Gorgi Kosev

code, music, math

Why I am switching to promises

Mon Oct 07 2013

I'm switching my node code from callbacks to promises. The reasons aren't merely aesthetical, they're rather practical:

Throw-catch vs throw-crash

We're all human. We make mistakes, and then JavaScript `throw`s an error. How do callbacks punish that mistake? They crash your process!

> *But spion, why don't you use domains?*

Yes, I could do that. I could crash my process gracefully instead of letting it just crash. But its still a crash no matter what lipstick you put on it. It still results with an inoperative worker. With thousands of requests, 0.5% hitting a throwing path means over 50 process shutdowns and most likely denial of service.

And guess what a user that hits an error does? Starts repeatedly refreshing the page, thats what. The horror!

Promises are throw-safe. If an error is thrown in one of the `.then` callbacks, only that single promise chain will die. I can also attach error or "finally" handlers to do any clean up if necessary - transparently! The process will happily continue to serve the rest of my users.

For more info see #5114 and #5149. To find out how promises can solve this, see bluebird #51

`if (err) return callback(err)`

That line is haunting me in my dreams now. What happened to the DRY principle?

I understand that its important to explicitly handle all errors. But I don't believe its important to explicitly *bubble them up* the callback chain. If I don't deal with the error here, thats because I can't deal with the error there - I simply don't have enough context.

> *But spion, why don't you wrap your calbacks?*

I guess I could do that and lose the callback stack when generating a `new Error()`. Or since I'm already wrapping things, why not wrap the entire thing with promises, rely on longStackSupport, and handle errors at my discretion?

Also, what happened to the DRY principle?

Promises are now part of ES6

Yes, they will become a part of the language. New DOM APIs will be using them too. jQuery already switched to promise...ish things. Angular utilizes promises everywhere (even in the templates). Ember uses promises. The list goes on.

Browser libraries already switched. I'm switching too.

Containing Zalgo

Your promise library prevents you from releasing Zalgo. You can't release Zalgo with promises. Its impossible for a promise to result with the release of the Zalgo-beast. Promises are Zalgo-safe (see section 3.1).

Callbacks getting called multiple times

Promises solve that too. Once the operation is complete and the promise is resolved (either with a result or with an error), it cannot be resolved again.

