**The If Works**

by James Coglan

About




# Callbacks, promises and simplicity

Since publishing Callbacks are imperative, promises are functional a couple of days ago I've received a huge volume of feedback. It's by far the most widely-read thing I've written and there have been far too many comments dotted about the web to cover them individually. But I'd like to highlight two articles written in response, both entitled 'Broken promises'.

The first, by Drew Crawford, is an excellent and detailed technical analysis of the limitations of promises, in particular their applicability to resource-constrained situations. It's derived from his experience trying to apply them to iOS programming, and failing. I encourage you to read the whole thing, but two important themes pop out for me.

First, the cost of modelling a solution using promises is increased memory usage. Callback-based programs (in JavaScript) store their state on the stack and in closures, since most data they use come from function arguments and local variables. The data they use tends to be quite fragmented; you don't typically build large object graphs while using callbacks, and this can help keep memory usage down. On the other hand, good promise-based programs use relationships between values, encoded in arrays and object references, to solve problems. Thus they tend to produce larger data structures, and this can be a problem if your solution won't fit in memory on your target runtime. If you hit a limit, you end up having to introduce manual control flow at which point callbacks become more appealing.

Second, promises let you delegate scheduling and timing to your tools, but sometimes that's not what you want. Sometimes there are hard limits on how long something can take, or you want to bail out early for some reason, or you want to manually control ordering of things. Again, in such situations, manual control flow is the right tool and promises loose their appeal.

I didn't mean to give the impression that promises are the one true solution to async programming (I try to avoid thinking of 'one true' *anything*, so Drew's casting of me as an 'evangelist' is a little bruising). There are certainly plenty of situations where they're not appropriate, and Drew catalogues a number of those very well. My only intention was to add a few more things to my mental toolbox so I can dig them out when I spot an appropriate occasion.

The second response is from Mikeal Rogers, whom I quote in my original article. (There's also a more recent version of his talk.) The bulk of this article concerns

the community effect of Node's design decision. Again, you should read his article rather than take my summary for granted.

Mikeal argues that the success of the Node ecosystem relies on how little the core platform provides. Instead of shipping with a huge standard library, Node has a bare minimum of building blocks you need to get started writing network apps. One of those building blocks is the conventions it sets up: the implicit contracts that mean all the modules in the ecosystem interoperate with each other. These include the `Stream` interface, the module system, and the `function(error, result) {}` callback style. Libraries that adhere to these interoperate with one another easily, and this is tremendously valuable.

I actually commend the Node team for their attitude on this stuff. I've been using Node, and Faye has been built on it, since v0.1. I was around before promises were taken out of core, and I can see that the volume of disagreement on them at the time meant they shouldn't have been standardized. And bizarrely, despite its early experimental status, I've found Node to be remarkably stable. I've been doing Ruby since 2006 and Node since early 2010, and I can honestly say Node has broken my stuff less while going from v0.1 to v0.10 than Ruby and its libraries have even during minor/patch releases. Paying attention to compatibility saves your users a huge amount of time and it's something I try to do myself.

I recognize that the Node team have their hands somewhat tied. Even if they wanted to go back to promises, doing so would break almost every module in npm, which would be a terrible waste of everyone's time. So in light of this, please take the following as a history lesson rather than a plea to change Node itself.

Mikeal's argument goes that minimizing core creates a market for new ideas in the library ecosystem, and we've seen that with various control flow libraries. This is certainly true, but I think there's an interesting difference where control flow is concerned, when compared to other types of problem. In most other popular web languages, the problem solved by the `function(error, result) {}` convention is solved at the language level: results are done with return values, errors with exceptions. There is no market for alternatives because this is usually a mechanism that cannot be meaningfully changed by users.

But I would also argue that the market for solutions to control flow in Node is *necessarily* constrained by what core does. Let's look at a different problem. Say I'm in the market for a hashtable implementation for JavaScript. I can pick any implementation I like so long as it does the job, because this problem is not constrained by the rest of the ecosystem. All that matters is that it faithfully implements a hashtable and performs reasonably well. If I don't like one, I can pick up another with a different API, and all I need to change is my code that uses the hashtable.

Most problems are like this: you just want a library that performs some task, and using it does not affect the rest of your system. But control flow is *not* like this: any control flow library out there must interoperate with APIs that use callbacks, which in Node means basically every library out there. So people trying to solve this problem do not have free reign over how to solve it, they are constrained by the ecosystem. No-one in the Ruby community thinks of return values and exceptions as being particularly constraining, but because in Node this concern is a library rather than a language feature, people have the freedom to try and change it. They just can't change it very much, because they must remain compatible with everything else out there.

