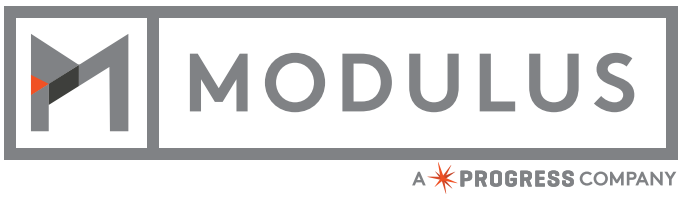


LEVEL UP WITH NODE.JS

OCTOBER 2014

LEVEL UP WITH NODE.JS

Getting started with Node.js is much like earning one star on Angry Birds. It feels nice and it's fun, but you know you can do better. And just like repeating a level in Angry Birds, the best way for a Node developer to level up is to build more apps. After your 2nd, 3rd, and fourth app, you'll start recognizing the same problem patterns over and over. From these problem patterns arise patterns for solutions.

In this guide, I hope to give you a head start on best practices developed over the years. Note that many of these topics are not really specific to Node, and can be applied to web software development in general.

MASTERING ASYNCHRONICITY

Okay, so this one is specific to Node. JavaScript code is executed in an event loop, on a single thread. The event loop queues up actions that won't run until the loop is available some time after the code that queued the action has finished executing. So code is said to execute asynchronously when it is queued to run sometime after the event loop is available.

Put another way, you can't exactly rely on your JavaScript running in the order that you wrote it. Without taking measures, you will try using the result of an operation before the result is ready. Furthermore, sometimes the result will be ready, causing a frustrating, intermittent bug.

There are a few different ways to handle asynchronous code, but we'll focus on the two most popular: callbacks and promises.

CALLBACKS

If you've ever written something like this (in jQuery) –

```
$("#button").on("click", function (e) {
  alert("Clicked!");
});
```

Congratulations! You've been using callbacks. The 2nd argument to the on function is another function that gets called once the original operation completes (in this case, the button click). Callbacks usually get passed through the result of the operation.

Callbacks are nice and simple. You pass a function that gets called at some point in the future. But there are land mines ahead. First, it's easy to forget where you are. Consider this example where we look up Mark Zuckerberg's Facebook photo:

```
function getZucksPhoto() {
  var url;

  HTTP.get("http://graph.facebook.com/zuck?fields=picture",     function(result) {
    url = result.picture.data.url;
  });

  return url;//Will probably be null
};
```

This seems okay at first glance. We fetch the result, and then return it. But because we are using the result of the request outside of the callback, we are almost guaranteed to run into a time when we return a null value. To fix the situation, we need to take a callback and pass our result to it:

```
function getZucksPhoto(cb) {
  HTTP.get("http://graph.facebook.com/zuck?fields=picture", function(result) {
    var url = result.picture.data.url;
    cb(url);
  });
};
```

A second common land mine is getting into callback hell. This occurs when you have a series of asynchronous operations that each depend on the result of the operation before it. For example, let's modify our example and download the Zuckerberg photo and save its URL to our database:

```
function saveZucksPhoto(cb) {
 //Get the photo URL
  HTTP.get("http://graph.facebook.com/zuck?fields=picture", function(result) {
    var url = result.picture.data.url;

   //Download the photo to the filesystem
    request(url).pipe(fs.createWriteStream("zuck.jpg")).on('close', function() {

     //Save the photo info to the database
      db.query("INSERT INTO photos SET ?", { name: "zuck", url: url }, function() {
        cb(url);
    });
  });
};
```

This callback nest is only three levels deep and is already hard to look at. Imagine if you had more operations. The deeper the nest, the harder it is to read, maintain, and debug. It's time to level up to JavaScript promises.

