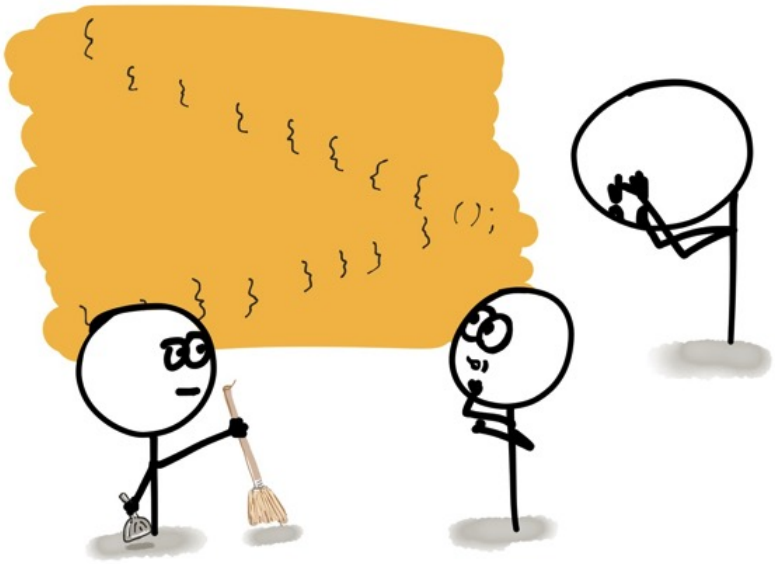
Ending the Nested Tree of Doom with Chained Promises

September 27, 2016 By derickbailey

Julia Jacobs recently asked a question in the WatchMeCode community slack, about some asynchronous code she wanted to clean up.

In this question, she wanted to know of good options for restructuring deeply nested promises without introducing any new libraries. Would it be possible to clean up the code and only use ES6 features?



It's a common pattern and problem – the Nested Tree of Doom – not only with promises, but with JavaScript callbacks in general.

The full question, is as follows

> *Hey all. Does anyone know any decent patterns for*
>
> *a) async ES6 classes other than shoving everything into a native promise in a getter.*
>
> *b) creating an async waterfall serialization flow with multiple functions using generators.*
>
> *I'll create some gists with samples of the code I'm trying to refactor. I'm trying to stay away from libraries and stick with native ES6 goodness.*

When I first read the question without looking at the example code, I wondered if promises were the right way to go.

Personally, I prefer the node style of callbacks as my go-to pattern for asynchronous code. Promises are definitely useful, but they are most useful in specific scenarios.

My initial response, however, was about ES6 generators.

> *the best way to take advantage of generators and async work is going to be with a small library [like co].*
>
> *you can write your own with only a few lines of code, though*
>
> *if you don't like getters that return promises, can use a method with a callback parameter.*
>
> *personally, i prefer callback methods over promises, until i have a specific use case for promises (like waiting for multiple things, only wanting to perform the action once but retrieve the value multiple times, chaining async ops, etc)*

Once Julia posted her code sample, my advice quickly changed.

While I don't reach for promises as my first choice, Julia's scenario and code samples quickly changed my mind.

Here is what she posted for the code in a hapi.js router:

> *It's the result of two weeks of banging out a hapijs arch which parses a huge XML response*

from a Java API and maps it to a huge json contract with very strict corporate requirements.

```
let xmltojsonhttpclient = new XMLtoJSONHttpC

xmltojsonhttpclient
  .get()
  .then((json) => {
    return SearchModel
    .get(json)
    .then((transform) => {
      return SearchController
      .get(transform)
      .then((mappedData) => {
        return ContractModel
        .get(mappedData)
        .then((contractData) => {
          return ContractController
          .get(contractData)
          .then((mappedContractData) => {
            return reply(mappedContractData)
          });
        });
      });
    });
  })
  .catch((err) => {
    console.error(util.inspect(err, false, 5
    reply(err);
  });
```

1.js hosted with ♥ by GitHub view raw

Frankly, this code is darn near as beautiful an example as you can find, when it comes to nested callbacks and promises.

And I don't mean "a beautiful mess" – I mean that Julia has written code that you dream of finding, when you are looking at restructuring.

Most of my own nested promises are garbage – closures; creating promises inside of callbacks; nested .catch and resolve statements and more of a mess than I care to show anyone.

This code is near spotlessly clean, already.

Unlike the mess that I tend to create with nested promises, the code Julia showed us is uniformly written.

It uses no closures.

It takes the results of the previous work and passes it directly to the next work, with nothing else.

And it does all of this through promises that are created by other functions.

The only thing this code really needs, is a small adjustment to remove the nested promises and turn them into chained promises.

As I noted in my response to Julia,

> *instead of doing `return SearchModel.get(json).then({ …` you can return the promise from `SearchModel.get` directly*
>
> *`return SearchModel.get(json);`*
>
> *Promises will take any return value from a `then` callback, and forward it to the next `then` for you*

if the return value is a Promise itself, it will wait for that promise to resolve or reject

so the above code should be functionally the same just with less nesting

When you take the first few lines of the above code and apply this idea, it becomes this:

```
xmltojsonhttpclient
    .get()
    .then((json) => {
        return SearchModel.get(json);
    })
```

2.js hosted with ♥ by GitHub view raw

And this is where the magic begins.

By returning the promise from SearchModel.get, the code nesting has been reduced one level.

Apply this same pattern throughout the rest of the sample, and you reduce the nesting by one level, at each level of nesting.

