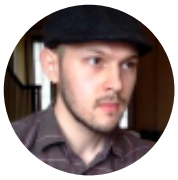
We have a problem with promises



By: Nolan Lawson
Published: 18 May 2015

Fellow JavaScripters, it's time to admit it: we have a problem with promises.

No, not with promises themselves. Promises, as defined by the A+ spec, are awesome.

The big problem, which has revealed itself to me over the course of the past year, as I've watched numerous programmers struggle with the PouchDB API and other promise-heavy APIs, is this:

Many of us are using promises *without really understanding them*.

If you find that hard to believe, consider this puzzle I recently posted to Twitter:

Q: What is the difference between these four promises?

```
doSomething().then(function () {
  return doSomethingElse();
});

doSomething().then(function () {
  doSomethingElse();
});

doSomething().then(doSomethingElse());

doSomething().then(doSomethingElse);
```

If you know the answer, then congratulations: you're a promises ninja. You have my permission to stop reading this blog post.

For the other 99.99% of you, you're in good company. Nobody who responded to my tweet could solve it, and I myself was surprised by the answer to #3. Yes, even though I wrote the quiz!

The answers are at the end of this post, but first, I'd like to explore why promises are so tricky in the first place, and why so many of us – novices and experts alike – get tripped up by them. I'm also going to offer what I consider to be the singular insight, the *one weird trick*, that makes promises a cinch to understand. And yes, I really do believe they're not so hard after that!

But to start with, let's challenge some common assumptions about promises.

Wherefore promises?

If you read the literature on promises, you'll often find references to the pyramid of doom, with some horrible callback-y code that steadily stretches toward the right side of the screen.

Promises do indeed solve this problem, but it's about more than just indentation. As explained in the brilliant talk "Redemption from Callback Hell", the real problem with callbacks it that they deprive us of keywords like `return` and `throw`. Instead, our program's entire flow is based on*side effects*: one function incidentally calling another one.

And in fact, callbacks do something even more sinister: they deprive us of the *stack*, which is something we usually take for granted in programming languages. Writing code without a stack is a lot like driving a car without a brake pedal: you don't realize how badly you need it, until you reach for it and it's not there.

The whole point of promises is to give us back the language fundamentals we lost when we went async: `return`, `throw`, and the stack. But you have to know how to use promises correctly in order to take advantage of them.

Rookie mistakes

Some people try to explain promises as a cartoon, or in a very noun-oriented way: "Oh, it's this thing you can pass around that represents an asynchronous value."

I don't find such explanations very helpful. To me, promises are all about code structure and flow. So I think it's better to just go over some common mistakes and show how to fix them. I call these "rookie mistakes" in the sense of, "you're a rookie now, kid, but you'll be a pro soon."

Quick digression: "promises" mean a lot of different things to different people, but for the purposes of this article, I'm only going to talk aboutthe official spec, as exposed in modern browsers as `window.Promise`. Not all browsers have `window.Promise` though, so for a good polyfill, check out the cheekily-named Lie, which is about the smallest spec-compliant library out there.

Rookie mistake #1: the promisey pyramid of doom

Looking at how people use PouchDB, which has a largely promise-based API, I see a lot of poor promise patterns. The most common bad practice is this one:

```
remotedb.allDocs({
  include_docs: true,
  attachments: true
}).then(function (result) {
  var docs = result.rows;
  docs.forEach(function(element) {
    localdb.put(element.doc).then(function(response) {
      alert("Pulled doc with id " + element.doc._id + " and added to local d
b.");
    }).catch(function (err) {
      if (err.status == 409) {
        localdb.get(element.doc._id).then(function (resp) {
```

```
          localdb.remove(resp._id, resp._rev).then(function (resp) {
// et cetera...
```

Yes, it turns out you can use promises as if they were callbacks, and yes, it's a lot like using a power sander to file your nails, but you can do it.

And if you think this sort of mistake is only limited to absolute beginners, you'll be surprised to learn that I actually took the above code from the official BlackBerry developer blog! Old callback habits die hard. (And to the developer: sorry to pick on you, but your example is instructive.)

