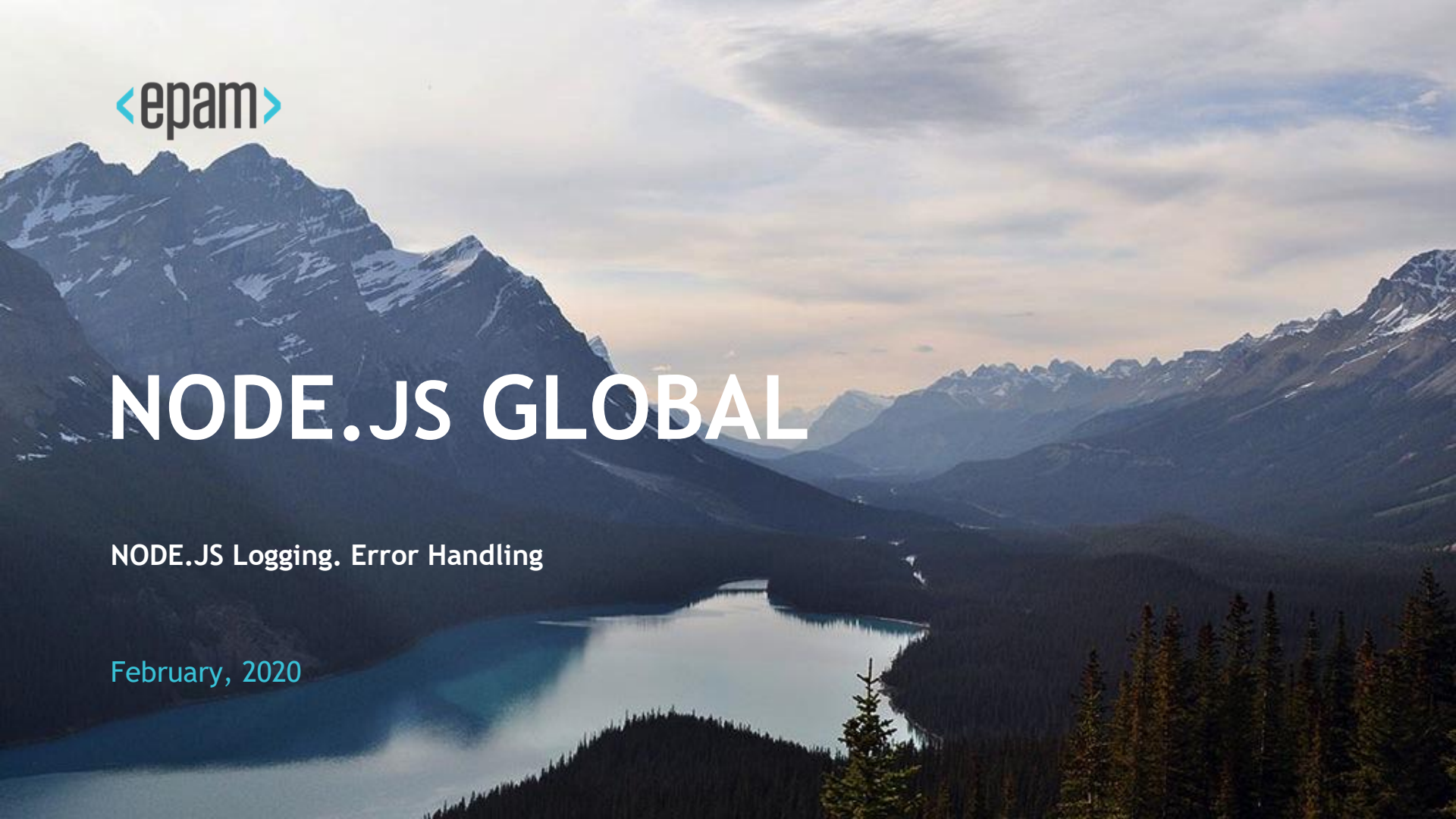


NODE.JS GLOBAL

NODE.JS Logging. Error Handling

February, 2020

DMITRY BELOICHUK

Software Engineer

- Full-stack developer
- Work with React.js, Node.js, AWS in production
- Play with *.js in non-production
- Give BSUIR and The Rolling Scopes lectures on CoreJS
- Give React.js, Vue.js and Node.js lectures and mentor collegues at Mentoring Programs

Contact:
https://github.com/Dmitry-White
https://www.linkedin.com/in/dmitry-white/

Agenda

- Logging
- Loggers
  - Debug
  - Bunyan
  - Winston
  - Morgan
- Error Handling
  - Types of Errors
  - Options
  - Uncaught Exceptions

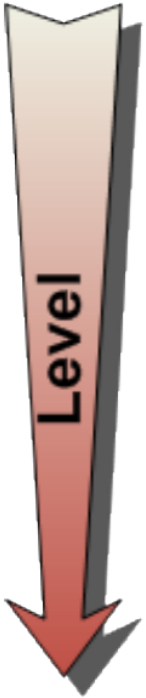
Logging

Logs - the events that reflect the various aspect of your application, it is the easiest mode of troubleshooting and diagnosing your application, if written correctly by the team.

