Gotchas about async/await and Promises

Massimo Artizzu Nov 18 '17 *Updated on Mar 27, 2018*

#javascript #webdev #promises #es6

JavaScript has always had an asynchronous nature. Most of the web's APIs were synchronous though, but things eventually changed also thanks to functions being first-class citizens in JavaScript. Now, basically every new JavaScript API is designed as asynchronous. (Even the decades-old API for cookies might get an asynchronous re-vamp.)

Problems came when we had to *serialize* those asynchronous tasks, which means executing an asynchronous method at the end of a callback, and so on. In practice we had to do this:

```
$.get('/api/movies/' + movieCode, function(movieData) {
  $.get('/api/directors/' + movieData.director, function(directorData) {
    $.get('/api/studios/' + directorData.studio, function(studioData) {
      $.get('/api/locations/' + studioData.hq, function(locationData) {
    });
  });
});
```

Yes, that's the pyramid of doom. (And that's just a simple case: when you had to execute asynchronous tasks *in parallel*, that's when things got crazy.)

Then `Promise` s came, together with ES2015. With the... huh, *promise* to turn our code into this:

```
doSomething()
  .then(data => doStuff(data))
  .then(result => doOtherStuff(result))
  .then(outcome => showOutcome(outcome));
```

Nice, easy to read, semantic. In practice, more often than expected, we ended up with something like this instead:

```
doSomething().then(data => {
  doStuff(data).then(result => {
    doOtherStuff(data, result).then(outcome => {
      showOutcome(outcome, result, data);
    });
  });
});
```

It's the pyramid all over again! What has happened?!

result of the previous one, but also on the results of prior tasks too. Of course, you could do this:

```
let _data;
let _result;
doSomething().then(data => {
  _data = data;
  return doStuff(data);
}).then(result => {
  _result = result;
  return doOtherStuff(_data, result);
}).then(outcome => {
  showOutcome(outcome, _result, _data);
});
```

I won't even start to point how awkward and jarring that is. We're declaring the variable we need way before assigning its value, and if you, like me, suffer from OCD of "must-use-`const`" whenever the value of a variable isn't expected to change, you'll feel those `let`s as stabs in your pupils.

But then ES2016 came, and it brought the `async`/`await` sweetness! That promised (...) to turn our mess into this sync-like code:

```
const data = await doSomething();
const result = await doStuff(data);
const outcome = await doOtherStuff(data, result);
await showOutcome(outcome, result, data);
```

Nice!

No promise should be left uncaught

This is especially true, since promise rejections are *not* thrown errors. Although browsers and Node got smarter in recent times, promises with unhandled rejections used to fail *silently*... and deadly. Not to mention the mess to debug.

Now, what happens when `await`ing a rejected promise?

It throws.

Solving this issue is therefore easy-peasy, you might think. We've had `try...catch` for eons:

```
try {
  const data = await doSomething();
} catch (e) {
  console.error('Haha, gotcha!', e.message);
}
```

... Now, I must ask. How many of you JavaScript developers feel *comfortable* writing `try...catch`es? JavaScript has always been such a forgiving language that most of the times we just needed to check if a value was `null` or something like that. Add that JavaScript isn't quite performant when dealing with `try...catch`, and you have a recipe for an awkward reaction.

(Although in recent times things have changed a bit. While before V8 didn't optimize code inside `try...catch`, it's not the

Chrome 60 and Node 8.3, and I guess other browser vendors will catch up soon. So we'll end up with the usual performance problems of native `Promise` s.)

Scoped woes

Ok, we had to change our nice `await` one-liners with 5 lines of `try...catch` . That's already bad enough, but unfortunately it's not all. Let's examine again the code:

```
try {
  const data = await doSomething();
} catch (e) { ... }

// Doing something with data...
```

Well, we're out of luck again: *we can't use* `data` because it's out of our scope! Its scope, in fact, lives only inside the `try` block! How can we solve that?

... And the solution is, again, ugly:

```
let data;
try {
  data = await doSomething();
} catch (e) { ... }

// Doing something with data...
```

compelled to use `var` again! *And actually it won't be that bad,* since with `async` / `await` your functions will probably have a *flat* scope and your variables will have a closure scope anyway. But linters will tell your code sucks, your OCD won't let you sleep, coffee will taste sour, kittens will get sad and so on.