A better style is this one:

```
remotedb.allDocs(...).then(function (resultOfAllDocs) {
  return localdb.put(...);
}).then(function (resultOfPut) {
  return localdb.get(...);
}).then(function (resultOfGet) {
  return localdb.put(...);
}).catch(function (err) {
  console.log(err);
});
```

This is called *composing promises*, and it's one of the great superpowers of promises. Each function will only be called when the previous promise has resolved, and it'll be called with that promise's output. More on that later.

Rookie mistake #2: WTF, how do I use `forEach()` with promises?

This is where most people's understanding of promises starts to break down. As soon as they reach for their familiar `forEach()` loop (or `for` loop, or `while` loop), they have no idea how to make it work with promises. So they write something like this:

```
// I want to remove() all docs
db.allDocs({include_docs: true}).then(function (result) {
  result.rows.forEach(function (row) {
    db.remove(row.doc);
  });
}).then(function () {
  // I naively believe all docs have been removed() now!
});
```

What's the problem with this code? The problem is that the first function is actually returning `undefined`, meaning that the second function isn't waiting for `db.remove()` to be called on all the documents. In fact, it isn't waiting on anything, and can execute when any number of docs have been removed!

This is an especially insidious bug, because you may not notice anything is wrong, assuming PouchDB removes those documents fast enough for your UI to be updated. The bug may only pop up in the odd race conditions, or in certain browsers, at which point it will be nearly impossible to debug.

The TLDR of all this is that `forEach()` / `for` / `while` are not the constructs you're looking for. You want `Promise.all()`:

```
db.allDocs({include_docs: true}).then(function (result) {
  return Promise.all(result.rows.map(function (row) {
    return db.remove(row.doc);
  }));
}).then(function (arrayOfResults) {
  // All docs have really been removed() now!
});
```

What's going on here? Basically `Promise.all()` takes an *array of promises* as input, and then it gives you another promise that only resolves when every one of those other promises has resolved. It is the asynchronous equivalent of a for-loop.

`Promise.all()` also passes an array of results to the next function, which can get very useful, for instance if you are trying to `get()` multiple things from PouchDB. The `all()` promise is also rejected if *any one of its sub-promises are rejected*, which is even more useful.

Rookie mistake #3: forgetting to add .catch()

This is another common mistake. Blissfully confident that their promises could never possibly throw an error, many developers forget to add a `.catch()` anywhere in their code. Unfortunately this means that any thrown errors *will be swallowed*, and you won't even see them in your console. This can be a real pain to debug.

To avoid this nasty scenario, I've gotten into the habit of simply adding the following code to my promise chains:

```
somePromise().then(function () {
  return anotherPromise();
}).then(function () {
  return yetAnotherPromise();
}).catch(console.log.bind(console)); // <-- this is badass
```

Even if you never expect an error, it's always prudent to add a `catch()`. It'll make your life easier, if your assumptions ever turn out to be wrong.

Rookie mistake #4: using "deferred"

This is a mistake I see all the time, and I'm reluctant to even repeat it here, for fear that, like Beetlejuice, merely invoking its name will summon more instances of it.

In short, promises have a long and storied history, and it took the JavaScript community a long time to get them right. In the early days, jQuery and Angular were using this "deferred" pattern all over the place, which has now been replaced with the ES6 Promise spec, as implemented by "good" libraries like Q, When, RSVP, Bluebird, Lie, and others.

So if you are writing that word in your code (I won't repeat it a third time!), you are doing something wrong. Here's how to avoid it.

First off, most promise libraries give you a way to "import" promises from third-party libraries. For instance, Angular's `$q` module allows you to wrap non-`$q` promises using `$q.when()`. So Angular users can wrap PouchDB promises this way:

```
$q.when(db.put(doc)).then(/* ... */); // <-- this is all the code you need
```

Another strategy is to use the revealing constructor pattern, which is useful for wrapping non-promise APIs. For instance, to wrap a callback-based API like Node's `fs.readFile()`, you can simply do:

```
new Promise(function (resolve, reject) {
  fs.readFile('myfile.txt', function (err, file) {
    if (err) {
      return reject(err);
    }
    resolve(file);
  });
}).then(/* ... */)
```

Done! We have defeated the dreaded def... Aha, caught myself. :)

> For more about why this is an anti-pattern, check out the Bluebird wiki page on promise anti-patterns.