Every log should contain the three most important parts:

- Source of log
- Timestamp
- Level and context

Logging



DEBUG — fine-grained informational events that are most useful to debug an application

INFO — informational messages that highlight the progress of the application at coarse-grained level

WARN — potentially harmful situations

ERROR — error events that might still allow the application to continue running

FATAL — very severe error events that will presumably lead the application to abort

Loggers: Debug

Debug - A tiny JavaScript debugging utility modelled after Node.js core's debugging technique. Works in Node.js and web browsers.

2. bash

```
$ DEBUG=* node examples/node/app.js
  http booting 'My App' +0ms
  worker:a doing lots of uninteresting work +0ms
  worker:b doing some work +0ms
  http listening +23ms
  worker:a doing lots of uninteresting work +424ms
  worker:a doing lots of uninteresting work +307ms
  worker:b doing some work +814ms
  worker:b doing some work +58ms
  worker:a doing lots of uninteresting work +312ms
  worker:a doing lots of uninteresting work +647ms
  worker:a doing lots of uninteresting work +469ms
  worker:b doing some work +1s
  worker:a doing lots of uninteresting work +797ms
  worker:a doing lots of uninteresting work +153ms
  worker:b doing some work +1s
  worker:a doing lots of uninteresting work +491ms
  worker:a doing lots of uninteresting work +323ms
  worker:b doing some work +602ms
```

Loggers: Debug

Usage

- debug exposes a function
- simply pass this function the name of your module, and it will return a decorated version of console.error for you to pass debug statements to
- this will allow you to toggle the debug output for different parts of your module as well as the module as a whole

```
const debug = require('debug')('my-namespace')
const name = 'my-app'
debug('booting %s', name)
```

Loggers: Debug

```
1   const express = require('express'),
2       app = express(),
3       debug = require('debug')('app:server');
4
5   debug('booting app');
6   app.get('/', require('./handler'))
7       .listen(3000, function () {
8           debug('listening');
9       });
10
```

```
1   const debug = require('debug')('app:handler');
2
3   module.exports = function (req, res) {
4       debug(req.method + ' ' + req.url);
5       res.end('hello\n');
6   }
7
8
9
10
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
→ DEBUG=app:* node 3.logging.js
app:server booting app +0ms
app:server listening +13ms
app:handler GET / +0ms
app:handler GET / +2s
```

Loggers: Debug

Features

- Simplest logger with minimum dependencies
- Can create multiple loggers with different IDs
- No log levels but can be added with “debug-levels” package
- Outputs to the standard error stream
- Mostly used for debugging purposes

Loggers: Bunyan

Bunyan - a simple and fast JSON logging library for node.js services and a CLI tool for nicely viewing those logs

Loggers: Bunyan

Usage

- bunyan exposes an object
- create a Logger instance; all loggers must provide a "name"
- call methods named after the logging levels

```
var bunyan = require('bunyan');
var log = bunyan.createLogger({name: "myapp"});
log.info("hi");
```

```
// hi.js
var bunyan = require('bunyan');
var log = bunyan.createLogger({name: 'myapp'});
log.info('hi');
log.warn({lang: 'fr'}, 'au revoir');
```

```
[2012-06-20T19:08:39.315Z]  WARN: amon-agent/3889 on headnode: error getting probe dat
a (continuing, presuming no probes)
    Error: connect ENOENT
        at errnoException (net.js:670:11)
        at Object.afterConnect [as oncomplete] (net.js:661:19)
[2012-06-20T19:08:39.315Z]  INFO: amon-agent/3889 on headnode: updated probes (numProb
es=0)
    changes: {
      "added": 0,
      "deleted": 0,
      "updated": 0,
      "errors": 0
    }
```

```
$ node hi.js
{"name":"myapp","hostname":"banana.local","pid":40161,"level":30,"msg":"hi","time":"2013-01-04T18:46:23.851Z","v":0}
{"name":"myapp","hostname":"banana.local","pid":40161,"level":40,"lang":"fr","msg":"au revoir","time":"2013-01-04T18:46:23.853Z","v":0}
```

Loggers: Bunyan

Manifesto

Server logs should be structured. JSON's a good format. Let's do that. A log record is one line of JSON.stringify’d output. Let's also specify some common names for the requisite and common fields for a log record.

