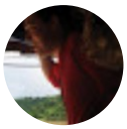

Daniel Brain Follow
works for PayPal, as a lead engineer in Checkout. Opinions expressed herein belong to him and not his employer. daniel@bluesuncorp.co.uk
Jan 18, 2016 · 7 min read

Understand promises before you start using async/await

With Babel now supporting async/await out of the box, and ES2016 (or ES7) just around the corner, more and more people are realizing how awesome this pattern is for writing asynchronous code, using synchronous code structure.

This is a good thing™, and ought to improve code quality a whole lot.

However, what a lot of people may have missed is that the entire foundation for async/await is **promises**. In fact **every *async* function you write will return a promise, and every single thing you *await* will ordinarily be a promise.**

Why am I emphasizing this? Because the majority of javascript written today is written using the callback pattern; a lot of people just never got on the promise bandwagon, and they're missing an important part of the async/await puzzle.

What even is a promise?

I'll keep this brief, since it's been covered extensively elsewhere.

A promise is a special kind of javascript object which *contains* another object. I could have a promise for the integer 17, or the string "hello world", or some arbitrary object, or *anything else* you could normally store in a javascript variable.

How do I access the data in a promise? I use `.then()`:

```
function getFirstUser() {
    return getUsers().then(function(users) {
        return users[0].name;
    });
}
```

How do I catch the errors from a promise chain? I use `.catch()`

```
function getFirstUser() {
    return getUsers().then(function(users) {
        return users[0].name;
    }).catch(function(err) {
        return {
          name: 'default user'
        };
    });
}
```

Even though promises will usually be for 'future' data, once I actually have a promise for something, I really don't need to care whether the data will be there in future, or it's already been retrieved. I just call `then()` in either case. As such, promises *force consistent asynchronicity* (and avoid releasing zalgo). It's like saying, 'this is going to be an async function, whether or not the return value is available now or later'.

Cool... so how does async/await tie in?

Well, consider the above code. `getUsers()` returns a promise. Any promise we have, using ES2016, we can *await*. That's literally all *await* means: it functions in exactly the same way as calling `.then()` on a promise (but without requiring any callback function). So the above code becomes:

```
async function getFirstUser() {
    let users = await getUsers();
    return users[0].name;
}
```

I can *await* any promise I want, whether it's already been resolved or not, whether I created it or not. *await* will simply pause the execution of my method until the value from the promise is available.

So… what about catching errors?

Simple, now we're writing synchronous style code, we can go back to using try/catch:

```
async function getFirstUser() {
    try {
        let users = await getUsers();
        return users[0].name;
    } catch (err) {
```

```
            return {
                name: 'default user'
            };
        }
    }
```

Alright, cool. So there's a way of writing it with promises, and a way of writing it with async/await. Why do I need to care about promises again?

Pitfall 1: not awaiting

If I absentmindedly call

```
let user = getFirstUser();
```

Even though my code doesn't *await*, it's not going to automatically error out!

In fact, I'm under no strict obligation to *await* anything. If I don't, `user` will refer to a promise object (rather than the resolved value), and I won't be able to do much with it. Since javascript has no strict typing, this isn't going to be obvious to me until I try to do something with my `user` variable (which is a promise rather than a user) and probably get a null pointer down the line when I least expect it, in some unrelated code.

So it's important to remember: async functions don't magically wait for themselves. You must *await*, or you'll get a promise instead of the value you expect.

That said, this can be a good thing if you actually want a promise. It gives us more control to do cool stuff like memoizing promises.

Pitfall 2: awaiting multiple values

So, here's one of the rubs of await: under normal usage, I can only await one thing at a time:

```
let foo = await getFoo();
let bar = await getBar();
```

Even though I should be able to get both at the same time, this code will get foo and bar sequentially.

One proposed way of dealing with this was an idea rejected from the ES2016 spec:

```
let [foo, bar] = await* [getFoo(), getBar()];
```

This is a shame, because it's exactly the kind of thing it's nice to have syntax for. Instead, right now, the way to do this is:

```
let [foo, bar] = await Promise.all([getFoo(), getBar()]);
```

This is kinda confusing—aren't we using async/await, not promises?! The reason this works is because async/await and promises *are the same thing under the hood*.

It's far easier to get a mental grasp of this if you understand what Promise.all means, so let's go back to promise basics. Fundamentally, Promise.all will take an *array* of promises, and compose them all into a *single* promise, which resolves **only when every child promise in the array has resolved itself**.

Then, in the above example, we're *awaiting* that 'super promise' we created using Promise.all. So yeah—it really helps to understand what Promise.all means before you start writing code using async/await.

