James K Nelson

Introduction to ES6 Promises – The Four Functions You Need To Avoid Callback Hell

posted in **ES6**, **Javascript** on **May 31, 2015** by **James K Nelson**

Update October 2018: *If you need to brush up on promises and async/await, I highly recommend you look instead at my* ***Mastering Asynchronous JavaScript*** *course on Frontend Armory. The course contains 47 live examples and exercises, and also comes with this spiffy* ***Promises and async/await cheatsheet***.

Of course, my original promises guide is still useful today! Here it is:

Apart from being new and shiny, Promises are a great way to clean up your code, reduce dependencies on external libraries, and prepare yourself for `async` and `await` in ES7. Developers who use them swear by them, and developers who don't just don't know what they're missing.

That said, promises can take a lot of work to understand. They feel completely different to the callbacks which we often use to accomplish the same thing, and there are a few surprises in their design which often cause beginners hours of debugging pain.

So why should I learn promises, again?

If the fact that promises are awesome doesn't convince you, maybe the alternative to promises will.

Up until promises arrived, developers in JavaScript land had been using callbacks. In fact, whether you realise it or not, you've probably used them too! `setTimeout` , `XMLHttpRequest` , and basically all browser-based asynchronous functions are callback based.

To demonstrate the problem with callbacks, let's do some animation, just without the HTML and CSS.

Say we want want to do the following:

1. run some code
2. wait one second
3. run some more code
4. wait another second
5. then run some more code again

This is a pattern you'd often use with CSS3 animations. Let's implement using trusty friend `setTimeout` . Our code will look something like this:

```
runAnimation(0);
setTimeout(function() {
    runAnimation(1);
    setTimeout(function() {
        runAnimation(2);
    }, 1000);
}, 1000);
```

Looks awful, right? But imagine for a moment you had 10 steps instead of 3 – you'd have so much whitespace you could build a pyramid. It gets so bad, in fact, that people have come up with a name for it – *callback hell*. And these callback pyramids appear *everywhere* – in handling HTTP requests, database manipulation, animation, inter-process communication, and all manner of other places. There is one exception, though:

They don't appear in code which uses promises.

But what *are* promises?

Perhaps the easiest way to grok promises is to work with what you already know and contrast them with callbacks. There are four major differences:

1. Callbacks are functions, promises are objects

Callbacks are just blocks of code which can be run *in response* to events such as as timers going off or messages being received from the server. Any function can be a callback, and every callback is a function.

Promises are objects which store information about *whether or not* those events have happened yet, and if they have, *what their outcome is*.

2. Callbacks are passed as arguments, promises are returned

Callbacks are defined independently of the functions they are called from – they are passed in as arguments. These functions then store the callback, and call it when the event actually happens.

Promises are created *inside* of asynchronous functions (those which might not return a response until later), and then returned. When an event happens, the asynchronous function will update the promise to notify the outside world.

3. Callbacks handle success and failure, promises don't handle anything

Callbacks are generally called with information on whether an operation succeeded or failed, and must be able to handle both scenarios.

Promises don't handle anything by default, but success and failure handlers are attached later.

4. Callbacks can represent multiple events, promises represent at most one

Callbacks can be called multiple times by the functions they are passed to.

Promises can only represent one event – they are either successful once, or failed once.

With this in mind, let's jump into the details.

The four functions you need to know

1. new Promise(fn)

ES6 Promises are instances of the `Promise` built-in, and are created by calling `new Promise` with a single function as an argument. For example:

```
// Creates a Promise instance which doesn't do anything.
// Don't worry, I'll explain this in more detail in a moment.
promise = new Promise(function() {});
```

Running `new Promise` will immediately call the function passed in as an argument. This function's purpose is to inform the Promise object when the event which the promise represents has **resolved** (i.e. successfully completed), or been **rejected** (i.e. failed).