Loggers: Bunyan

Features

- elegant log method API
- extensible streams system for controlling where log records go
- CLI for pretty-printing and filtering of Bunyan logs
- simple include of log call source location (file, line, function) with “src: true”
- lightweight specialization of Logger instances with “log.child”
- custom rendering of logged objects with “serializers”
- Support for a few runtime environments: Node.js, Browserify, Webpack, NW.js

Loggers: Winston

Winston – a multi-transport async logging library for Node.js. Simply put - a logger for just about everything!

Loggers: Winston

Usage

- winston exposes an object
- You can use its methods named after the logging levels right away
- Or create your own logger, the simplest way to do this is using “winston.createLogger”

```
1   const logger = require('winston');
2   module.exports = function (req, res) {
3       logger.info('Request: ' + req.method + ' ' + req.url);
4       if (req.path === '/cats' || req.path === '/dogs') {
5           logger.debug('IP: ' + req.ip);
6           res.end('hello\n');
7           return;
8       }
9       logger.error(req.path + ' - unknown route');
10      res.status(404).end('Not found');
11  }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
o → node 5.logging-winston.js
2017-09-24T12:26:51.427Z - info: Got message: GET /cats
2017-09-24T12:26:55.722Z - info: Got message: GET /dogs
2017-09-24T12:26:58.936Z - info: Got message: GET /flies
2017-09-24T12:26:58.936Z - error: /flies - unknown route
```

```
const winston = require('winston');

const logger = winston.createLogger({
  level: 'info',
  format: winston.format.json(),
  defaultMeta: { service: 'user-service' },
  transports: [
    //
    // - Write all logs with level `error` and below to `error.log`
    // - Write all logs with level `info` and below to `combined.log`
    //
    new winston.transports.File({ filename: 'error.log', level: 'error' }),
    new winston.transports.File({ filename: 'combined.log' })
  ]
});

//
// If we're not in production then log to the `console` with the format:
// `${info.level}: ${info.message} JSON.stringify({ ...rest }) `
//
if (process.env.NODE_ENV !== 'production') {
  logger.add(new winston.transports.Console({
    format: winston.format.simple()
  }));
}
```

Loggers: Winston

```
let transports = [
    new winston.transports.Console({
        timestamp: true,
        colorize: true,
        level: 'info'
    }),
    new winston.transports.File({
        filename: 'debug.log',
        name: 'debug',
        level: 'debug'
    }),
    new winston.transports.File({
        filename: 'error.log',
        name: 'error',
        level: 'error'
    })];
return new winston.Logger({transports: transports});
```

debug.log

```
1  {"level":"info","message":"Request: GET /cats",
   "timestamp":"2017-09-24T12:36:45.055Z"}
2  {"level":"debug","message":"IP: ::1",
   "timestamp":"2017-09-24T12:36:45.057Z"}
3  {"level":"info","message":"Request: GET /dogs",
   "timestamp":"2017-09-24T12:36:48.428Z"}
4  {"level":"debug","message":"IP: ::1",
   "timestamp":"2017-09-24T12:36:48.429Z"}
5  {"level":"info","message":"Request: GET /flies",
   "timestamp":"2017-09-24T12:36:51.237Z"}
6  {"level":"error","message":"/flies - unknown route",
   "timestamp":"2017-09-24T12:36:51.237Z"}
```

error.log

```
1  {"level":"error","message":"/flies - unknown route",
   "timestamp":"2017-09-24T12:36:51.237Z"}
```

Loggers: Winston

```
9   let transports = [
10      new winston.transports.Console({
11          timestamp: function () {
12              return Date.now();
13          },
14          formatter: function (options) {
15              return 'New format! ' + options.timestamp() + ' ' + options.level.toUpperCase() +
16                  ' ' + (options.message ? options.message : '');
17          }
18      }),
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
o → node 5.logging-winston.js
New format! 1506261180965 INFO Request: GET /cats
New format! 1506261184488 INFO Request: GET /dogs
New format! 1506261187732 INFO Request: GET /flies
New format! 1506261187733 ERROR /flies - unknown route
```