While we're on the topic—I just want to clear up one thing which I've noticed is a common misconception. Promise.all *does not* 'dispatch' or 'create' the promises you pass into it. By the time I've created the array —

```
[getFoo(), getBar()]
```

— these operations are now *already in progress*. All Promise.all is doing is grouping them into a single new promise, and waiting for them to both complete. Promise.all means "Wait for these things" *not* "Do these

things". It is **not equivalent** to async.parallel which actually calls the methods you pass into it.

—

For interest's sake, here's another interesting way to do things in parallel with async/await (though not one I'd recommend):

```
let fooPromise = getFoo();
let barPromise = getBar();
let foo = await fooPromise;
let bar = await barPromise;
```

This code again requires you to understand promises to figure out what's happening.

1. First. we dispatch `getFoo` and `getBar` and save the promises they return in `fooPromise` and `barPromise`.
2. These actions are now in progress, they're happening, there's no stopping them or delaying them
3. We await each promise in turn

Doesn't awaiting mean the two async actions are run in serial? Not so! In this case we've dispatched *both* async actions before we await *anything*, so they're both run concurrently. By the time we start awaiting, it's too late to delay either of them.

Obviously, don't do this. It's not exactly readable. But it's another demonstration of how promises can still come into play in async/await land.

Pitfall 3: your whole stack needs to be async

If I start using await somewhere, I now have the problem that it affects my entire stack. In order to call one of my async functions, ideally the caller itself should be an async function. This has a knock-on effect on my whole stack and makes it difficult to incrementally convert from callbacks to async/await.

[Note: the same is *not* true if you're already using promises in your stack, because remember—async functions return promises, and await

awaits promises, so you're already 90% of the way there in terms of compatibility]

That said: if you understand how promises work, you can get around this by treating the result of an async function as a promise, which itself accepts callbacks. Consider the following:

```
function getFirstUser(callback) {
    return getUsers().then(function(user) {
        return callback(null, users[0].name);
    }).catch(function(err) {
        return callback(err);
    });
}
```

Cool, I just converted an async function (getUsers) into a callback response, without a huge amount of boilerplate. In fact, a lot of promise libraries even do this for you with a `nodeify()` method, like:

```
function getFirstUser(callback) {
    return getUsers().then(function(user) {
        return users[0].name;
    }).nodeify(callback);
}
```

Anyway, what about the other way around? What if I have an async function and I need to call some callback code from within it?

Again, it helps to understand promises here, because really the only way is to transform my callback method into a promise returning method—and in ES6, converting a callback to a promise is pretty easy:

```
function callbackToPromise(method, ...args) {
    return new Promise(function(resolve, reject) {
        return method(...args, function(err, result) {
            return err ? reject(err) : resolve(result);
        });
    });
}
```

Then:

```
async function getFirstUser() {
    let users = await callbackToPromise(getUsers);
    return users[0].name;
}
```

Pitfall 4: Gotta remember to handle errors

The age-old problem with promises is also a problem with async/await: you need to remember to catch errors, or there's the chance they could get lost in the ether.

Consider the following:

```
myApp.endpoint('GET', '/api/firstUser', async function(req,
res) {
    let firstUser = await getFirstUser();
    res.json(firstUser)
});
```

Unless `myApp.endpoint` (which is a function you probably don't own) is promise/async aware, and is calling `await` or `.catch()` on anything returned from the handler function I pass in, any errors are going to be lost.

If you imagine the equivalent code written with promises, you'll see why immediately:

```
myApp.endpoint('GET', '/api/firstUser', function(req, res) {
    return getFirstUser().then(function(firstUser) {
        res.json(firstUser)
    });
});
```

Here, we've passed in a callback to handle the *success* case of `getFirstUser`, but we've neglected to pass in any kind of *error* handler. And because `getFirstUser` is an asynchronous function, even if it throws an error, it's not going to automatically be caught by `myApp.endpoint`.

As such, the top level of your code *always* needs to be enclosed in a try/catch (gotta catch 'em all) to make sure you're handling any errors:

```
myApp.registerEndpoint('GET', '/api/firstUser', async
function(req, res) {
    try {
        let firstUser = await getFirstUser();
        res.json(firstUser)
    } catch (err) {
        console.error(err);
        res.status(500);
    }
});
```

Hopefully soon, more frameworks will be async/await (or promise) aware, which will make this less of a concern.

———

So, what is there to take from all this?

Unless you understand promises, you will come across use cases and bugs that are *really* difficult to understand while using async/await.

Also, as a bonus, even if you're not interested in using babel, you can get your code totally ready to benefit from ES2016's async/await by converting to promises now. Woot!

See the follow-up to this article here