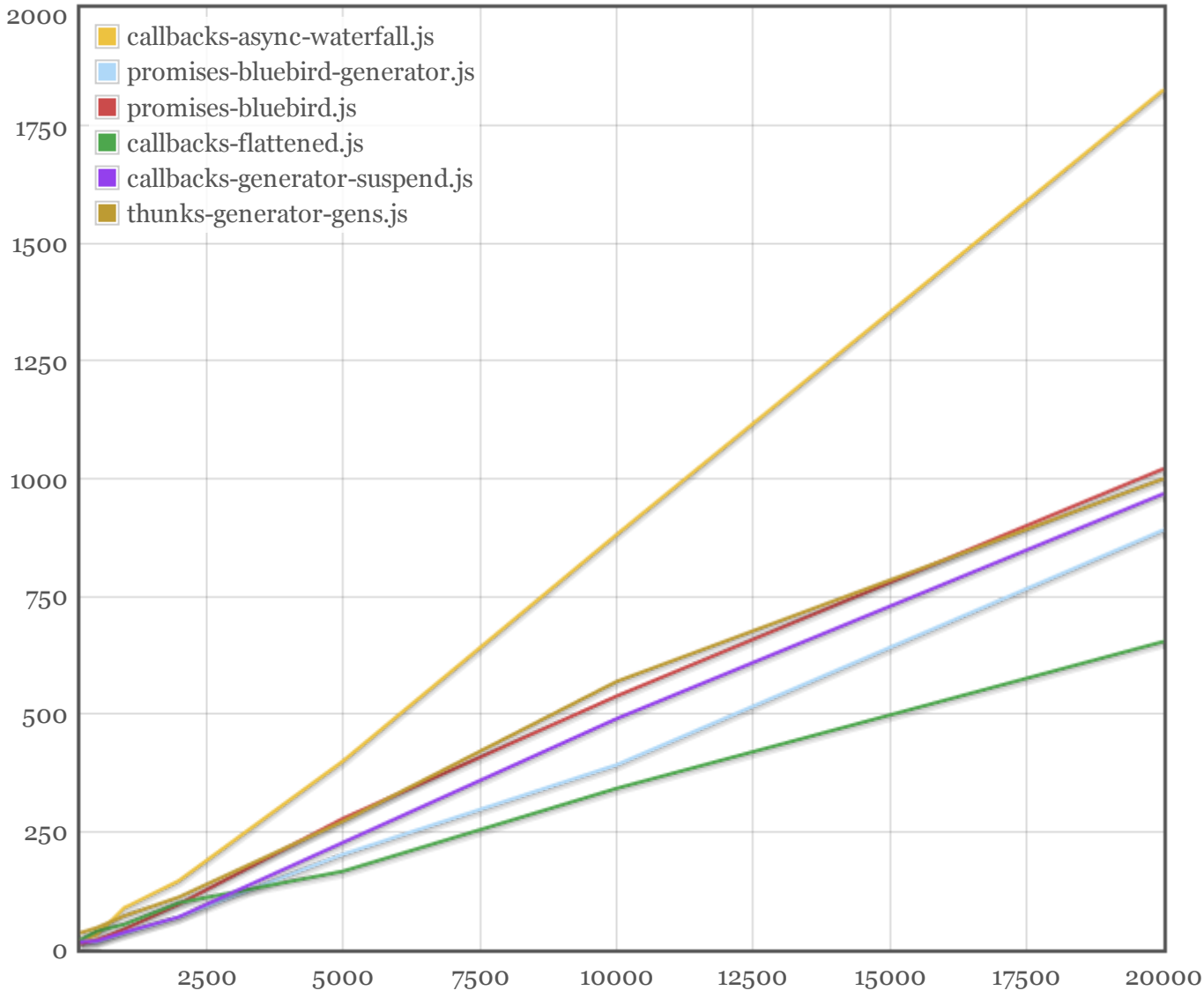
Promises can do your laundry

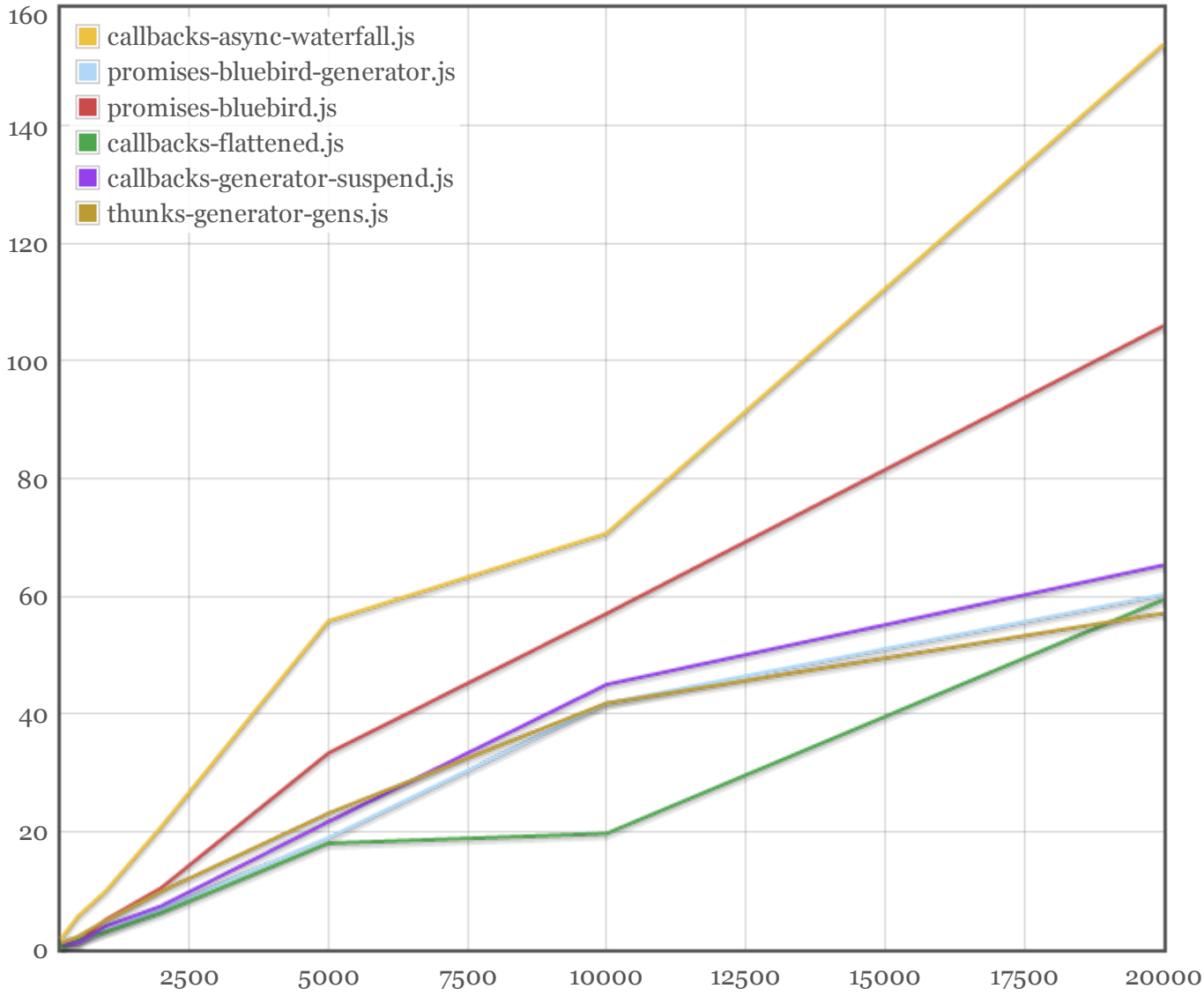
Oops, unfortunately, promises wont do that. You still need to do it manually.

