BUILDING WITH NODE.JS

Everything you need to build and scale up a Node app properly

(Third Part of the Node.js at Scale Series)

From the Engineers of trace by RisingStack

Table of contents

CHAPTER ONE:
PROJECT STRUCTURING

Project structuring is an important topic because the way you bootstrap your application can determine the whole development experience throughout the life of the project.

In this chapter I'll answer some of the most common questions we receive at RisingStack about structuring advanced Node applications, and help you with structuring a complex project.

These are the goals that we are aiming for:

* Writing an application that is easy to scale and maintain.
* The config is well separated from the business logic.
* Our application can consist of multiple process types.

THE NODE.JS PROJECT STRUCTURE

Our example application is listening on Twitter tweets and tracks certain keywords. In case of a keyword match, the tweet will be sent to a RabbitMQ queue, which will be processed and saved to Redis. We will also have a REST API exposing the tweets we have saved.

You can take a look at the code on GitHub. The file structure for this project looks like the following (head over to next page):

```
.
|-- config
|   |-- components
|   |   |-- common.js
|   |   |-- logger.js
|   |   |-- rabbitmq.js
|   |   |-- redis.js
|   |   |-- server.js
|   |   `-- twitter.js
|   |-- index.js
|   |-- social-preprocessor-worker.js
|   |-- twitter-stream-worker.js
|   `-- web.js
|-- models
|   |-- redis
|   |   |-- index.js
|   |   `-- redis.js
|   |-- tortoise
|   |   |-- index.js
|   |   `-- tortoise.js
|   `-- twitter
|       |-- index.js
|       `-- twitter.js
|-- scripts
|-- test
|   `-- setup.js
|-- web
|   |-- middleware
|   |   |-- index.js
|   |   `-- parseQuery.js
|   |-- router
|   |   |-- api
|   |   |   |-- tweets
|   |   |   |   |-- get.js
|   |   |   |   |-- get.spec.js
|   |   |   |   `-- index.js
|   |   |   `-- index.js
|   |   `-- index.js
|   |-- index.js
|   `-- server.js
|-- worker
|   |-- social-preprocessor
|   |   |-- index.js
|   |   `-- worker.js
|   `-- twitter-stream
|       |-- index.js
|       `-- worker.js
|-- index.js
`-- package.json
```

In this example we have 3 processes:

* `twitter-stream-worker`: The process is listening on Twitter for keywords and sends the tweets to a RabbitMQ queue.
* `social-preprocessor-worker`: The process is listening on the RabbitMQ queue and saves the tweets to Redis and removes old ones.
* `web`: The process is serving a REST API with a single endpoint: GET /api/v1/tweets?limit&offset.

We will get to what differentiates a web and a worker process, but let's start with the config.

HOW TO HANDLE DIFFERENT ENVIRONMENTS AND CONFIGURATIONS?

Load your deployment specific configurations from environment variables and never add them to the codebase as constants. These are the configurations that can vary between deployments and runtime environments, like CI, staging or production. Basically, you can have the same code running everywhere.

A good test for whether the config is correctly separated from the application internals is that the codebase could be made public at any moment. This means that you can be protected from accidentally leaking secrets or compromising credentials on version control.

The environment variables can be accessed via the `process.env` object. Keep in mind that all the values have a type of `String`, so you might need to use type conversions.

```
// config/config.js
'use strict'

// required environment variables
[
  'NODE_ENV',
  'PORT'
].forEach((name) => {
  if (!process.env[name]) {
    throw new Error(`Environment variable ${name} is missing`)
  }
})

const config = {
  env: process.env.NODE_ENV,
  logger: {
    level: process.env.LOG_LEVEL || 'info',
    enabled: process.env.BOOLEAN ?
process.env.BOOLEAN.toLowerCase() === 'true' : false
  },
  server: {
    port: Number(process.env.PORT)
  }
  // ...
}

module.exports = config
```

CONFIG VALIDATION

Validating environment variables is also a quite useful technique. It can help you catching configuration errors on startup before your application does anything else. You can read more about the benefits of early error detection of configurations by Adrian Colyer in this blog post.

This is how our improved config file looks like with schema validation using the `joi` validator:

```
// config/config.js
'use strict'

const joi = require('joi')

const envVarsSchema = joi.object({
  NODE_ENV: joi.string()
    .allow(['development', 'production', 'test', 'provision'])
    .required(),
  PORT: joi.number()
    .required(),
  LOGGER_LEVEL: joi.string()
    .allow(['error', 'warn', 'info', 'verbose', 'debug',
'silly'])
    .default('info'),
  LOGGER_ENABLED: joi.boolean()
    .truthy('TRUE')
    .truthy('true')
    .falsy('FALSE')
    .falsy('false')
    .default(true)
}).unknown()
  .required()

const { error, value: envVars } = joi.validate(process.env,
envVarsSchema)
if (error) {
  throw new Error(`Config validation error: ${error.message}`)
}

const config = {
  env: envVars.NODE_ENV,
  isTest: envVars.NODE_ENV === 'test',
  isDevelopment: envVars.NODE_ENV === 'development',
  logger: {
    level: envVars.LOGGER_LEVEL,
    enabled: envVars.LOGGER_ENABLED
  },
  server: {
    port: envVars.PORT
  }
  // ...
}

module.exports = config
```