And to me that's the history lesson: when you're designing a platform, you want to minimize the number of contracts people must agree to in order to interoperate. When people perceive they have the ability to change things, but they really can't, a tension arises because alternative solutions won't mesh well with the ecosystem. The more concepts require these contracts, the fewer ways users can experiment with alternative models.

Mikeal also goes into the relative ease of use of callbacks and my promise-based solutions. I disagree with most of this but that's not really important. We all consider different things to be easy, have different sets of knowledge, and are working on different problems with different requirements. That's fine, so long as you don't make technical decisions by simply pandering to the most ignorant. While we all need to write code that others can use and maintain, I hope part of that process involves trying to increase our collective knowledge rather than settling for what we know right now. Short of a usability study, any assertions I can make here about ease of use would be meaningless.

But I want to finish up on a word that's frequently interpreted to mean 'easy': 'simple'. I will assert that my promise-based solutions are simpler than using callbacks, but not in the ease-of-use sense. I mean in the sense that Rich Hickey uses in his talk Simple Made Easy, which is to say 'not complex', not having unrelated things woven together. This is an objective, or at least observable and demonstrable, property of a program, that we can examine by seeing how much you must change a program to change its results.

Let's revisit my two solutions from the previous article, one written with callbacks and one with promises. This program calls `fs.stat()` on a collection of paths, then uses the size of the first file for some task and uses the whole collection of stats for some unrelated task. Here are the two solutions:

```
var paths = ['file1.txt', 'file2.txt', 'file3.txt'],
    file1 = paths.shift();

async.parallel([
  function(callback) {
    fs.stat(file1, function(error, stat) {
```

```
      // use stat.size
      callback(error, stat);
    });
  },
  function(callback) {
    async.map(paths, fs.stat, callback);
  }
], function(error, results) {
  var stats = [results[0]].concat(results[1]);
  // use the stats
});
```

```
var fs_stat = promisify(fs.stat);

var paths = ['file1.txt', 'file2.txt', 'file3.txt'],
    statsPromises = list(paths.map(fs_stat));

statsPromises[0].then(function(stat) {
  // use stat.size
});

statsPromises.then(function(stats) {
  // use the stats
});
```

Now, I would say the first is uglier than the second, but this is neither objective nor particularly instructive. To see how much more complex the first solution is, we must observe what happens when we try to change the program. Let's say we no longer want to do the task with the size of the first file. Then the promise solution simply involves removing the code for that task:

```
var fs_stat = promisify(fs.stat);

var paths = ['file1.txt', 'file2.txt', 'file3.txt'],
    statsPromises = list(paths.map(fs_stat));

statsPromises.then(function(stats) {
  // use the stats
});
```

It would go similarly if we had a set of unrelated operations waiting to complete rather than the same operation on a set of inputs. We would just remove that operation from a list of promises and we'd be done. Often, changing promise-based solutions is the same as changing synchronous ones: you just remove a line, or change a variable reference or array index, or modify a data structure. These are changes to your program's data, not to its syntactic structure.

Now let's remove that task from the callback solution. We start by removing all the code that treats the first file as a special case:

```
var paths = ['file1.txt', 'file2.txt', 'file3.txt'];

async.parallel([
```

```
  function(callback) {
    async.map(paths, fs.stat, callback);
  }
], function(error, results) {
  var stats = results[0];
  // use the stats
});
```

We've removed the additional variable at the start, the special treatment of the first file, and the array-munging at the end. But there's no point having an `async.parallel()` with one item in it, so let's remove that too:

```
var paths = ['file1.txt', 'file2.txt', 'file3.txt'];

async.map(paths, fs.stat, function(error, stats) {
  // use the stats
});
```

So removing that one task was a tiny change to the promise solution, and a huge change to the callback solution: often, changing what you want a callback-based program to do involves changing its syntactic structure. The difference between the two approaches is that the promise solution keeps the notions of promises, collections and asynchrony and ordering separate from one another, whereas the callback-based solution conflates all these things. This is why the promise-based program requires much less change.

So while I think it's fruitless to argue about how *easy* a task is, I think you can demonstrate fairly objectively how *simple* it is. And while I admire what Node has achieved in many areas around interoperability through simplicity, this is one area where I wish it had gone the other way. Fortunately, JavaScript is such that we have ways of routing around designs we don't care for.

Thanks to Drew and Mikeal for taking the time to respond. I would welcome any further feedback about how to improve either of my above approaches.

—

If you've enjoyed this article, you might enjoy my recently published book JavaScript Testing Recipes. It's full of simple techniques for writing modular, maintainable JavaScript apps in the browser and on the server.

Posted on April 1, 2013. This entry was posted in JavaScript, Functional, Evented, Node. Bookmark the permalink.

← Callbacks are imperative, promises are functional: Node's biggest missed opportunity

→ I'm not helping

Theme: Publish by Konstantin Kovshenin.