PROMISES

Before getting into the nuts and bolts, let's rewrite our photo crawler to use promises (using the promise module).

```
function saveZucksPhoto(cb) {
  new Promise(function(resolve) {
    HTTP.get("http://graph.facebook.com/zuck?fields=picture", function(result) {
      resolve(result.picture.data.url);
    });
  })
  .then(function(url) {
    return new Promise(function(resolve) {
      request(url).pipe(fs.createWriteStream("zuck.jpg")).on("close", function() {
        resolve(url);
      });
    });
  })
  .then(function(url) {
    return new Promise(function(resolve) {
      db.query("INSERT into photos SET ?", { name: "zuck", url: url }, function() {
        resolve(url);
      });
    });
  })
  .done(function(url) {
    cb(url);
  });
};
```

This is a lot more readable and maintainable, not to mention prettier, than the callback version. (There are extra lines in this example in order to wrap our functions in promises. Some libraries make this easier than others.)

So what's going on? Instead of returning results, our methods must return promises. A promise isn't a result, but an eventual result (or a promise of a result). By definition, a promise is always in one of three states:

- Pending - The initial state of a promise
- Resolved - The state of a promise representing a successful operation
- Rejected - The state of a promise representing a failed operation

Once a promise is resolved or rejected, it can never change again.

The promise/then framework keeps track of the callbacks until the promises start completing. As promises are fulfilled, the callbacks are run in order, and given the results of the previous operation. Promises provide a powerful way to handle asynchronous code. Promises enable developers to think and express asynchronous code in a more synchronous way.

DEPENDENCY INJECTION

Put simply, dependency injection is a pattern whereby dependencies of a class or object are explicitly injected into the class instead of the class creating or referencing them. Dependencies are usually injected by passing them into a class's constructor. So here's a simple example of a Notification class that sends messages out over SMS:

```
var Sms = require('sms');

function Notification() {
  this.transport = new Sms();

  this.send = function(msg) {
    this.transport.send(msg);
  };
};

// Use it like this:

var notification = new Notification();
notification.send('New friend request!');
```

But what if you wanted to send it over email instead of SMS? Let's modify this to inject the transport dependency via the constructor:

```
function Notification(transport) {
  this.transport = transport;

  this.send = function(msg) {
    this.transport.send(msg);
  };
};

// Use it like this:

var Email = require('email');
var emailer = new Email();

var notification = new Notification(emailer);
notification.send('New friend request!');
```

Now the caller can specify a different transport dependency depending on the context of the situation, with no change to the Notification class. Separating the instantiation of objects like this provides several benefits:

1. Code is more maintainable and re-usable
2. Class interfaces are elegant and clearer
3. Code is more testable!

Dependency injection (DI) simplifies testing classes that depend on resources like a 3rd-party API. With it, you can mock out the API and pass in the mock, instead of relying on the actual service.

DI is a high-level pattern that is platform agnostic and does not require a framework. However, without a framework, you often end up with areas of bootstrapping code where you instantiate all your objects and their dependencies and then wire them together. Dependency injection frameworks save you from this boilerplate. It makes wiring the application declarative rather than imperative.

Here's what a DI framework-driven notification send might look like (using the di module):

```
var di = require('di');
var Email = require('email');

function Notification(transport) {
  this.transport = transport;

  this.send = function(msg) {
    this.transport.send(msg);
  };
};

di.annotate(Notification, new di.Inject(Email));

// Use it like this:

var notification = new di.Injector().get(Notification);
notification.send('New friend request');
```

It's a small example, but imagine if you had many classes that depended on each other, and depended on data adaptors that depended on database connections and network APIs. DI helps keep instantiations and testing simple.

TESTING

Repeatable and automated testing is crucial to avoid regressions, prove quality, and most importantly, avoid developer stress!