CONFIG SPLITTING

Splitting the configuration by components can be a good solution to forego a single, growing config file.

```
// config/components/logger.js
'use strict'

const joi = require('joi')

const envVarsSchema = joi.object({
  LOGGER_LEVEL: joi.string()
    .allow(['error', 'warn', 'info', 'verbose', 'debug',
'silly'])
    .default('info'),
  LOGGER_ENABLED: joi.boolean()
    .truthy('TRUE')
    .truthy('true')
    .falsy('FALSE')
    .falsy('false')
    .default(true)
}).unknown()
  .required()

const { error, value: envVars } = joi.validate(process.env,
envVarsSchema)
if (error) {
  throw new Error(`Config validation error: ${error.message}`)
}

const config = {
  logger: {
    level: envVars.LOGGER_LEVEL,
    enabled: envVars.LOGGER_ENABLED
  }
}

module.exports = config
```

Then in the `config.js` file we only need to combine the components.

```
// config/config.js
'use strict'

const common = require('./components/common')
const logger = require('./components/logger')
const redis = require('./components/redis')
const server = require('./components/server')

module.exports = Object.assign({}, common, logger, redis,
server)
```

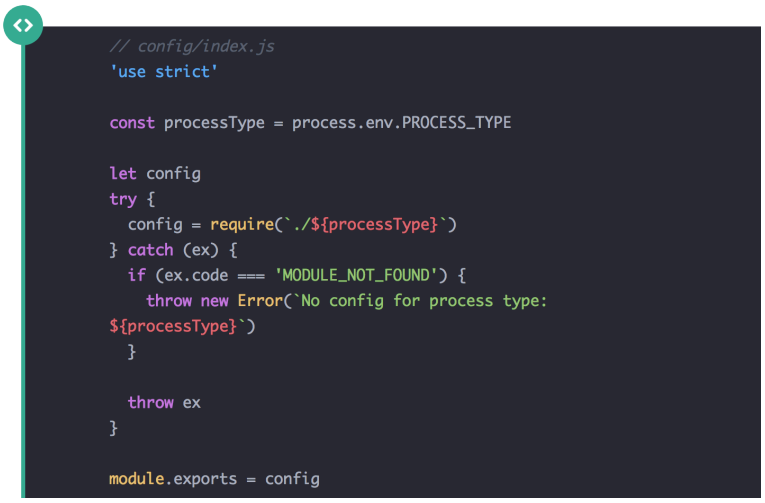
You should never group your config together into “environment” specific files, like `config/production.js` for production. It doesn’t scale well as your app expands into more deployments over time.

HOW TO ORGANIZE A MULTI-PROCESS APPLICATION?

The process is the main building block of a modern application. An app can have multiple stateless processes, just like in our example. HTTP requests can be handled by a web process and long-running or scheduled background tasks by a worker. They are stateless, because any data that needs to be persisted is stored in a stateful database. For this reason, adding more concurrent processes are very simple. These processes can be independently scaled based on the load or other metrics.

In the previous section, we saw how to break down the config into components. This comes very handy when having different process types. Each type can have its own config only requiring the components it needs, without expecting unused environment variables.

In the `config/index.js` file:



```
// config/index.js
'use strict'

const processType = process.env.PROCESS_TYPE

let config
try {
  config = require(`./${processType}`)
} catch (ex) {
  if (ex.code === 'MODULE_NOT_FOUND') {
    throw new Error(`No config for process type:
${processType}`)
  }

  throw ex
}

module.exports = config
```

In the root `index.js` file we start the process selected with the `PROCESS_TYPE` environment variable:

```
// index.js
'use strict'

const processType = process.env.PROCESS_TYPE

if (processType === 'web') {
  require('./web')
} else if (processType === 'twitter-stream-worker') {
  require('./worker/twitter-stream')
} else if (processType === 'social-preprocessor-worker') {
  require('./worker/social-preprocessor')
} else {
  throw new Error(`${processType} is an unsupported process
type. Use one of: 'web', 'twitter-stream-worker', 'social-
preprocessor-worker'!`)
}
```

The nice thing about this is that we still got one application, but we have managed to split it into multiple, independent processes. Each of them can be started and scaled individually, without influencing the other parts. You can achieve this without sacrificing your DRY codebase, because parts of the code, like the models, can be shared between the different processes.

How to organize your test files?

Place your test files next to the tested modules using some kind of naming convention, like `<module_name>.spec.js` and `<module_name>.e2e.spec.js`. Your tests should live together with the tested modules, keeping them in sync. It would be really hard to find and maintain the tests and the corresponding functionality when the test files are completely separated from the business logic.

A separated `/test` folder can hold all the additional test setup and utilities not used by the application itself.

WHERE TO PUT YOUR BUILD AND SCRIPT FILES?

We tend to create a `/scripts` folder where we put our bash and node scripts for database synchronization, front-end builds and so on. This folder separates them from your application code and prevents you from putting too many script files into the root directory. List them in your npm scripts for easier usage.

CHAPTER TWO:
CLEAN CODING

Writing clean code is what you must know and do in order to call yourself a professional developer. There is no reasonable excuse for doing anything less than your best.

In this chapter, we will cover general clean coding principles for naming and using variables & functions, as well as some JavaScript specific clean coding best practices.