But you said promises are slow!

Yes, I know I wrote that. But I was wrong. A month after I wrote the giant comparison of async patterns, Petka Antonov wrote Bluebird. Its a wicked fast promise library, and here are the charts to prove it:

Time to complete (ms)


callbacks-async-waterfall.js
promises-bluebird-generator.js
promises-bluebird.js
callbacks-flattened.js
callbacks-generator-suspend.js
thunks-generator-gens.js
2000
1750
1500
1250
1000
750
500
250
0
2500
5000
7500
10000
12500
15000
17500
20000
Parallel requests
Memory usage (MB)



And now, a table containing many patterns, 10 000 parallel requests, 1 ms per I/O op. Measure ALL the things!

file	time(ms)	memory(MB)
callbacks-original.js	316	34.97
callbacks-flattened.js	335	35.10
callbacks-catcher.js	355	30.20
promises-bluebird-generator.js	364	41.89
dst-streamline.js	441	46.91

file	time(ms)	memory(MB)
callbacks-deferred-queue.js	455	38.10
callbacks-generator-suspend.js	466	45.20
promises-bluebird.js	512	57.45
thunks-generator-gens.js	517	40.29
thunks-generator-co.js	707	47.95
promises-compose-bluebird.js	710	73.11
callbacks-generator-genny.js	801	67.67
callbacks-async-waterfall.js	989	89.97
promises-bluebird-spawn.js	1227	66.98
promises-kew.js	1578	105.14
dst-stratifiedjs-compiled.js	2341	148.24
rx.js	2369	266.59
promises-when.js	7950	240.11
promises-q-generator.js	21828	702.93
promises-q.js	28262	712.93
promises-compose-q.js	59413	778.05

Promises are not slow. At least, not anymore. Infact, bluebird generators are almost as fast as regular callback code (they're also the fastest generators as of now). And bluebird promises are definitely at least two times faster than `async.waterfall`.

Considering that bluebird wraps the underlying callback-based libraries **and** makes your own callbacks exception-safe, this is really amazing. `async.waterfall` doesn't do this. exceptions still crash your process.

What about stack traces?

Bluebird has them behind a flag that slows it down about 5 times. They're even longer than Q's `longStackSupport`: bluebird can give you the entire event chain. Simply enable the flag in development mode, and you're suddenly in debugging nirvana. It may even be viable to turn them on in production!

What about the community?

This is a valid point. Mikeal said it: If you write a library based on promises, nobody is going to use it.

However, both bluebird and Q give you `promise.nodeify`. With it, you can write a library with a dual API that can both take callbacks and return promises:

```
module.exports = function fetch(itemId, callback) {
    return locate(itemId).then(function(location) {
        return getFrom(location, itemId);
    }).nodeify(callback);
}
```

And now my library is not imposing promises on you. Infact, my library is even friendlier to the community: if I make a dumb mistake that causes an exception to be thrown in the library, the exception will be passed as an error to your callback instead of crashing your process. Now I don't have to fear the wrath of angry library users expecting zero downtime on their production servers. Thats always a plus, right?