In order to do so, the function you pass to the constructor can take two arguments, which are themselves callable functions – `resolve` and `reject` . Calling `resolve(value)` will mark the promise as resolved and cause any success handlers will be run. Call `reject(error)` , will cause any failure handler to be run. You should not call both. `resolve` and `reject` both take an argument which represents the event's details.

Let's apply this to our animation example above. The above example uses the `setTimeout` function, which takes a callback – but we want to return a promise instead. `new Promise` lets us do this:

```
// Return a promise which resolves after the specified interval
function delay(interval) {
    return new Promise(function(resolve) {
        setTimeout(resolve, interval);
    });
}

var oneSecondDelay = delay(1000);
```

Great, we've now got a promise which resolves after a second has passed. I know you're probably itching to learn how to actually *do* something after a second has passed – we'll get to that in a moment in the second function you need to know, `promise.then` .

The function we passed to `new Promise` in the above example takes a `resolve` argument, but we've omitted `reject` . This is because setTimeout never fails, and thus there is no scenario where we'd need to call reject.

But say we're specifically interested in setting timeouts for animation, and if the animation is going to fail due to lack of browser support, we want to know immediately instead of after the timeout. If an `isAnimationSupported(step)` function is available, we could implement this with `reject` :

```
function animationTimeout(step, interval) {
    new Promise(function(resolve, reject) {
        if (isAnimationSupported(step)) {
            setTimeout(resolve, interval);
        } else {
            reject('animation not supported');
        }
    });
}

var firstKeyframe = animationTimeout(1, 1000);
```

Finally, it is important to note that if an exception is thrown within the passed-in function, the promise will automatically be marked as rejected, with the exception object being stored as the rejected value, just as if it had been passed as the argument to `reject` .

One way to think of this is that the contents of each function you pass to `new Promise` are wrapped in a try/catch statement, like this:

```
var promise = new Promise(function(resolve, reject) {
    try {
        // your code
    }
    catch (e) {
        reject(e)
    }
});
```

So. Now you understand how to create a promise. But once we have one, how do we handle success/failure? We use it's `then` method.

2. promise.then(onResolve, onReject)

`promise.then(onResolve, onReject)` allows us to assign handlers to a promise's events. Depending on which arguments you supply, you can handle success, failure, or both:

```
// Success handler only
promise.then(function(details) {
    // handle success
});

// Failure handler only
promise.then(null, function(error) {
    // handle failure
});

// Success & failure handlers
promise.then(
    function(details) { /* handle success */ },
    function(error) { /* handle failure */ }
);
```

Tip: Don't try and handle errors from the `onResolve` handler in the `onError` handler of the same `then`, it doesn't work.

```
// This will cause tears and consternation
promise.then(
    function() {
        throw new Error('tears');
    },
    function(error) {
        // Never gets called
        console.log(error)
    }
);
```

If this is all `promise.then` did, it wouldn't really have any any advantage over callbacks. Luckily, it does more: handlers passed to `promise.then` don't just handle the result of the previous promise – whatever they return is turned into a *new* promise.

> `promise.then` **always returns a promise**

Naturally, this works with numbers, strings and any old value:

```
delay(1000)
    .then(function() {
        return 5;
    })
    .then(function(value) {
        console.log(value); // 5
    });
```

But more importantly, it also works with other promises – returning a promise from a `then` handler passes that promise through to the return value of `then`. This allows you to chain promises:

```
delay(1000)
    .then(function() {
        console.log('1 second elapsed');
        return delay(1000);
    })
    .then(function() {
        console.log('2 seconds elapsed');
    });
```

And as you can see, chained promises no longer cause callback pyramids. No matter how many levels of callback hell you'd be in for, the equivalent promises code is *flat*.

Can you use what you've learned so far to flatten the animation example from before, using the above `delay` function? I've written it out again for your convenience – *once you've had a shot* you can check your work by touching or hovering over the blank box below.

```
runAnimation(0);
setTimeout(function() {
    runAnimation(1);
    setTimeout(function() {
        runAnimation(2);
    }, 1000);
}, 1000);
```

```

runAnimation(0);
delay(1000)
    .then(function() {
        runAnimation(1);
        return delay(1000);
    })
    .then(function() {
        runAnimation(2);
    });
```

Edit & run this code

Which one do you find easier to understand? The promise version, or the callback version? Let me know in the comments!