"Even bad code can function. But if the code isn't clean, it can bring a development organization to its knees."

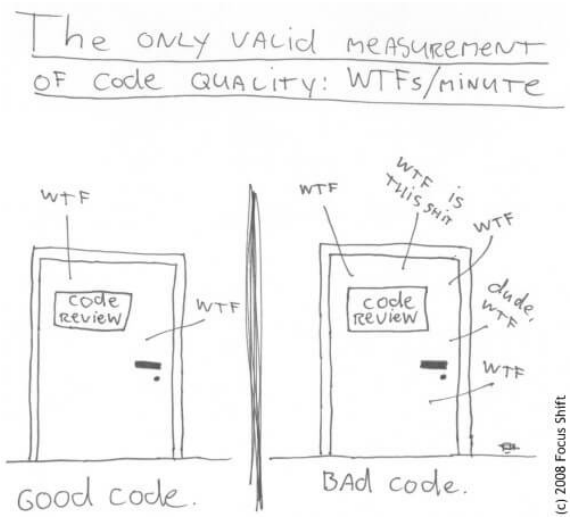
— Robert C. Martin (Uncle Bob)

FIRST OF ALL, WHAT DOES CLEAN CODING MEAN?

Clean coding means that in the first place you write code for your later self and for your co-workers and not for the machine.

Your code must be easily understandable for humans.

You know you are working on a clean code when each routine you read turns out to be pretty much what you expected.



Now that we know what every developer should aim for, let's go through the best practices!

How should I name my variables?

Use intention-revealing names and don't worry if you have long variable names instead of saving a few keyboard strokes.

If you follow this practice, your names become searchable, which helps a lot when you do refactors or you are just looking for something.

```
// DON'T
let d
let elapsed
const ages = arr.map((i) => i.age)

// DO
let daysSinceModification
const agesOfUsers = users.map((user) => user.age)
```

Also, make meaningful distinctions and don't add extra, unnecessary nouns to the variable names.

```
// DON'T
let nameString
let theUsers

// DO
let name
let users
```

Make your variable names easy to pronounce, because for the human mind it takes less effort to process.

When you are doing code reviews with your fellow developers, these names are easier to reference.

```
// DON'T
let fName, lName
let cntr

let full = false
if (cart.size > 100) {
  full = true
}

// DO
let firstName, lastName
let counter

const MAX_CART_SIZE = 100
// ...
const isFull = cart.size > MAX_CART_SIZE
```

In short, don't cause extra mental mapping with your names.

How should I write my functions?

Your functions should do one thing only on one level of abstraction.

Functions should do one thing. They should do it well.
They should do it only.

— Robert C. Martin (Uncle Bob)

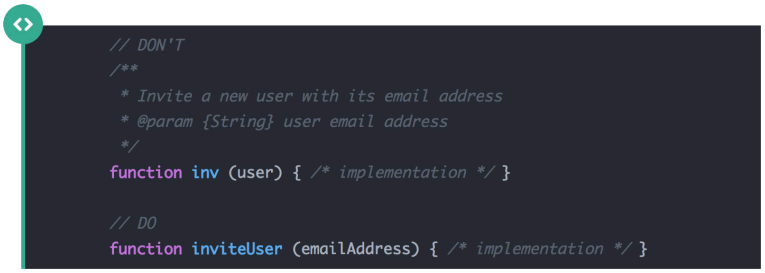
```
// DON'T
function getUserRouteHandler (req, res) {
  const { userId } = req.params
  // inline SQL query
  knex('user')
    .where({ id: userId })
    .first()
    .then((user) => res.json(user))
}

// DO
// User model (eg. models/user.js)
const tableName = 'user'
const User = {
  getOne (userId) {
    return knex(tableName)
      .where({ id: userId })
      .first()
  }
}

// route handler (eg. server/routes/user/get.js)
function getUserRouteHandler (req, res) {
  const { userId } = req.params
  User.getOne(userId)
    .then((user) => res.json(user))
}
```

After you wrote your functions properly, you can test how well you did with CPU profiling - which helps you to find bottlenecks.

Use long, descriptive names

A function name should be a verb or a verb phrase, and it needs to communicate its intent, as well as the order and intent of the arguments.

A long descriptive name is way better than a short, enigmatic name or a long descriptive comment.



```
// DON'T
/**
 * Invite a new user with its email address
 * @param {String} user email address
 */
function inv (user) { /* implementation */ }

// DO
function inviteUser (emailAddress) { /* implementation */ }
```

Avoid long argument list

Use a single object parameter and destructuring assignment instead. It also makes handling optional parameters much easier.

```
// DON'T
function getRegisteredUsers (fields, include, fromDate,
toDate) { /* implementation */ }
getRegisteredUsers(['firstName', 'lastName', 'email'],
['invitedUsers'], '2016-09-26', '2016-12-13')

// DO
function getRegisteredUsers ({ fields, include, fromDate,
toDate }) { /* implementation */ }
getRegisteredUsers({
  fields: ['firstName', 'lastName', 'email'],
  include: ['invitedUsers'],
  fromDate: '2016-09-26',
  toDate: '2016-12-13'
})
```