There's no shortage of philosophies, techniques and tools for JavaScript testing. Let's look at a simple example. Imagine we have a blog Post class:

```
function Post(db) {
  this.title;
  this.body;
  this.slug;
  this.db;

  this.save = function() {
    var title = this.title.toLowerCase();
    this.slug = title.replace(/[^\w ]+/g, '').replace(new RegExp(' +', 'g'), '-');
    this.db.save(this);
  };
};
```

This Post class is pretty simple. You can set a title, a body, and then call save(). When you call save(), you want to run some transformation logic and generate a post slug based on the title.

Let's write a test to check that the slug is generated correctly (using mocha):

```
var assert = require('assert');
var db = {
  save: function() {}
};

describe('Post', function() {
  describe('save()', function() {
    it('should generate a slug based on the title', function() {
      var post = new Post(db);
      post.title = 'Level Up With Node.js';
      post.body = "Let's look at some best practices of Node.";
      post.save();
      assert.equal(post.slug, 'level-up-with-nodejs');
    });
  });
});
```

(The astute reader will note that since we inject the database dependency, we don't have to worry about setting up a real database in our test. We can pass in any fake database object that we want!)

Running the test:

```
$ mocha test.js

 Post
   .save()
    ✓ should generate a slug based on the title
  1 passing (6ms)
$
```

You now have a single-test test suite that will forever verify that slugs are generated correctly. If you encounter any issues (for example, slugify-ing title with non-ascii characters), you can write a test for that.

Now, as you continue to build out your Post class, you can run the test to ensure that your slug feature did not regress. You can make changes with stronger confidence and less stress, which is good for everyone.

MANAGING DEPENDENCIES

npm is the Node package manager. But npm is more than a package manager. Simple package managers make it easy to download and install packages to your project. npm is a dependency manager. A dependency manager allows you to:

1. Declare the modules that your project depends on.
2. Find the modules that those modules depend on, and any modules that those modules depend on, and so on.

3. Download and install all these modules in such a way that everything has what it needs and nothing conflicts (or be informed if modules can't be installed).
4. Not worry about any of this.

npm offers Node developers over 95,000 community-developed Node packages or modules, ready to go. Need to parse a csv file? Don't write your own. Just run `npm install csv`.

Dependency management with 3rd-party modules is related to dependency injection in your own app. Both are key to successful projects because you want to focus on your requirements and not worry about module load order or module versions or module incompatibilities. Embrace dependency management.

CODING WITH STYLE

Tabs vs spaces? Braces or no braces with single-line blocks? One var statement for each variable or all at once? JavaScript or (gasp) Coffeescript?

Coding conventions may not be the most important factor in code quality, but don't underestimate the mental cycles saved when you don't have to think about such things when scanning code. Consistent code can save time when maintaining code that others have written, or that you wrote long ago.

While this:

```
var x;//I personally don't like this
var y;
var z;
```

and this:

```
var x,//Are you a commas-after...
   y,
   z;
```

and this:

```
var x// ...Or a commas-before type of person?
  , y
  , z;
```

are all equivalent and common, imagine seeing it written differently in every JavaScript block of scope in a program. And while this is legal:

```
if (x)
  alert('x is defined'); //This should be obvious
```

it might accidentally lead to this mistake:

```
if (x)
  alert('x is defined');//But this is not as obvious
  alert('this will run regardless of x');// (This isn't Coffeescript)
```

So it's better to just avoid the confusion and always use braces, or always put one-line blocks on one line:

```
if (x) alert('x is defined');//This is good

if (x) {
  alert('x is defined');//I like this too
}

/* Hey, how should we write comments? */
```

