From Callbacks to Observer

Node Patterns

Azat Mardan

Node Patterns: From Callbacks to Observer

Azat Mardan

Table of Contents

This report has a video course 🎥 on Node.University and slides 📄 at https://github.com/azat-co/node-patterns. You can clone the source code **with the slides** like this:

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

Before we can get started with Node patterns, let's touch on some of the main advantages and features of using Node. They'll help us later to understand why we need to deal with certain problems.

Node Advantages and Features

Here are some of the main reasons people use Node:

- JavaScript: Node runs on JavaScript so you can re-use your browser code, libraries and files.
- Asynchronous + Event Driven: Node executes tasks concurrently with the use of asynchronous code and patterns, thanks to event loop.
- Non-Blocking I/O: Node is extremely fast due to its non-blocking input/output architecture and Google Chrome V8 engine.

That's all neat but async code is hard. Human brains just didn't evolve to process things in an asynchronous manner where event loop schedules different pieces of logic in the future. Their order often is not the same order in which they were implemented.

To make the issue worse, most traditional languages, Computer Science programs and dev bootcamps focus on synchronous programming. This makes teaching asynchronous harder, because you really need to wrap your head around and start thinking in async.

JavaScript is the advantage and a disadvantage at the same time. For a long time, JavaScript was considered a toy language. 😒 It prevented some software engineering from taking time to learn it. Instead, they would assume taht they can just copy some code from Stackoverflow, cross their fingers and how it works. JavaScript is the only programming language which developers think they don't need to learn. Wrong!

JavaScript has its bad parts, that's why it's even more important to know the patterns. And please, take time to learn the fundamentals.

Then as you know the code complexity grows exponentially. Each module A used by module B is also used by module C which uses module B and so far so on. If you have an issue with A, then it affects a lot of other modules.

So the good code organization is important. That's why we, Node engineers, need to care about its patterns.

All You Can Eat Callbacks

How to schedule something in the future? In other words, how to ensure that after a certain event, our code will be executed, i.e., ensure the right sequence. Callbacks All the Way!

Callbacks are just functions and functions are first-class citizens which means you can treat them as variables (strings, numbers). You can toss them around to other functions. When we pass a function `t` as an argument and call it later, it's called a callback:

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

`t` is a callback. And there's a certain callback convention. Take a look at this snippet which reads the data from a file:

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

The following are Node 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 the order matters. Node.js won't enforce the arguments. Convention is not a guarantee—it's just a style. Read documentation or source code.

Named Functions

Now a new problem arises: How to ensure the right sequence? Control flow 😕
For example, there are three HTTP requests to perform the following tasks:

1. Get an auth token
2. Fetch data using auth token
3. PUT an update using data fetched in step 2

They must be executed in a certain order as shown in the following pseudo code:

```
... // 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)
        })
    })
})
```

Therefore, welcome to callback hell. This snippet was taken from callbackhell.com (yes, it exists—a place where bad code goes to die):

```
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))
        }
      })
    })
  }
}
```

Callback hell is also known as nested approach and pyramid of doom. It's only good to ensure a high job security for a developer because no one else will understand his/her code (joke, don't do it). The distinct characteristics of callback hell are:

- Hard to read
- Hard to modify/maintain/enhance
- Easy for developers to make bugs
- Closing parenthesis - 😈

Some of the solutions include:

- Abstract into named functions (hoisted or variables)
- Use obververs
- Use advanced libraries and techniques

We start with the named functions approach. The code of three nested requests can be refactored into three functions:

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

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

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

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

Modularization in Node

Moreover, you can modularize functions into separate files to keep your files lean and clean. Also, modularization will allow you to re-use the code in other projects. The main entry point will contain just two statements:

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

This is the `response.js` module with the first callback:

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

Similarly in `response2.js`, we import the `response3.js` and export with the second callback:

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

The final callback:

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

Node.js Middleware Pattern

Let's take callbacks to extreme. We can implement continuity-passing pattern known simply as the middleware pattern.

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
}
```

Middleware is often used in Express where request is coming from a client and response is sent back to the client. Request travels through a series of middleware:

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

The `next()` argument is simply a callback which tells Node and Express.js to proceed to the next step:

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

Node Modules Patterns

As we started talking about modularization, there are many ways to skin a catfish. The new problem is how to modularize code properly?

The main module patterns are:

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

Note: `exports = {...}` is anti-pattern because it won't export anything. You are just creating a variable, not assigning `module.exports`.

The second and third approaches are identical except that you need to type fewer characters when you use `exports.obj = {...}`.

The difference between first and second/third is your intent. When exporting a single monolithic object/class with components that interact with each other (e.g., methods, properties), then use `module.exports = {...}`.

On the other hand, when dealing with things which don't interact with each other but maybe categorically the same, you can put them in the same file but use `exports.obj = {...}` or `module.exports = {...}`.

Exporting objects and static things is clear now. But how to modularize dynamic code or where to initialize?

The solution is to export a function which will act as an initializer/constructor:

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

The same sidenote about `module.exports.name` and `exports.name` being identical apply to functions as well. The functional approach is more flexible because you can return an object but you can also execute some code before returning it.

This approach is sometimes called substack approach, because its' favorited by the prolific Node contributor substack.

If you remember that functions are objects in JavaScript (from reading on the JavaScript fundamentals maybe), then you know that we can create properties on functions. Therefore, it's possible to combine two patterns:

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

This is rarely used though as it's considered a Node Kung Fu. The best approach is to have one export per file. This will keep files lean and small.

Code in Node Modules

What about the code outside of the exports? You can have that too, but it works differently from the code inside of the exports. It has something to do with the way Node imports modules and caches them. For example, we have code A outside of exports and code B inside of it:

```
//import-module.js
console.log('Code A')
module.exports = function(options){
  console.log('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`, because the modules are cached by their resolved filename (you can trick Node by changing case and paths!).