Reduce side effects

Use pure functions without side effects, whenever you can. They are really easy to use and test.

```
// DON'T
function addItemToCart (cart, item, quantity = 1) {
  const alreadyInCart = cart.get(item.id) || 0
  cart.set(item.id, alreadyInCart + quantity)
  return cart
}

// DO
// not modifying the original cart
function addItemToCart (cart, item, quantity = 1) {
  const cartCopy = new Map(cart)
  const alreadyInCart = cartCopy.get(item.id) || 0
  cartCopy.set(item.id, alreadyInCart + quantity)
  return cartCopy
}

// or by invert the method location
// you can expect that the original object will be mutated
// addItemToCart(cart, item, quantity) -> cart.addItem(item,
quantity)
const cart = new Map()
Object.assign(cart, {
  addItem (item, quantity = 1) {
    const alreadyInCart = this.get(item.id) || 0
    this.set(item.id, alreadyInCart + quantity)
    return this
  }
})
```

Organize your functions in a file according to the stepdown rule

Higher level functions should be on top and lower levels below. It makes it natural to read the source code.

```
// DON'T
// "I need the full name for something..."
function getFullName (user) {
  return `${user.firstName} ${user.lastName}`
}

function renderEmailTemplate (user) {
  // "oh, here"
  const fullName = getFullName(user)
  return `Dear ${fullName}, ...`
}

// DO
function renderEmailTemplate (user) {
  // "I need the full name of the user"
  const fullName = getFullName(user)
  return `Dear ${fullName}, ...`
}

// "I use this for the email template rendering"
function getFullName (user) {
  return `${user.firstName} ${user.lastName}`
}
```

Query or modification

Functions should either do something (modify) or answer something (query), but not both.

EVERYONE LIKES TO WRITE JAVASCRIPT DIFFERENTLY, WHAT TO DO?

As JavaScript is dynamic and loosely typed, it is especially prone to programmer errors.

Use project or company wise linter rules and formatting style.

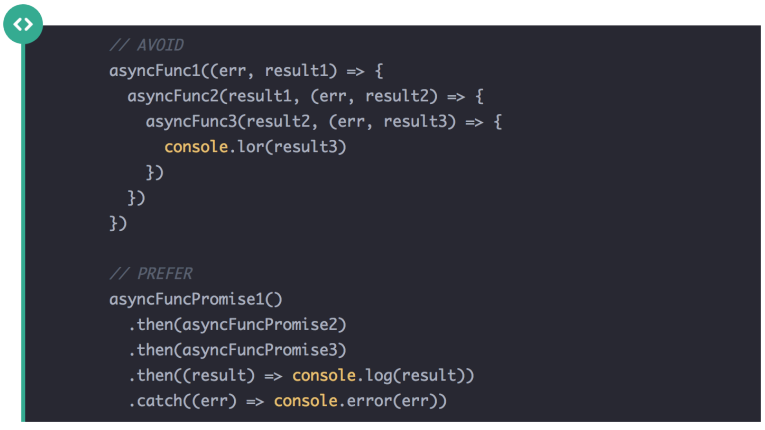
The stricter the rules, the less effort will go into pointing out bad formatting in code reviews. It should cover things like consistent naming, indentation size, whitespace placement and even semicolons.

The standard JS style is quite nice to start with, but in my opinion, it isn't strict enough. I can agree most of the rules in the Airbnb style.

HOW TO WRITE NICE ASYNC CODE?

Use Promises whenever you can.

Promises are natively available from Node 4. Instead of writing nested callbacks, you can have chainable Promise calls.



```
// AVOID
asyncFunc1((err, result1) => {
  asyncFunc2(result1, (err, result2) => {
    asyncFunc3(result2, (err, result3) => {
      console.lor(result3)
    })
  })
})

// PREFER
asyncFuncPromise1()
  .then(asyncFuncPromise2)
  .then(asyncFuncPromise3)
  .then((result) => console.log(result))
  .catch((err) => console.error(err))
```

Most of the libraries out there have both callback and promise interfaces, prefer the latter. You can even convert callback APIs to promise based one by wrapping them using packages like es6-promisify.

```
// AVOID
const fs = require('fs')

function readJSON (filePath, callback) {
  fs.readFile(filePath, (err, data) => {
    if (err) {
      return callback(err)
    }

    try {
      callback(null, JSON.parse(data))
    } catch (ex) {
      callback(ex)
    }
  })
}

readJSON('./package.json', (err, pkg) => { console.log(err,
pkg) })

// PREFER
const fs = require('fs')
const promisify = require('es6-promisify')

const readFile = promisify(fs.readFile)
function readJSON (filePath) {
  return readFile(filePath)
    .then((data) => JSON.parse(data))
}

readJSON('./package.json')
  .then((pkg) => console.log(pkg))
  .catch((err) => console.error(err))
```

