x1b[0m'
  return _log.apply(null, args)
}
```

GLOBAL IS POWERFUL... ANTI-PATTERN

SIMILAR `window.jQuery = jQuery`

USE IT SPARRINGLY

PROBLEM 6: NO CLASSES

HOW TO ORGANIZE YOUR MODULAR CODE INTO CLASSES?

PROTOTYPES

(AT LEAST IN ES5)

OBJECTS INHERIT FROM OTHER OBJECTS

FUNCTIONS ARE OBJECTS TOO.

SOLUTION

```
module.exports = function(options) {
  // initialize
  return {
    getUsers: function() {...},
    findUserById: function(){...},
    limit: options.limit || 10,
    // ...
  }
}
```

SOLUTION

```
require('util').inherits(child, parent)
```

CALLBACKS EXTREME

NODE.JS MIDDLEWARE PATTERN

WHAT IS MIDDLEWARE

MIDDLEWARE PATTERN IS A SERIES OF PROCESSING UNITS CONNECTED TOGETHER. WHERE THE OUTPUT OF ONE UNIT IS THE INPUT FOR THE NEXT ONE. IN NODE.JS, THIS OFTEN MEANS A SERIES OF FUNCTIONS IN THE FORM:

```
function(args, next) {
  // ... Run some code
  next(output) // Error or real output
}
```

CONTINUITY

REQUEST IS COMING FROM A CLIENT AND RESPONSE IS SENT BACK TO THE CLIENT.

```
request->middleware1->middleware2->...middlewareN->route->response
```

EXPRESS.JS MIDDLEWARE

```
app.use(function(request, response, next) {
  // ...
  next()
}, function(request, response, next) {
  next()
}, function(request, response, next) {
  next()
})
```

PROBLEM 7

CALLBACKS ARE STILL HARD TO MANAGE EVEN IN MODULES!

EXAMPLE

1. MODULE JOB IS PERFORMING A TASK.
2. IN THE MAIN FILE. WE IMPORT JOB.

HOW DO WE SPECIFY A CALLBACK (SOME FUTURE LOGIC) ON THE JOB'S TASK COMPLETION?

MAYBE:

```
var job = require('./job.js')(callback)
```

WHAT ABOUT MULTIPLE CALLBACKS?

NOT VERY SCALABLE 😢

SOLUTION

OBSERVER PATTERN WITH EVENT EMITTERS!

```
// module.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
```

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

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

job.process()
```

```
emitter.listeners(eventName)

emitter.on(eventName, listener)

emitter.once(eventName, listener)

emitter.removeListener(eventName, listener)
```

DEPENDENCY INJECTION

```
// server.js
var app = express()
app.set(port, 3000)
...
app.use(logger('dev'))
...
var boot = require('./routes')(app)
boot({...}, function(){...})
```

FUNCTION WHICH RETURNS A FUNCTION

```
// routes/index.js
module.exports = function(app){
  return function(options, callback) {
    app.listen(app.get('port'), options, callback)
  }
}
```

THERE ARE MORE PATTERNS!

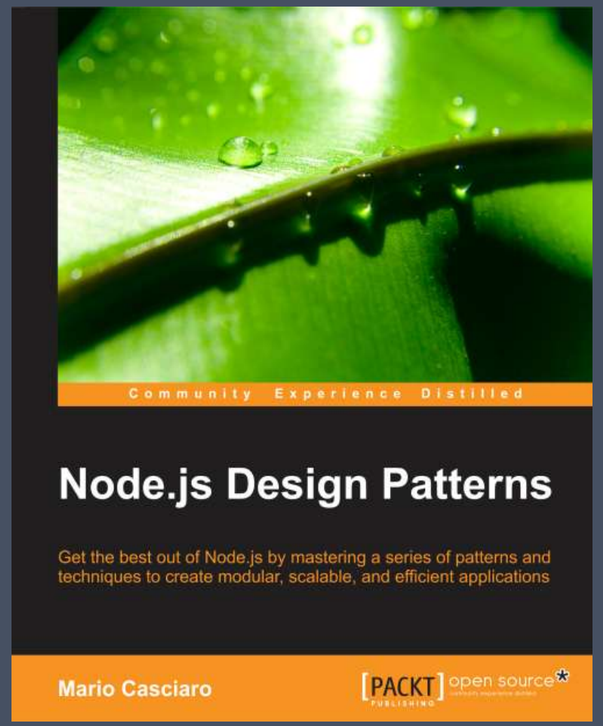
FURTHER ASYNC

- `async` AND `neo-async`
- PROMISES - NOT REALLY HELPING MUCH
- GENERATORS - PROMISING
- ASYNC AWAIT - NICE WRAPPER FOR PROMISES

FURTHER STUDY

- hooks
- require-dir, require-directory AND require-all

FURTHER READING



HTTP://AMZN.TO/21HXXTY

30-SECOND SUMMARY

1. CALLBACKS
2. OBSERVER
3. SINGLETON
4. PLUGINS
5. MIDDLEWARE
6. BUNCH OF OTHER STUFF 

THE END

I KNOW IT'S BEEN A LOT 😓 EVENT EMITTERS, MODULES AND CALLBACKS ARE AT THE CORE OF NODE. KNOW THY PATTERNS!

WE LOSE WHAT WE DON'T USE.

LEARNING NODE+REACT

NODEPROGRAM.COM

WHAT: NODE+EXPRESS+MONGODB+REACT
WHERE: FLATIRON SCHOOL. NYC
WHEN: MARCH 12-13. NYC

RATE THIS TALK 👍

SCALE 1-10 (10 IS HIGHEST)

ANYONE BELOW 8?

THIS IS YOUR CHANCE ASK A QUESTION TO MAKE IT 10!

Q&A ❓🙋👍

SEND BUGS 🐛 TO

HTTPS://GITHUB.COM/AZAT-CO/NODE-PATTERNS/ISSUES

TWITTER: @AZAT_CO
EMAIL: HI@AZAT.CO