The only progress we've made is that we can use `let` *right before* the `try...catch` block, so things are a little less jarring:

```
let data;
try {
  data = await doSomething();
} catch (e) { ... }

let result;
try {
  result = await doStuff(data);
} catch (e) { ... }
```

The *Pokémon* solution

If you care about kittens being happy, you need to do something. Here's the common, easy, f-that-I-ve-stuff-to-do way:

```
try {
  const data = await doSomething();
  const result = await doStuff(data);
  const outcome = await doOtherStuff(data, result);
  await showOutcome(outcome, result, data);
} catch(e) {
```

Let me tell you, you still won't get sleep. Yes, you "gotta catch 'em all", but not like that. You've been taught countless of times that this is bad and you should feel bad, *especially* in JavaScript where you can't rely on multiple `catch` blocks for telling exception types apart, and instead you have to check them with `instanceof` or even the `message` property.

Do by the book

You pinky-promise that you'll *never* do that and do things as they should be. Likely scenario:

```
try {
  const data = await doSomething();
  const result = apparentlyInnocentFunction(data);
  return result;
} catch(e) {
  console.error('Error when doingSomething, check your data', e.message);
}
```

We're catching rejected promises, that's right. But what's happening after that? Nothing much, we're just calling an innocent (apparently) function to transform the data.

... Are we sure about that? Is that function all that innocent?

The problem is that a `try...catch` is *still* a `try...catch`. It won't just catch out `await`ed promises, it will catch *all* the correctly, we should use `try...catch` to wrap *just* the `await` ed promise.

Ugly. Verbose. Painful. But necessary.

And we've already seen this when just using `Promise` s, so this shouldn't be new. In short, don't do this:

```
doSomething.then(data => {
  const result = apparentlyInnocentFunction(data);
  return result;
}).catch(error => {
  console.error('Error when doingSomething, check your data', e.message);
});
```

Do this instead:

```
doSomething.then(data => {
  const result = apparentlyInnocentFunction(data);
  return result;
}, error => { // <= catching with the second argument of `then`!
  console.error('Error when doingSomething, check your data', e.message);
});
```

A good compromise?

So, how can we deal with this mess? A nice solution would be getting rid of `try...catch` blocks altogether and taking advantage of `Promise` s and remember that they have a `catch` are.

```
const data = await doSomething()
    .catch(e => console.error('Error when doingSomething', e.message));
if (!data) { /* Bail out somehow */ }
```

Personally, I've mixed feelings about this. Is it nicer? Are we mixing techniques? I guess most of this depends on what we're dealing with, so here you are.

Just keep in mind that:

- `await` doesn't just resolve `Promise` s, but *any* object that has a `then` method - a *thenable* (try this: `await {then() {console.log('Foo!')}}`);
- more than that, you can `await` *any* object, even strings or `null` .

This means that `then` or `catch` might not be defined, or not what you think they are. (Also remember that `.catch(f)` is sugar for `.then(null, f)` , so the latter is all you need to define a thenable.)

Hidden parallelism

How to resolve multiple parallel (or better, concurrent) promises all at once? We've always been relying on `Promise.all` :

```
// or in terms of await:
await Promise.all([ doSomething(), doSomethingElse() ]);
```

But Cory House has recently given this tip:

Cory House 🏠
@housecor
Tip: With async/await, you can run multiple async operations in parallel. How? Put the await statements on the same…
twitter.com/i/web/status/9…

16:20 PM - 13 Nov 2017

318 697

So it's possible to resolve concurrent promises *without* it too:

```
const a = doSomething();
const b = doSomethingElse();
// Just like await Promise.all([a, b])
await a, await b;
```

The trick here is that the promises have been *initiated* before being `await` ed. Awaiting the function calls directly instead of `a` and `b` would have resulted in serialized execution.

My suggestions here are: watch out for these possible concurrency problems; and don't be "clever" and try to exploit this. Using `Promise.all` is much clearer in terms of readability.

You might have heard that `async` / `await` is, like many other new features of JavaScript, just *syntactic sugar* for something you could already do with the classic ES5 JavaScript. It's *mostly* true but, just like many other cases (classes, arrow functions, etc.), there's more to it.

As Mathias Bynens recently pointed out, the JS engine must do a lot of work to get a decent stack trace out of `Promise` chains, so using `async` / `await` is undeniably preferable.

The problem is that we can't just use it as we wish. We still have to support older browsers like IE or Node 6.x that don't support the new syntax. But let's not overlook browsers like UC and Samsung Internet that don't support it either! In the end we'll have to transpile it all, and will do that for a while too.