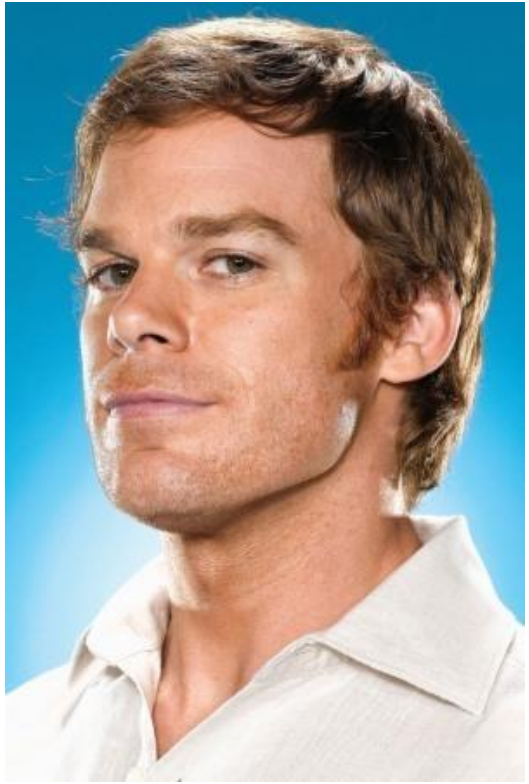
Loggers: Winston

Features

- uses Node.js streams for performant way of chunking up data processing, e.g. data manipulation on large amounts of data extensible streams system for controlling where log records go
- you may also want to stream your logs to a central location or store them in a database for querying later. Winston allows you to do that easily with a concept called transports and you can have as many transports as you like
- has some advanced features for formatting log code before stashing it away, e.g. offers JSON formatting, coloring log output, and the ability to fiddle with formats before they’re posted off to your end log locations
- adds log level layer as metadata for logs. Log levels tell you the severity, ranging from as severe as a system outage to as small as a deprecated function.

Loggers: Morgan

Morgan - a great logging tool as a middleware that allows us to easily log requests, errors and more to the console.



Loggers: Morgan

Usage

- morgan exposes a function with additional configurational methods on its API
- create a new logger middleware function using the given “format” and “options”
- the function will be called with three arguments ”token”, “req” and “res”
- use either predefined formats or create your own

```
// EXAMPLE: only log error responses
morgan('combined', {
  skip: function (req, res) { return res.statusCode < 400 }
})
```

```
morgan.token('id', function getId(req) {
    return req.id
});

var loggerFormat = ':id [:date[web]] ":method :url" :status :response-time';

app.use(morgan(loggerFormat, {
    skip: function (req, res) {
        return res.statusCode < 400
    },
    stream: process.stderr
}));
```

Loggers: Morgan

Supported Morgan Tokens

- :date[format]
  - CLF for the common log format
  - ISO (ISO 8601)
  - web (default, RFC 1123)
- :http-version
- :method
- :referrer
- :remote-addr
- :remote-user
- :req[header]
- :res[header]
- :response-time[digits]
- :status
- :url
- :user-agent

Loggers: Morgan

Features

- ideal for middleware-backed libraries like Express.js
- written as middleware
- very easy to set up and use out of the box
- extensible through the use of streams
- only has a small performance impact
- configurable format using predefined tokens
- perfect choice for logging requests in areas such as web apps, audit logs, and debugging

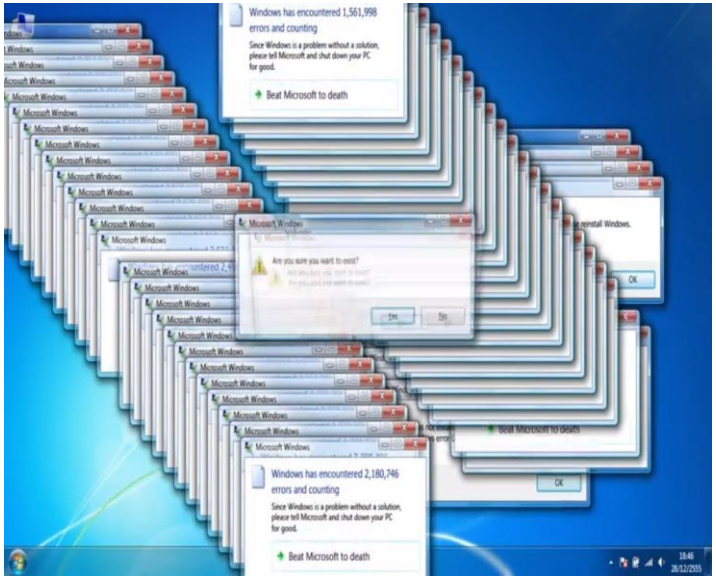
Error Handling

Errors

Every programming language has them under one name or another, and the Node.js environment is no different.