What about generators?

To use generators with callbacks you have two options

1. use a resumer style library like suspend or genny
2. wrap callback-taking functions to become thunk returning functions.

Since #1 is proving to be unpopular, and #2 already involves wrapping, why not just `s/thunk/promise/g` in #2 and use generators with promises?

But promises are unnecessarily complicated!

Yes, the terminology used to explain promises can often be confusing. But promises themselves are pretty simple - they're basically like lightweight streams for single values.

Here is a straight-forward guide that uses known principles and analogies from node (remember, the focus is on simplicity, not correctness):

Edit (2014-01-07): I decided to re-do this tutorial into a series of short articles called promise nuggets. The content is CC0 so feel free to fork, modify, improve or send pull requests. The old tutorial will remain available within this article.

Promises are objects that have a `then` method. Unlike node functions, which take a single callback, the `then` method of a promise can take two callbacks: a success callback and an error callback. When one of these two callbacks returns a value or throws an exception, `then` must behave in a way that enables stream-like chaining and simplified error handling. Lets explain that behavior of `then` through examples:

Imagine that node's `fs` was wrapped to work in this manner. This is pretty easy to do - bluebird already lets you do something like that with `promisify()`. Then this code:

```
fs.readFile(file, function(err, res) {
    if (err) handleError();
    doStuffWith(res);
});
```

will look like this:

```
fs.readFile(file).then(function(res) {
    doStuffWith(res);
}, function(err) {
    handleError();
});
```

Whats going on here? `fs.readFile(file)` starts a file reading operation. That operation is not yet complete at the point when readFile returns. This means we can't return the file content. But we can still return something: we can return the reading operation itself. And that operation is represanted with a promise.

This is sort of like a single-value stream:

```
net.connect(port).on('data', function(res) {
    doStuffWith(res);
}).on('error', function(err) {
    hadnleError();
});
```

So far, this doesn't look that different from regular node callbacks - except that you use a second callback for the error (which isn't necessarily better). So when does it get better?

Its better because you can attach the callback later if you want. Remember, `fs.readFile(file)` *returns* a promise now, so you can put that in a var, or return it from a function:

```
var filePromise = fs.readFile(file);
// do more stuff... even nest inside another promise, then
filePromise.then(function(res) { ... });
```

Yup, the second callback is optional. We're going to see why later.

Okay, that's still not much of an improvement. How about this then? You can attach more than one callback to a promise if you like:

```
filePromise.then(function(res) { uploadData(url, res); });
filePromise.then(function(res) { saveLocal(url, res); });
```

Hey, this is beginning to look more and more like streams - they too can be piped to multiple destinations. But unlike streams, you can attach more callbacks and get the value even *after* the file reading operation completes.

Still not good enough?

What if I told you... that if you return something from inside a .then() callback, then you'll get a promise for that thing on the outside?

Say you want to get a line from a file. Well, you can get a promise for that line instead:

```
var filePromise = fs.readFile(file)

var linePromise = filePromise.then(function(data) {
    return data.toString().split('\n')[line];
});

var beginsWithHelloPromise = linePromise.then(function(line) {
    return /^hello/.test(line);
});
```

Thats pretty cool, although not terribly useful - we could just put both sync operations in the first `.then()` callback and be done with it.

But guess what happens when you return a *promise* from within a `.then` callback. You get a promise for a promise outside of `.then()`? Nope, you just get the same promise!

```
function readProcessAndSave(inPath, outPath) {
    // read the file
    var filePromise = fs.readFile(inPath);
    // then send it to the transform service
    var transformedPromise = filePromise.then(function(content) {
```

```
        return service.transform(content);
    });
    // then save the transformed content
    var writeFilePromise = transformedPromise.then(function(transformed) {
        return fs.writeFile(otherPath, transformed)
    });
    // return a promise that "succeeds" when the file is saved.
    return writeFilePromise;
}
readProcessAndSave(file, url, otherPath).then(function() {
    console.log("Success!");
}, function(err) {
    // This function will catch *ALL* errors from the above
    // operations including any exceptions thrown inside .then
    console.log("Oops, it failed.", err);
});
```

Now its easier to understand chaining: at the end of every function passed to a `.then()` call, simply return a promise.