Update (March 2018): Samsung Internet and UC Browser now both support `async` / `await` , but watch out for older versions.

Conclusions

I don't know yours, but my experience with transpiled `async` functions has been... less than optimal so far. It looks like Chrome has some bugs dealing with sourcemaps, or maybe they're not well defined, but whatever.

Do I use `async` / `await` ? Yes, certainly, but I think I'm not using it as much as I'd like due to all the mentioned problems.

a grain of salt.

What's your experience with `async` / `await` ?

❤ 47 🦄 4 🔖 10

Massimo Artizzu + FOLLOW
Senior web developer. Foolish enough to reinvent the wheel.
MaxArt2501 MaxArt2501

Add to the discussion

PREVIEW SUBMIT

K

Nov 18 '17

I switch between async/await and plain promises often. I also mix await with .catch()

Some code gets simpler with await, especially when I need to mix results of multiple promises and following requests are based on the results of the last ones.

```
const user = await getUser();
const posts = await getPosts(user.id);
return posts.map(p => ({...p, author: user.name});
```

Some code gets simpler with plain promises, like parallelisation. When I need to retrieve the data of multiple views in one screen I often drop them off as promise and *then* the result into

```
this.setState({docsLoading: true, foldersLoading: true});
getDocs().then(docs => this.setState({docs, docsLoading: false}));
getFolders().then(folders => this.setState({folders, foldersLoading: false}));
```

10 REPLY

Jonathan Boudreau

Nov 18 '17

```
doSomething().then(data => {
  doStuff(data).then(result => {
    doOtherStuff(data, result).then(outcome => {
      showOutcome(outcome, result, data);
    });
  });
});
```

Instead of this you can do something like the following:

```
const concat = _.curry(_.concat, 2)
doSomething()
  .then(data =>
    doStuff(data).then(concat(data))
  .then([data, result] =>
    doOtherStuff(data, result).then(concat([data, result]))
  )
  .then([data, result, outcome] =>
    showOutcome(data, result, outcome)
  )
```

7 REPLY

K

Nov 18 '17

Or like this:

```
doSomething()
  .then(data => Promise.all([data, doStuff(data)]))
  .then(([data, result]) => Promise.all([data, result, doOtherStuff(data, result)]))
  .then(([data, result, outcome]) => showOutcome(data, result, outcome));
```

9 REPLY

Jonathan Boudreau Nov 18 '17

My main point was that you can still get a "flat" result when you have interdependent calls.

You can also pull this off with callbacks.

2 REPLY

Ben Halpern Nov 19 '17

Nice post. I've had a bit of a modern JS phobia I'm just starting to shake and this helps.

3 REPLY

Ben Halpern

Hey there, we see you aren't signed in. (Yes you, the reader. This is a fake comment.)

Please consider creating an account on dev.to. It literally takes a few seconds and ***we'd appreciate the support so much***.

Plus, no fake comments when you're signed in.

JOIN THE DEV COMMUNITY

wassano Jan 20

Very interesting! I'm not an expert in javascript and I have faced this callback hell sometimes. I use the caolan/async module to solve this in nodejs and also for web pages, is there any problem I'm missing? Like loosing too much performance or other things?

1 REPLY

Kalpesh Mange Nov 20 '17

This is an eye-opener. Also, my kittens like this post. ;) :)

2 REPLY

Classic DEV Post from Jul 8 '17

Seven useful programming habits

Bart Karalus

Few habits positively identified as making an impact in my programming work.

203 31

READ POST SAVE FOR LATER

From one of our Community Sponsors

Under the Hood of the Most Powerful Video JavaScript API

The JW Player Team

In this article, our goal is to demonstrate how to leverage our JavaScript API effectively to deliver a better video experience on your website through code walkthroughs & demos. We'll then wrap up with some details under the hood of JW Player, explaining how we're the fastest player on the web.

37 3

READ POST SAVE FOR LATER

Another Post You Might Like

objects? No... array, please!

Fabio Russo

Use array, instead of an object

47 22

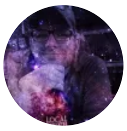
READ POST SAVE FOR LATER

How To Build Your First Chrome Extension

Himashi Hettege Dona - May 31

An alternative to handle state in React: the URL !

GaelS - Jun 1


How to Use PHP Traits

Matt Sparks - May 31

Intro to Realm Database

Lemuel Ogbunude - May 31