The next step would be to use `async/await` (≥ Node 7) or `generators` with `co` (≥ Node 4) to achieve synchronous like control flows for your asynchronous code.

```
const request = require('request-promise-native')

function getExtractFromWikipedia (title) {
  return request({
    uri: 'https://en.wikipedia.org/w/api.php',
    qs: {
      titles: title,
      action: 'query',
      format: 'json',
      prop: 'extracts',
      exintro: true,
      explaintext: true
    },
    method: 'GET',
    json: true
  })
    .then((body) => Object.keys(body.query.pages).map((key) =>
body.query.pages[key].extract))
    .then((extracts) => extracts[0])
    .catch((err) => {
      console.error('getExtractFromWikipedia() error:', err)
      throw err
    })
}

// PREFER
async function getExtractFromWikipedia (title) {
  let body
  try {
    body = await request({ /* same parameters as above */ })
  } catch (err) {
    console.error('getExtractFromWikipedia() error:', err)
    throw err
  }

  const extracts = Object.keys(body.query.pages).map((key) =>
body.query.pages[key].extract)
  return extracts[0]
}

// or
const co = require('co')

const getExtractFromWikipedia = co.wrap(function * (title) {
  let body
  try {
    body = yield request({ /* same parameters as above */ })
  } catch (err) {
    console.error('getExtractFromWikipedia() error:', err)
    throw err
  }

  const extracts = Object.keys(body.query.pages).map((key) =>
body.query.pages[key].extract)
  return extracts[0]
})

getExtractFromWikipedia('Robert Cecil Martin')
  .then((robert) => console.log(robert))
```

HOW SHOULD I WRITE PERFORMANT CODE?

In the first place, you should write clean code, then use profiling to find performance bottlenecks.

Never try to write performant and smart code first, instead, optimize the code when you need to and refer to true impact instead of micro-benchmarks.

Although, there are some straightforward scenarios like eagerly initializing what you can (eg. joi schemas in route handlers, which would be used in every request and adds serious overhead if recreated every time) and using asynchronous instead of blocking code.

CHAPTER THREE:
ASYNC BEST PRACTICES

In this chapter, we cover what tools and techniques you have at your disposal when handling Node.js asynchronous operations: async.js, promises, generators and async functions. After reading this part, you'll know how to avoid the despised callback hell!

Asynchronous programming in Node.js

Previously we have gathered a strong knowledge about asynchronous programming in JavaScript and understood how the Node.js event loop works.

If you did not read these articles, I highly recommend them as introductions!

THE PROBLEM WITH NODE.JS ASYNC

Node.js itself is single threaded, but some tasks can run parallelly - thanks to its asynchronous nature.

But what does running parallelly mean in practice?

Since we program a single threaded VM, it is essential that we do not block execution by waiting for I/O, but handle them concurrently with the help of Node.js's event driven APIs.

Let's take a look at some fundamental patterns, and learn how we can write resource efficient, non-blocking code, with the built-in solutions of Node.js and some third-party libraries.

THE CLASSICAL APPROACH - CALLBACKS

Let's take a look at these simple async operations. They do nothing special, just fire a timer and call a function once the timer finished.

```
function fastFunction (done) {
  setTimeout(function () {
    done()
  }, 100)
}

function slowFunction (done) {
  setTimeout(function () {
    done()
  }, 300)
}
```

Seems easy, right?

Our higher-order functions can be executed sequentially or parallelly with the basic “pattern” by nesting callbacks - but using this method can lead to an untameable callback-hell.

```
function runSequentially (callback) {
  fastFunction((err, data) => {
    if (err) return callback(err)
    console.log(data)   // results of a

    slowFunction((err, data) => {
      if (err) return callback(err)
      console.log(data) // results of b

      // here you can continue running more tasks
    })
  })
}
```

Never use the nested callback approach for handling asynchronous Node.js operations!

AVOIDING CALLBACK HELL WITH CONTROL FLOW MANAGERS

To become an efficient Node.js developer, you have to avoid the constantly growing indentation level, produce clean and readable code and be able to handle complex flows.

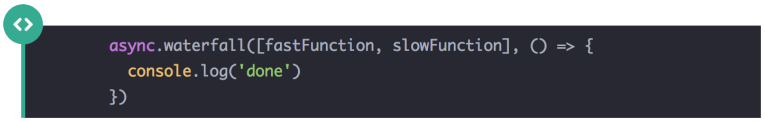
Let us show you some of the libraries we can use to organize our code in a nice and maintainable way!

#1: Meet the Async Module

Async is a utility module which provides straight-forward, powerful functions for working with asynchronous JavaScript.

Async contains some common patterns for asynchronous flow control with the respect of error-first callbacks.

Let's see how our previous example would look like using async!



```
async.waterfall([fastFunction, slowFunction], () => {
  console.log('done')
})
```

What kind of witchcraft just happened?

Actually, there is no magic to reveal. You can easily implement your async job-runner which can run tasks parallelly and wait for each to be ready.

Let's take a look at what async does under the hood!

```
// taken from
https://github.com/caolan/async/blob/master/lib/waterfall.js
function(tasks, callback) {
    callback = once(callback || noop);
    if (!isArray(tasks)) return callback(new Error('First
argument to waterfall must be an array of functions'));
    if (!tasks.length) return callback();
    var taskIndex = 0;

    function nextTask(args) {
        if (taskIndex === tasks.length) {
            return callback.apply(null, [null].concat(args));
        }

        var taskCallback = onlyOnce(rest(function(err, args) {
            if (err) {
                return callback.apply(null,
[err].concat(args));
            }
            nextTask(args);
        }));

        args.push(taskCallback);

        var task = tasks[taskIndex++];
        task.apply(null, args);
    }

    nextTask([]);
}
```

Essentially, a new callback is injected into the functions, and this is how async knows when a function is finished.