Lets make our code even shorter:

```
function readProcessAndSave(file, url, otherPath) {
    return fs.readFile(file)
        .then(service.transform)
        .then(fs.writeFile.bind(fs, otherPath));
}
```

Mind = blown! Notice how I don't have to manually propagate errors. They will automatically get passed with the returned promise.

What if we want to read, process, then upload, then also save locally?

```
function readUploadAndSave(file, url, otherPath) {
    var content;
    // read the file and transform it
    return fs.readFile(file)
    .then(service.transform)
    .then(function(vContent)
        content = vContent;
        // then upload it
        return uploadData(url, content);
    }).then(function() { // after its uploaded
```

```
        // save it
        return fs.writeFile(otherPath, content);
    });
}
```

Or just nest it if you prefer the closure.

```
function readUploadAndSave(file, url, otherPath) {
    // read the file and transform it
    return fs.readFile(file)
        .then(service.transform)
        .then(function(content)
            return uploadData(url, content).then(function() {
                // after its uploaded, save it
                return fs.writeFile(otherPath, content);
            });
        });
}
```

But hey, you can also upload and save in parallel!

```
function readUploadAndSave(file, url, otherPath) {
    // read the file and transform it
    return fs.readFile(file)
        .then(service.transform)
        .then(function(content) {
            // create a promise that is done when both the upload
            // and file write are done:
            return Promise.join(
                uploadData(url, content),
                fs.writeFile(otherPath, content));
        });
}
```

No, these are not "conveniently chosen" functions. Promise code really is that short in practice!

Similarly to how in a `stream.pipe` chain the last stream is returned, in promise pipes the promise returned from the last `.then` callback is returned.

Thats all you need, really. The rest is just converting callback-taking functions to promise-returning functions and using the stuff above to do your control flow.

You can also return values in case of an error. So for example, to write a `readFileOrDefault` (which returns a default value if for example the file doesn't exist) you would simply return the default value from the error callback:

```
function readFileOrDefault(file, defaultContent) {
    return fs.readFile(file).then(function(fileContent) {
        return fileContent;
    }, function(err) {
        return defaultContent;
    });
}
```

You can also throw exceptions within both callbacks passed to `.then`. The user of the returned promise can catch those errors by adding the second .then handler

Now how about `configFromFileOrDefault` that reads and parses a JSON config file, falls back to a default config if the file doesn't exist, but reports JSON parsing errors? Here it is:

```
function configFromFileOrDefault(file, defaultConfig) {
    // if fs.readFile fails, a default config is returned.
    // if JSON.parse throws, this promise propagates that.
    return fs.readFile(file).then(JSON.parse,
           function ifReadFails() {
               return defaultConfig;
           });
    // if we want to catch JSON.parse errors, we need to chain another
    // .then here - this one only captures errors from fs.readFile(file)
}
```

Finally, you can make sure your resources are released in all cases, even when an error or exception happens:

```
var result = doSomethingAsync();

return result.then(function(value) {
```

```
        // clean up first, then return the value.
        return cleanUp().then(function() { return value; })
    }, function(err) {
        // clean up, then re-throw that error
        return cleanUp().then(function() { throw err; });
    })
```

Or you can do the same using `.finally` (from both Bluebird and Q):

```
var result = doSomethingAsync();
return result.finally(cleanUp);
```

The same promise is still returned, but only after `cleanUp` completes.

But what about async?

Since promises are actual values, most of the tools in async.js become unnecessary and you can just use whatever you're using for regular values, like your regular `array.map` / `array.reduce` functions, or just plain for loops. That, and a couple of promise array tools like `.all`, `.spread` and `.some`

You already have async.waterfall and async.auto with .then and .spread chaining:

```
files.getLastTwoVersions(filename)
    .then(function(items) {
        // fetch versions in parallel
        var v1 = versions.get(items.last),
            v2 = versions.get(items.previous);
        return [v1, v2];
    })
    .spread(function(v1, v2) {
        // both of these are now complete.
        return diffService.compare(v1.blob, v2.blob)
    })
    .then(function(diff) {
        // voila, diff is ready. Do something with it.
    });
```

`async.parallel` / `async.map` are straightforward:

```
// download all items, then get their names
var pNames = ids.map(function(id) {
    return getItem(id).then(function(result) {
        return result.name;
    });
});
// wait for things to complete:
Promise.all(pNames).then(function(names) {
    // we now have all the names.
});
```