While everything so far has been relatively straightforward, there are a couple of things which trip people up. For example:

> **Rejection handlers in `promise.then` return resolved promises, not rejected ones.**

The fact that *rejection handlers* return a *resolved promise* by default caused me a lot of pain when I was first learning – don't let it happen to you. Here is an example of what to watch out for:

```
new Promise(function(resolve, reject) {
    reject(' :( ');
})
    .then(null, function() {
        // Handle the rejected promise
        return 'some description of :(';
    })
    .then(
        function(data) { console.log('resolved: '+data); },
        function(error) { console.error('rejected: '+error); }
    );
```

What does this print to the screen? Once you're certain, check your answer by touching or hovering over the empty box below:

resolved: some description of 🙁

Edit & run the above code

The take-away here is that if you want to process your errors, make sure you don't just return the processed value: reject the processed value. So, instead of:

```
return 'some description of :('
```

Use this magic incarnation to return a failed promise with a given value instead of a successful one:

```
// I'll explain this a little later
return Promise.reject({anything: 'anything'});
```

Alternatively, if you can throw an error, you can utilise the fact that

> `promise.then` **turns exceptions into rejected promises**

This means that you can cause a handler (either success or failure) to return a rejected promise by doing this:

```
throw new Error('some description of :(')
```

Keep in mind that just like the function passed to `new Promise`, *any* exception thrown within the handlers passed to `promise.then` will be turned into a rejected promise – as opposed to landing in your console like you might expect. Because of this, it is important to make sure that you finish each `then` chain with a rejection-only handler – if you leave this out, you'll invariably spend hours pulling your hair out over missing error messages (see **Are JavaScript Promises swallowing your errors?** for another solution).

Here is a contrived example to demonstrate this point:

```
delay(1000)
    .then(function() {
        throw new Error("oh no.");
    })
    .then(null, function(error) {
        console.error(error);
    });
```

This is so important, in fact, that there is a shortcut. It happens to be function number three:

3. promise.catch(onReject)

This one is simple. `promise.catch(handler)` is equivalent to `promise.then(null, handler)`.

No seriously, thats all there is to it.

One pattern you're generally going to want to follow is to put a `catch` at the end of each of your `then` chains. Let's go back to the animation example to demonstrate.

Say we've got a 3-step animation, with a 1 second gap between each step. Each step could throw an exception – say, due to lack of browser support – so we'll follow each `then` with a `catch` containing a backup function which causes the same changes without animation.

Can you write this using `delay` from above, assuming that each animation step can be called with `runAnimation(number)`, and each backup step can be called with `runBackup(number)`? Treat each step separately, in case the browser can run some of the animations, but not all of them. Give it a shot, and *if you get stuck*, touch or hover over the blank box below for an answer.

```
try {
    runAnimation(0);
}
catch (e) {
    runBackup(0);
}

delay(1000)
    .then(function() {
        runAnimation(1);
        return delay(1000);
    })
    .catch(function() {
        runBackup(1);
    })
    .then(function() {
        runAnimation(2);
    })
    .catch(function() {
        runBackup(2);
    });
```

How does your answer compare with mine? If you're unsure about why I've done it the way I have, leave a comment!

One interesting thing in the above example is the similarity between the try/catch block and the promise blocks. Some people like to think about promises as a kind of delayed try/catch block – I don't, but I guess it can't hurt.

Aaand, that wraps up *the three functions you need to know to use promises*. But to use them well, you need to know four.

4. Promise.all([promise1, promise2, ...])

`Promise.all` is amazing. What it accomplishes is so painful using callbacks that I chickened out from even writing a callback-based example, yet the abstraction it provides is so simple.