#2: Using co - generator based flow-control for Node.js

In case you wouldn't like to stick to the solid callback protocol, then co can be a good choice for you.

co is a generator based control flow tool for Node.js and the browser, using promises, letting you write non-blocking code in a nice-ish way.

co is a powerful alternative which takes advantage of generator functions tied with promises without the overhead of implementing custom iterators.

```
const fastPromise = new Promise((resolve, reject) => {
  fastFunction(resolve)
})

const slowPromise = new Promise((resolve, reject) => {
  slowFunction(resolve)
})

co(function * () {
  yield fastPromise
  yield slowPromise
}).then(() => {
  console.log('done')
})
```

As for now, I suggest to go with co, since one of the most waited Node.js async/await functionality is only available in the nightly, unstable v7.x builds. But if you are already using Promises, switching from co to async function will be easy.

This syntactic sugar on top of Promises and Generators will eliminate the problem of callbacks and even help you to build nice flow control structures. Almost like writing synchronous code, right?

Stable Node.js branches will receive this update in the near future, so you will be able to remove co and just do the same.

FLOW CONTROL IN PRACTICE

As we have just learned several tools and tricks to handle async, it is time to do some practice with fundamental control flows to make our code more efficient and clean.

Let's take an example and write a route `handler` for our web app, where the request can be resolved after 3 steps: `validateParams`, `dbQuery` and `serviceCall`.

If you'd like to write them without any helper, you'd most probably end up with something like this. Not so nice, right?

```
// validateParams, dbQuery, serviceCall are higher-order
functions
// DONT
function handler (done) {
  validateParams((err) => {
    if (err) return done(err)
    dbQuery((err, dbResults) => {
      if (err) return done(err)
      serviceCall((err, serviceResults) => {
        done(err, { dbResults, serviceResults })
      })
    })
  })
}
```

Instead of the callback-hell, we can use the async library to refactor our code, as we have already learned:

```
// validateParams, dbQuery, serviceCall are higher-order
functions
function handler (done) {
  async.waterfall([validateParams, dbQuery, serviceCall],
done)
}
```

Let's take it a step further! Rewrite it to use Promises:

```
// validateParams, dbQuery, serviceCall are thunks
function handler () {
  return validateParams()
    .then(dbQuery)
    .then(serviceCall)
    .then((result) => {
      console.log(result)
      return result
    })
}
```

Also, you can use co powered generators with Promises:

```
// validateParams, dbQuery, serviceCall are thunks
const handler = co.wrap(function * () {
  yield validateParams()
  const dbResults = yield dbQuery()
  const serviceResults = yield serviceCall()
  return { dbResults, serviceResults }
})
```

It feels like a “synchronous” code but still doing async jobs one after each other.

Lets see how this snippet should work with `async / await`.

```
// validateParams, dbQuery, serviceCall are thunks
async function handler () {
  await validateParams()
  const dbResults = await dbQuery()
  const serviceResults = await serviceCall()
  return { dbResults, serviceResults }
})
```

TAKEAWAY RULES FOR NODE.JS & ASYNC

Fortunately, Node.js eliminates the complexities of writing thread-safe code. You just have to stick to these rules to keep things smooth:

* As a rule of thumb, prefer async over sync API, because using a non-blocking approach gives superior performance over the synchronous scenario.

* Always use the best fitting flow control or a mix of them in order reduce the time spent waiting for I/O to complete.

You can find all of the code from this article in this repository.

CHAPTER FOUR: EVENT SOURCING

Event Sourcing is a powerful architectural pattern to handle complex application states that may need to be rebuilt, re-played, audited or debugged.

From this chapter you can learn what Event Sourcing is, and when should you use it. We'll also take a look at some Event sourcing examples with code snippets.

EVENT SOURCING

Event Sourcing is a software architecture pattern which makes it possible to reconstruct past states (latest state as well). It's achieved in a way that every state change gets stored as a sequence of events.

The State of your application is like a user's account balance or subscription at a particular time. This current state may only exist in memory.

Good examples for Event Sourcing are version control systems that stores current state as diffs. The current state is your latest source code, and events are your commits.

WHY IS EVENT SOURCING USEFUL?

In our hypothetical example, you are working on an online money transfer site, where every customer has an account balance. Imagine that you just started working on a beautiful Monday morning when it suddenly turns out that you made a mistake and used a wrong currency exchange for the whole past week. In this case, every account which sent and received money in a last seven days are in a corrupt state.

With event sourcing, there's no need to panic!

If your site uses event sourcing, you can revert the account balances to their previous uncorrupted state, fix the exchange rate and replay all the events until now. That's it, your job and reputation is saved!

Other use-cases

You can use events to audit or debug state changes in your system. They can also be useful for handling SaaS subscriptions. In a usual subscription based system, your users can buy a plan, upgrade it, downgrade it, pro-rate a current price, cancel a plan, apply a coupon, and so on... A good event log can be very useful to figure out what happened.

So with event sourcing you can:

* Rebuild states completely
* Replay states from a specific time
* Reconstruct the state of a specific moment for temporary query

WHAT IS AN EVENT?

An Event is something that happened in the past. An Event is not a snapshot of a state at a specific time; it's the action itself with all the information that's necessary to replay it.