What if you want to wait for the current item to download first (like `async.mapSeries` and `async.series`)? Thats also pretty straightforward: just wait for the current download to complete, then start the next download, then extract the item name, and thats exactly what you say in the code:

```
// start with current being an "empty" already-fulfilled promise
var current = Promise.fulfilled();
var namePromises = ids.map(function(id) {
    // wait for the current download to complete, then get the next
    // item, then extract its name.
    current = current
        .then(function() { return getItem(id); })
        .then(function(item) { return item.name; });
    return current;
});
Promise.all(namePromises).then(function(names) {
    // use all names here.
});
```

The only thing that remains is mapLimit - which is a bit harder to write - but still not that hard:

```
var queued = [], parallel = 3;
var namePromises = ids.map(function(id) {
    // How many items must download before fetching the next?
    // The queued, minus those running in parallel, plus one of
    // the parallel slots.
    var mustComplete = Math.max(0, queued.length - parallel + 1);
    // when enough items are complete, queue another request for an item
    return Promise.some(queued, mustComplete)
        .then(function() {
            var download = getItem(id);
```

```
            queued.push(download);
            return download;
        }).then(function(item) {
            // after that new download completes, get the item's name.
            return item.name;
        });
  });
Promise.all(namePromises).then(function(names) {
    // use all names here.
});
```

That covers most of async.

What about early returns?

Early returns are a pattern used throughout both sync and async code. Take this hypothetical sync example:

```
function getItem(key) {
    var item;
    // early-return if the item is in the cache.
    if (item = cache.get(key)) return item;
    // continue to get the item from the database. cache.put returns the item.
    item = cache.put(database.get(key));

    return item;
}
```

If we attempt to write this using promises, at first it looks impossible:

```
function getItem(key) {
    return cache.get(key).then(function(item) {
        // early-return if the item is in the cache.
        if (item) return item;
        return database.get(item)
    }).then(function(putOrItem) {
        // what do we do here to avoid the unnecessary cache.put ?
    })
}
```

How can we solve this?

We solve it by remembering that the callback variant looks like this:

```
function getItem(key, callback) {
    cache.get(key, function(err, res) {
        // early-return if the item is in the cache.
        if (res) return callback(null, res);
        // continue to get the item from the database
        database.get(key, function(err, res) {
            if (err) return callback(err);
            // cache.put calls back with the item
            cache.put(key, res, callback);
        })
    })
}
```

The promise version can do pretty much the same - just nest the rest of the chain inside the first callback.

```
function getItem(key) {
    return cache.get(key).then(function(res) {
        // early return if the item is in the cache
        if (res) return res;
        // continue the chain within the callback.
        return database.get(key)
            .then(cache.put);
    });
}
```

Or alternatively, if a cache miss results with an error:

```
function getItem(key) {
    return cache.get(key).catch(function(err) {
        return database.get(key).then(cache.put);
    });
}
```

That means that early returns are just as easy as with callbacks, and sometimes even easier (in case of errors)

What about streams?

Promises can work very well with streams. Imagine a `limit` stream that allows at most 3 promises resolving in parallel, backpressuring otherwise, processing items from leveldb:

```
originalSublevel.createReadStream().pipe(limit(3, function(data) {
    return convertor(data.value).then(function(converted) {
        return {key: data.key, value: converted};
    });
})).pipe(convertedSublevel.createWriteStream());
```

Or how about stream pipelines that are safe from errors without attaching error handlers to all of them?

```
pipeline(original, limiter, converted).then(function(done) {

}, function(streamError) {

})
```

Looks awesome. I definitely want to explore that.

The future?

In ES7, promises will become monadic (by getting flatMap and unit). Also, we're going to get generic syntax sugar for monads. Then, it trully wont matter what style you use - stream, promise or thunk - as long as it also implements the monad functions. That is, except for callback-passing style - it wont be able to join the party because it doesn't produce values.

I'm just kidding, of course. I don't know if thats going to happen. Either way, promises are useful and practical and will remain useful and practical in the future.

trevnorris — That I can understand. My original discussion/rant was from the context of devs wanting to replace the ...

Guest — In my experience, the error handling does not exit the thread as I am expecting. Must you reject the promise ...