The result is that there are never any nested promises!

```
xmltojsonhttpclient
  .get()
  .then((json) => {
    return SearchModel.get(json);
  }).then((transform) => {
    return SearchController.get(transform)
  }).then((mappedData) => {
    return ContractModel.get(mappedData)
```

```
9   }).then((contractData) => {
10    return contractController.get(contractData)
11  }).then((mappedContractData) => {
12    return reply(mappedContractData);
13  })
14  .catch((err) => {
15    console.error(util.inspect(err, false, 5, true));
16    reply(err);
17  });
```

3.js hosted with ♥ by GitHub view raw

But the magic doesn't end here.

Reducing the nested promises is only the first steps in reorganizing this code.

I continued the example by talking about extracting named functions

> *if you really want to reduce this code further, extract each of those callbacks into a named function*
>
> *then you can do `xml.tojsonhttpclient.get().then(transformit).then(mapit).then(…)`*

This works because these functions are very clean. They take the result of the previous promise and pass it into the next promise. There's no closures or other code, as I mentioned before.

The result of extracting the callbacks into named functions looks like this:

```
1  // do the work, at a high level
2  xmltojsonhttpclient
3    .get()
```

```
4     .then(transformit)
5     .then(mapit)
6     .then(getcontractdata)
7     .then(mapcontractdata)
8     .then(sendreply);
9
10  // extract the named functions as details
11
12  function transformit (json) {
13    return SearchModel.get(json);
14  }
15
16  function mapit (transform) {
17    return SearchController.get(transform);
18  }
19
20  function getcontractdata(mappedData) {
21    return ContractModel.get(mappedData);
22  }
23
24  function mapcontractdata (contractData) {
25    return ContractController.get(contractData);
26  }
27
28  function sendreply(mappedContractData) => {
29    return reply(mappedContractData);
30  }
```

4.js hosted with ♥ by GitHub view raw

Now this is beautiful code!

But if we're talking ES6, Let's use ES6.

Shortly after posting this, there was some discussion on twitter, and Richard Livsey pointed out that ES6 arrow functions could be used with implicit returns, instead of extracting named functions.

Derick Bailey @derickbailey 27 Sep

ending the nested tree of doom, with chained promises derickbailey.com/2016/09/27/end…

Richard Livsey
@rlivsey

Follow

@derickbailey great post. Another intermediate step could be to use implicit returns vs extracting functions

6:06 PM - 27 Sep 2016 · City of London, London

With this post, he also provided an example that shows the use of arrow functions to achieve this:

```
1  xmltojsonhttpclient
2    .get()
3    .then(json => SearchModel.get(json))
4    .then(transform => SearchController.get(transform))
5    .then(mappedData => ContractModel.get(mappedData))
6    .then(contractData => ContractController.get(contractDa
7    .then(mappedContractData => reply(mappedContractData))
8    .catch(err => {
9      console.error(util.inspect(err, false, 5, true));
10     reply(err);
11   });
```

3.1.js hosted with ♥ by GitHub view raw

The code I have above may be slightly cleaner when it comes to reading the chained promises, but this is a great option if you're concerned about lines of code or having all those extra little functions laying around.

The goal may not be to reduce the number of lines of code, though.

In this scenario, the goal was to make the code easier to reason about and easier to change, not just reducing the number lines of code (though this may be an important aspect in other scenarios).

By taking what was already clean code and reorganizing it into chained promises instead of nested, the code became easier to follow.

When the functions were extracted and named, though, that's when the code became easier to understand at a higher workflow level.

You no longer have to dive into the details of each promise's callback to understand what happens next. Instead, you can look at the high level flow of chained promises and see a simplified name that represents what will happen next.

The code is easier to modify, as well. If you need to insert a new method, change a detail, remove a step, it's all right there in the high level chaining.

This type of restructuring is not always going to work, though.

When Julia brought this question to the WatchMeCode community slack, she was already in a good place.

Most of the time when I'm looking at nested promises, I'm in much worse shape.

If you face code that has closures around variables, re-using them between promise callbacks, for example, you're going to run into problems.

If you have nested promises being created within the promise callbacks directly, or worse, you have nested promise chains being resolved and rejected, it can be incredibly difficult to fix.

You may find yourself in a situation where a promise chain is simply the wrong way to solve the problem.

But if you're looking at code which is clean and concise like what Julia brought to us, or if you can move your code from where it is, to this clean and uniform state, then you should be able to take full advantage of chained promises.

RELATED POST

10 Myths About Docker That Stop Developers Cold

Docker Recipes Update: Speed Up npm install In Mou...

Update and a Bonus Recipe for the Docker Recipes e...

A Sneak Peak at Docker Recipes for Node.js Develop...

Docker Recipes for Node.js: Pre-sale Dates & ...

Filed Under: Callbacks, Community, ES6, JavaScript, NodeJS, Patterns, Promises, WatchMeCode

About derickbailey

Derick Bailey is a developer, entrepreneur, author, speaker and technology leader in central Texas (north of Austin). He's been a professional developer since the late 90's, and has been writing code since the late 80's. In his spare time, he gets called a spamming marketer by people on Twitter, and blurts out all of the stupid / funny things he's ever done in his career on his email newsletter.

DERICK BAILEY AROUND THE WEB

Twitter: @derickbailey

Google+: DerickBailey

Screencasts: WatchMeCode.net

eBook: Building Backbone Plugins