Node.js includes only a handful of predefined errors (like RangeError, SyntaxError, and others), all of which inherit from Error.

Note that the official documentation for Node.js refers to exceptions as well, but don’t be confused: exceptions are targets of a throw statement, so essentially, they are errors.



Error Handling: Types of Errors

OPERATIONAL ERRORS

- failed to connect to server
- failed to resolve hostname
- invalid user input
- request timeout
- server returned a 500 response
- socket hang-up
- system is out of memory

SHOULD BE HANDLED

PROGRAMMING ERRORS

- tried to read property of "undefined"
- called an asynchronous function without a callback
- passed a "string" where an object was expected
- passed an object where an IP address string was expected

SHOULD NOT BE HANDLED

Error Handling: Options

- **Throw an error and generate exception**
  Only used in synchronous scenarios, mostly when parsing data.
- **Invoke an error-first callback**
  Typical for standard Node modules and most of Node applications.
- **Reject a promise**
  Any callback-based function can be promisified.
- **Generate an error event**
  Used when working with EventEmitters and in larger applications.

Error Handling: Options: Try...Catch Blocks

```
1   try {
2       JSON.parse('Not a JSON!');
3   } catch(e) {
4       console.log('parsing error');
5   }
6
7   try {
8       setTimeout(() => {
9           JSON.parse('Not a JSON!');
10      }, 1000);
11  } catch(e) {
12     console.log('callback from error');
13  }
```

```
○ → node 6.error-handling.js
parsing error
undefined:1
Not a JSON!
^

SyntaxError: Unexpected token N in JSON at position 0
    at JSON.parse (<anonymous>)
    at Timeout.setTimeout [as _onTimeout] (/Users/gal
ina_kasatkina/Documents/lecture/6.error-handling.js:9
:14)
    at ontimeout (timers.js:365:14)
    at tryOnTimeout (timers.js:237:5)
    at Timer.listOnTimeout (timers.js:207:5)
```

Error Handling: Options: Error-First Callbacks

```
1   const fs = require('fs');
2   fs.readFile('nonexistent', (err, data) => {
3       if (err) {
4           console.log(err);
5           return;
6       }
7       //do sth
8   });
```

Error Handling: Options: Promise Rejection

```
1   const promisify = require("util").promisify;
2   const fs = require('fs');
3   const readFile = promisify(fs.readFile);
4
5   readFile('nonexistent')
6       .then((data) => {
7           JSON.parse(data);
8       }).catch((err) => {
9           console.log(err);
10      });
```

Error Handling: Options: Error Events

```
1   const http = require('http');
2   const server = http.createServer((req, res) => {
3     res.end('Hello!')
4   });
5
6   server.on('error', (err) => {
7       console.error('ERROR!!!');
8       console.error(err);
9   });
10
```

```
ERROR!!!
{ Error: listen EACCES 0.0.0.0:80
    at Object._errnoException (util.js:1026:11)
    at _exceptionWithHostPort (util.js:1049:20)
    at Server.setupListenHandle [as _listen2] (net.js:1
326:19)
    at listenInCluster (net.js:1391:12)
    at Server.listen (net.js:1474:7)
    at Object.<anonymous> (/Users/galina_kasatkina/Docu
ments/lecture/7.error-handling-events.1.js:11:8)
    at Module._compile (module.js:624:30)
    at Object.Module._extensions..js (module.js:635:10)
    at Module.load (module.js:545:32)
    at tryModuleLoad (module.js:508:12)
  code: 'EACCES',
  errno: 'EACCES',
  syscall: 'listen',
  address: '0.0.0.0',
  port: 80 }
```

Error Handling: Uncaught Exceptions

- Can be caught using *process.on('uncaughtException')*.
- What you SHOULD NOT do in the handler:
  - Attempt to restore the program's normal operation
- What you SHOULD do:
  - Log errors,
  - Free all resources,
  - Exit the process with an appropriate error code.

SUMMARY Q&A

Useful links:

- Logging in Node.js
- Error Handling in Node.js
- Logging Best Practices
- Error Handling Patterns

QUESTIONS / IDEAS