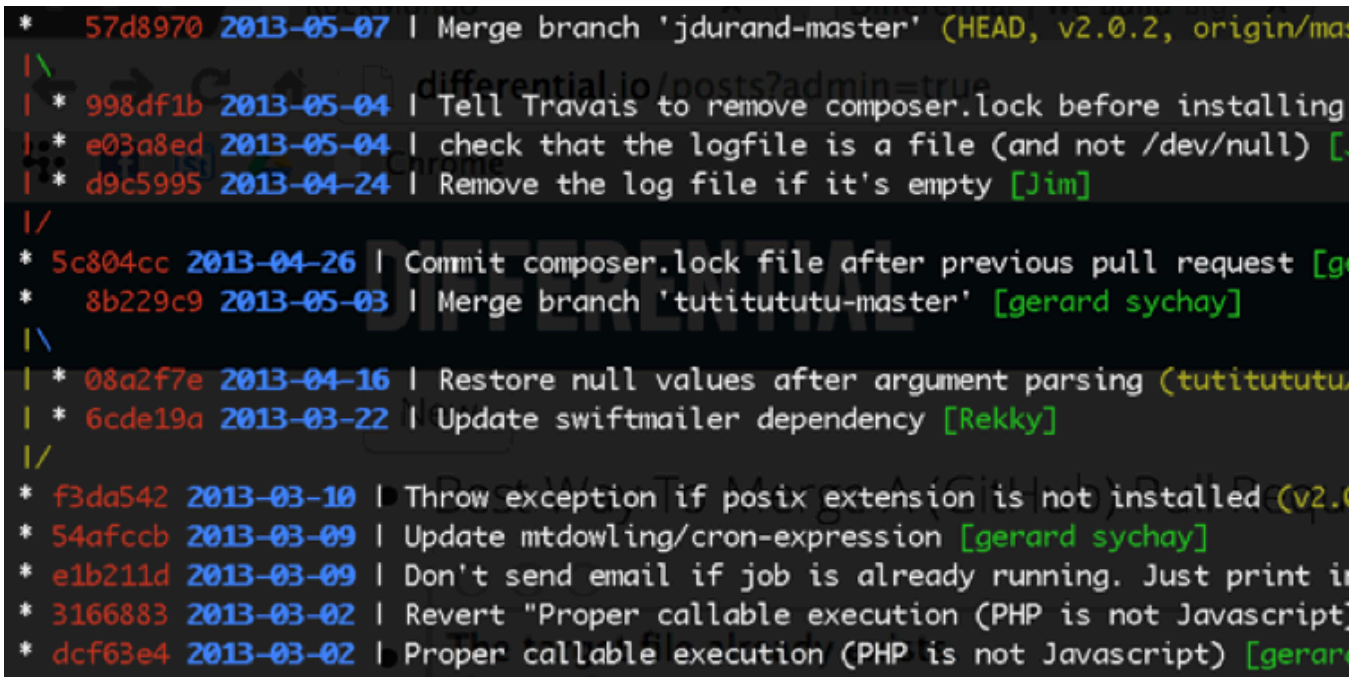
You can find many style guides online, but Airbnb has a nice one.

MANAGING SOFTWARE CONFIGURATION

The funny thing about software development is that you can work on a single software application forever if you wanted to (and you could afford to). There are always changes. There are always bugs, new features, and enhancements. Software changes a great deal over time. Part of being an advanced and professional software developer means controlling those changes and version. That's where source control comes in.

Source control allows you to:

- Keep a record of what code was changed, when changes were made, and who made them.
- Revert back to a previous state of the system if new bugs are introduced.
- Maintain separate and parallel "states" of the project for different purposes (e.g., a release branch that is running and a development branch that contains new unstable features).
- Identify a single code change, delete it, or copy it to another branch of the system.
- Establish a workflow for identifying an issue, implementing/fixing the issue, reviewing and approving the change, and finally committing the change to the project, all while tying the changes and comments to a single ID in your issue tracker.
- Do even more!



Software configuration is not a Node-specific concept, and you should adhere to good software configuration control in any software project you work on. There are many good source control tools out there, but you can't go wrong with git.

CONCLUSION

As with most tasks, Node development experience breeds expertise. But developing bad habits early on can slow app development, and lessen app quality. These tips on best practices for mastering asynchronicity, dependencies, testing, and other Node aspects should help ensure that you gain the right experience to more quickly level up your Node development skills.