Events should be a simple object which describes some action that occurred. They should be immutable and stored in an append-only way. Their immutable append-only nature makes them suitable to use as audit logs too.

For proper Event Sourcing, you must create an event for every state change & preserve the order of events as well. This is what makes possible to undo and redo events or even replay them from a specific timestamp.

Be careful with External Systems!

As any software pattern, Event Sourcing can be challenging at some points as well.

The external systems that your application communicates with are usually not prepared for event sourcing, so you should be careful when you replay your events. I'm sure that you don't wish to charge your customers twice or send all welcome emails again.

To solve this challenge, you should handle replays in your communication layers!

COMMAND SOURCING

Command Sourcing is a different approach from Event Sourcing - make sure you don't mix 'em up by accident!

Event Sourcing:

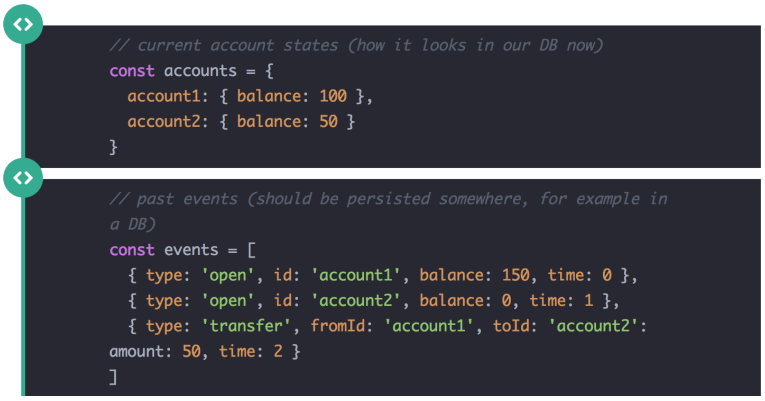
* Persist only changes in state
* Replay can be side-effect free

Command Sourcing:

* Persist Commands
* Replay may trigger side-effects

EXAMPLE FOR EVENT SOURCING

In this simple example, we will apply Event Sourcing for our accounts:



```
// current account states (how it looks in our DB now)
const accounts = {
  account1: { balance: 100 },
  account2: { balance: 50 }
}
```

```
// past events (should be persisted somewhere, for example in
a DB)
const events = [
  { type: 'open', id: 'account1', balance: 150, time: 0 },
  { type: 'open', id: 'account2', balance: 0, time: 1 },
  { type: 'transfer', fromId: 'account1', toId: 'account2':
amount: 50, time: 2 }
]
```

Let's rebuild the latest state from scratch, using our event log:

```
// complete rebuild
const accounts = events.reduce((accounts, event) => {
  if (event.type === 'open') {
    accounts[event.id].balance = event.balance
  } else if (event.type === 'transfer') {
    accounts[event.fromId].balance -= event.amount
    accounts[event.toId].balance += event.amount
  }
  return accounts
}, {})
```

Undo the latest event:

```
// undo last event
const accounts = events.splice(-1).reduce((accounts, event) =>
{
  if (event.type === 'open') {
    delete accounts[event.id]
  } else if (event.type === 'transfer') {
    accounts[event.fromId].balance += event.amount
    accounts[event.toId].balance -= event.amount
  }
  return accounts
}, {})
```

Query accounts state at a specific time:

```
// query specific time
function getAccountsAtTime (time) {
  return events.reduce((accounts, event) => {
    if (time > event.time {
      return accounts
    }

    if (event.type === 'open') {
      accounts[event.id].balance = event.balance
    } else if (event.type === 'transfer') {
      accounts[event.fromId].balance -= event.amount
      accounts[event.toId].balance += event.amount
    }
    return accounts
  }, {})
}

const accounts = getAccountsAtTime(1)
```

LEARNING MORE..

For more detailed examples, you can check out our Event Sourcing Example repository.

For more general and deeper understanding of Event Sourcing I recommend to read these articles:

* Martin Fowler - Event Sourcing
* MSDN - Event Sourcing Pattern

CHAPTER FIVE: CQRS EXPLAINED

What is CQRS?

CQRS is an architectural pattern, where the acronym stands for Command Query Responsibility Segregation. We can talk about CQRS when the data read operations are separated from the data write operations, and they happen on a different interface.

In most of the CQRS systems, read and write operations use different data models, sometimes even different data stores. This kind of segregation makes it easier to scale, read and write operations and to control security - but adds extra complexity to your system.

The level of segregation can vary in CQRS systems:

* single data stores and separated model for reading and updating data
* separated data stores and separated model for reading and updating data

In the simplest data store separation, we can use read-only replicas to achieve segregation.

WHY AND WHEN TO USE CQRS?

In a typical data management system, all CRUD (Create Read Update Delete) operations are executed on the same interface of the entities in a single data storage. Like creating, updating, querying and deleting table rows in an SQL database via the same model.

CQRS really shines compared to the traditional approach (using a single model) when you build complex data models to validate and fulfil your business logic when data manipulation happens. Read operations compared to update and write operations can be very different or much simpler - like accessing a subset of your data only.