Finally, you need to invoke the object to run code B, because we exported a function definition. It needs to be invoked. Knowing this, the script below will print only “Code A”. It will do it just once.

```
var f = require('./import-module.js')

require('./import-module.js')
```

The caching of modules works across different files, so requiring the same module many times in different files will trigger “Code A” just once.

Singleton Pattern in Node

Software engineers familiar with singleton pattern know that their purpose is to provide a single usually global instance. Set aside the conversations that singletons are bad, how do you implement them in Node?

We can leverage the caching feature of modules, i.e., `require` caches the modules. For example, we have a variable b which we export with value 2:

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

Then, in the script file (which imports the module), increment the value of b and import module `main`:

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

The module `main` imports `module` again, but this time the value of b is not 2 but 3!

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

A new problem on hand: modules are cached on based on their resolved filename. For this reason, filename will break the caching:

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

Or different paths will break the caching. The solution is to use `global`

```
global.name = ...
GLOBAL.name = ...
```

Consider this example which changes our beloved `console.log` from the default white to alarming red:

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

You need to require this module once and all your logs will become red. You don't even need to invoke anything because we don't export anything.

Using global is powerful... but anti-pattern, because it's very easy to mess up and overwrite something other modules use. Therefore, you should know about it because you might use a library which relies on this pattern (e.g., should behavior-driven development), but use it sparingly, only when needed.

It's very similar to browser `window.jQuery = jQuery` pattern. However, in browsers we don't have modules, it's better to use explicit exports in Node, than to use globals.

Importing Folders

Continuing with importing, there's an interesting feature in Node which allows you to import not just JavaScript/Node files, or JSON files, but whole folders.

Importing a folder is an abstraction pattern which is often used to organize code into packages or plugins (or modules—synonymous here). To importing a folder, create `index.js` in that folder with an `module.exports` assignment:

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

Then, in the main file you can import the folder by the name:

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

All the properties in `index.js` such as users, accounts, etc. will be properties of `routes` in `main.js`. The folder import pattern is used by almost all npm modules. There are libraries to automagically export ALL files in a given folder:

- `require-dir`
- `require-directory`
- `require-all`

Function Factory Pattern

There are no classes in Node. So how to organize your modular code into classes? Objects inherit from other objects, and functions are objects too.

Note: Yes, there are classes in ES6, but they don't support properties. Time will show if they are a good replacement to pseudo-classical inheritance. Node developers prefer function factory pattern for its simplicity to a clunky pseudo-classical one.

The solution is to create a function factory a.k.a. functional inheritance pattern. In it, the function is an expression which take options, initializes and returns the object. Each invocation of the expression will create a new instance. The instances will have the same properties.

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

Unlike pseudo-classical, the methods will not be from the prototype. Each new object will have its own copy of methods, so you don't need to worry about having a change in the prototype affecting all your instances.

Sometimes, you just have to use pseudo-classical (e.g., for Event Emitters), then there's `inherits`. Use it like this:

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

Node Dependency Injection

Every now and then, you have some dynamic objects which you need in modules. In other words, there are dependencies in the modules on something which is in the main file.

For example, in using a port number to start a server, consider an Express.js entry file `server.js`. It has a module `boot.js` which needs the configurations of the `app` object. It's straight forward to implement `boot.js` as a function export and pass `app`:

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

Function Which Returns a Function

The `boot.js` file actually uses another (probably my most favorite) pattern which I simply call function which returns a function. This simple pattern allows you to create different modes/versions of the inner function, so to speak.

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

One time I read a blog post where this pattern was called monad, but then one angry fan of functional programming told me that this is not a monad (and was angry about it as well). Oh well.

Observer Pattern in Node

Still, callbacks are hard to manage even with modules! For example, you have this:

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 we pass a callback to the module:

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

What about multiple callbacks? Not very development scalable 😢

The solution is quite elegant and is actually used a lot especially in core Node modules. Meet observer pattern with event emitters!

This is our module which emits the event `done` when everything is finished:

```
// 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
```

In the main script, we can customize *what to do* when the job is done.

```
// 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()
```

It's like a callback, only better, because you can have multiple events and you can remove, or execute them once.

```
emitter.listeners(eventName)
emitter.on(eventName, listener)
emitter.once(eventName, listener)
emitter.removeListener(eventName, listener)
```

30-Second Summary

1. Callbacks
2. Observer
3. Singleton
4. Plugins
5. Middleware
6. Bunch of other stuff 💥

Further Study

Obviously, there are more patterns like streams. Managing asynchronous code is a whole new set of problems, solutions and patterns. However, this essay is long enough already. Thanks for reading!

Start with these cornerstone Node patterns, use them where needed. To master Node, look at your favorite modules; how do they implement certain things?

These are things that are worth looking at for further study:

- `async` and `neo-async`: Great libraries for managing async code
- Promises: Come with ES6
- Generators: Promising
- Async await: Nice wrapper for promises coming soon-ish
- `hooks`: Hooks pattern module
- Node Design Patterns book not mine, I'm just reading it at the moment.