What does it do? It returns a promise which resolves when *all* of it's argument promises have resolved, or is rejected when *any* of it's argument promises are rejected. **The returned promise resolves to an array containing the results of every promise, or fails with the error with which the first promise was rejected.**

What is it useful for? Well, say we wanted to animate two things in parallel, followed by a third in series.

```
parallel animation 1  \
                                     + - subsequent animation
parallel animation 2  /
```

You *could* rip your hair out doing this with callbacks. You *could* find, download and include a 3rd-party library which makes it a little easier.

Or, you could just use `Promise.all`.

Let's say that we have three functions which start an animation and return promises which resolve when the animation is done – `parallelAnimation1()`, `parallelAnimation2()` and `finalAnimation()`. The above flow can be implemented as follows:

```
Promise.all([
    parallelAnimation1(),
    parallelAnimation2()
]).then(function() {
    finalAnimation();
});
```

Simple, right?

Other usages for `Promise.all` include downloading multiple HTTP requests concurrently, running multiple processes concurrently, or making multiple database requests concurrently. What do these all have in common? **`Promise.all` makes concurrency easy.**

Test your knowledge

Now that you know the four functions which you need to know, let's stretch your brain muscles with an exercise.

Your task is to:

1. Download two files from a server
2. Extract some data from each of them
3. Use this to decide on a third file to download
4. Display the data from the third file to the user with `alert()`, or in case of an error, display an `alert()` with the received error message instead

While the functions to download the first two files return promises, the function which accesses the third file unfortunately only accepts a callback.

Here are the functions available:

- `initialRequestA()` returns a promise to an A object
- `initialRequestB()` returns a promise to an B object
- `getOptionsFromInitialData(a, b)` returns the `options` argument required for `finalRequest`
- `finalRequest(options, callback)` retrieves the third file from the server, calling `callback(error, data)` when done. `data` is `undefined` in case of an error, and `error` is `undefined` in case of success (*note: this is a common pattern in node.js*)

You can test your work with **this jsbin**, where I've set up the above functions for you.

Once you've written your own answer, check it with mine by touching or hovering over the box below.

```
function finalRequestPromise(options) {
    return new Promise(function(resolve, reject) {
        finalRequest(options, function(error, data) {
            if (error) {
                reject(error);
            }
            else {
                resolve(data);
            }
        });
    });
}

Promise
    .all([initialRequestA(), initialRequestB()])
    .then(function(results) {
        var options = getOptionsFromInitialData(results[0], results[1]);
        return finalRequestPromise(options);
    })
    .then(
        function(file) { alert(file); },
        function(error) { alert('ERROR: '+error); }
    );
```

BONUS: The two incredibly useful functions that you can now learn with minimal effort

Now that you know the four functions which you *need* to know, there are two functions which are incredibly *useful* to know, and basically come for free on top of the above.

Promise.resolve(value)

`Promise.resolve(value)` returns a promise which resolves to it's argument. It is equivalent to this:

```
new Promise(function(resolve) {
    resolve();
}).then(function() {
    return value;
});
```

It is useful when you need a promise and don't have one, or aren't sure if you do – for example when you're building an array of promises to pass to Promise.all, or when you're not sure if the argument to a function is a promise or a value.

Promise.reject(value)

`Promise.reject(value)` returns a promise which fails with the passed in value. It is equivalent to this:

```
new Promise(function(resolve, reject) {
    reject(value);
});
```

This is incredibly useful when you want to process error objects in a `catch` handler, but don't want to return a *successful* promise afterwards.

Useful links

Learn more about ES6 at my guide to **The Bits You'll Actually Use**.

Want to read more about promises instead? You may find these helpful:

- **Mastering Asynchronous JavaScript**
- **Why Async: JavaScript and the real world**
- **The `await` operator**
- **Are JavaScript Promises swallowing your errors?**
- **`Promise` at Mozilla Developer Network**
- **The Bluebird promise library**

Thanks so much for reading – and if you there is anything that you don't find crystal clear, but sure to leave a comment!