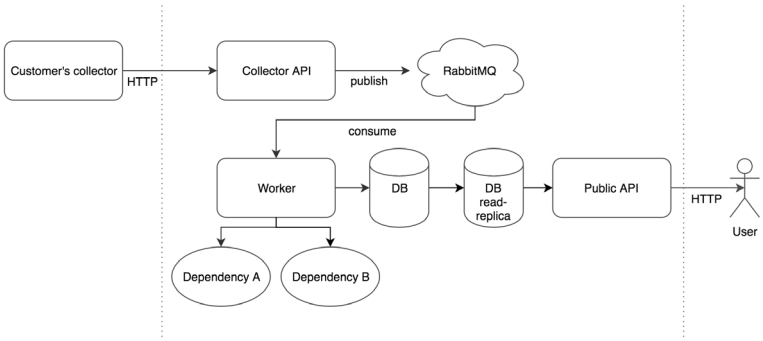
REAL WORLD EXAMPLE

In our Node.js Monitoring Tool, we use CQRS to segregate saving and representing the data. For example, when you see a distributed tracing visualization on our UI, the data behind it arrived in smaller

chunks from our customers application agents to our public collector API.

In the collector API, we only do a thin validation and send the data to a messaging queue for processing. On the other end of the queue, workers are consuming messages and resolving all the necessary dependencies via other services. These workers are also saving the transformed data to the database.

If any issue happens, we send back the message with exponential backoff and max limit to our messaging queue. Compared to this complex data writing flow, on the representation side of the flow, we only query a read-replica database and visualize the result to our customers.



Trace by RisingStack data processing with CQRS

CQRS AND EVENT SOURCING

I've seen many times that people are confusing these two concepts. Both of them are heavily used in event driven infrastructures like in an event driven microservices, but they mean very different things.

REPORTING DATABASE - DENORMALIZER

In some event driven systems, CQRS is implemented in a way that the system contains one or multiple Reporting databases.

A Reporting database is an entirely different read-only storage that models and persists the data in the best format for representing it. It's okay to store it in a denormalized format to optimize it for the client needs. In some cases, the reporting database contains only derived data, even from multiple data sources.

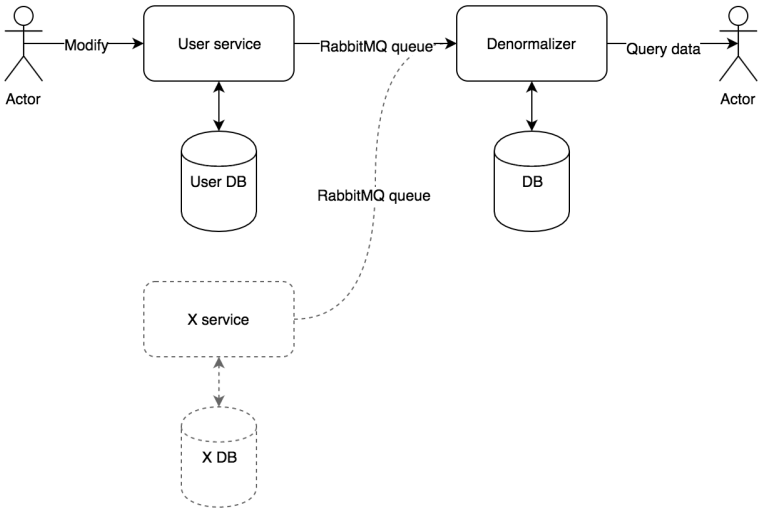
In a microservices architecture, we call a service the Denormalizer if it listens for some events and maintains a Reporting Database based on these. The client is reading the denormalized service's reporting database.

An example can be that the user profile service emits a user.edit event with `{ id: 1, name: 'John Doe', state: 'churn' }` payload, the Denormalizer service listens to it but only stores the `{ name: 'John Doe' }` in its Reporting Database, because the client is not interested in the internal state `churn` of the user.

It can be hard to keep a Reporting Database in sync. Usually, we can only aim to eventual consistency.

A CQRS NODE.JS EXAMPLE REPO

For our CQRS with Denormalizer Node.js example visit our cqrs-example GitHub repository.



OUTRO

CQRS is a powerful architectural pattern to segregate read and write operations and their interfaces, but it also adds extra complexity to your system.

In most of the cases, you shouldn't use CQRS for the whole system, only for specific parts where the complexity and scalability make it necessary.

To read more about CQRS and Reporting databases, I recommend to check out these resources:

* CQRS - Martin Fowler
* CQRS - MSDN
* CQRS, Task Based UIs, Event Sourcing agh! - Greg Young
* CQRS and Event Sourcing - Code on the Beach 2014 - Greg Young
* ReportingDatabase - Martin Fowler

+1: THE IMPORTANCE OF NODE.JS MONITORING

Getting insights into production systems is critical when you are building Node.js applications! You have an obligation to constantly detect bottlenecks and figure out what slows your product down.

An even greater issue is to handle and preempt downtimes. You must be notified as soon as they happen, preferably before your customers start to complain. **Based on these needs, proper monitoring should give you at least the following features and insights into your application's behavior:**

Profiling on a code level
Monitoring network connections
Performance dashboard
Real-time alerting

We at RisingStack build a solution which excels with all of these functionalities, and even more.

If you'd like to give it a try, you can now do it for free - just head over to our landing page at: trace.risingstack.com