Rookie mistake #5: using side effects instead of returning

What's wrong with this code?

```
somePromise().then(function () {
  someOtherPromise();
}).then(function () {
  // Gee, I hope someOtherPromise() has resolved!
  // Spoiler alert: it hasn't.
});
```

Okay, this is a good point to talk about everything you ever need to know about promises.

Seriously, this is the *one weird trick* that, once you understand it, will prevent all of the errors I've been talking about. You ready?

As I said before, the magic of promises is that they give us back our precious `return` and `throw`. But what does this actually look like in practice?

Every promise gives you a `then()` method (or `catch()`, which is just sugar for `then(null, ...)`). Here we are inside of a `then()` function:

```
somePromise().then(function () {
  // I'm inside a then() function!
});
```

What can we do here? There are three things:

1. `return` another promise
2. `return` a synchronous value (or `undefined`)
3. `throw` a synchronous error

That's it. Once you understand this trick, you understand promises. So let's go through each point one at a time.

1. Return another promise

This is a common pattern you see in the promise literature, as in the "composing promises" example above:

```
getUserByName('nolan').then(function (user) {
  return getUserAccountById(user.id);
}).then(function (userAccount) {
  // I got a user account!
});
```

Notice that I'm `return`ing the second promise – that `return` is crucial. If I didn't say `return`, then the `getUserAccountById()` would actually be a *side effect*, and the next function would receive `undefined` instead of the `userAccount`.

2. Return a synchronous value (or undefined)

Returning `undefined` is often a mistake, but returning a synchronous value is actually an awesome way to convert synchronous code into promisey code. For instance, let's say we have an in-memory cache of users. We can do:

```
getUserByName('nolan').then(function (user) {
  if (inMemoryCache[user.id]) {
    return inMemoryCache[user.id];    // returning a synchronous value!
  }
  return getUserAccountById(user.id); // returning a promise!
}).then(function (userAccount) {
  // I got a user account!
});
```

Isn't that awesome? The second function doesn't care whether the `userAccount` was fetched synchronously or asynchronously, and the first function is free to return either a synchronous or asynchronous value.

Unfortunately, there's the inconvenient fact that non-returning functions in JavaScript technically return `undefined`, which means it's easy to accidentally introduce side effects when you meant to return something.

For this reason, I make it a personal habit to *always return or throw* from inside a `then()` function. I'd recommend you do the same.

3. Throw a synchronous error

Speaking of `throw`, this is where promises can get even more awesome. Let's say we want to `throw` a synchronous error in case the user is logged out. It's quite easy:

```
getUserByName('nolan').then(function (user) {
  if (user.isLoggedOut()) {
    throw new Error('user logged out!'); // throwing a synchronous error!
  }
  if (inMemoryCache[user.id]) {
    return inMemoryCache[user.id];       // returning a synchronous value!
  }
  return getUserAccountById(user.id);    // returning a promise!
}).then(function (userAccount) {
  // I got a user account!
}).catch(function (err) {
  // Boo, I got an error!
});
```

Our `catch()` will receive a synchronous error if the user is logged out, and it will receive an asynchronous error if *any of the promises are rejected*. Again, the function doesn't care whether the error it gets is synchronous or asynchronous.

This is especially useful because it can help identify coding errors during development. For instance, if at any point inside of a `then()` function, we do a `JSON.parse()`, it might throw a synchronous error if the JSON is invalid. With callbacks, that error would get swallowed, but with promises, we can simply handle it inside our `catch()` function.

Advanced mistakes

Okay, now that you've learned the single trick that makes promises dead-easy, let's talk about the edge cases. Because of course, there are always edge cases.

These mistakes I'd classify as "advanced," because I've only seen them made by programmers who are already fairly adept with promises. But we're going to need to discuss them, if we want to be able to solve the puzzle I posed at the beginning of this post.

Advanced mistake #1: not knowing about `Promise.resolve()`

As I showed above, promises are very useful for wrapping synchronous code as asynchronous code. However, if you find yourself typing this a lot:

```
new Promise(function (resolve, reject) {
  resolve(someSynchronousValue);
}).then(/* ... */);
```

You can express this more succinctly using `Promise.resolve()`:

```
Promise.resolve(someSynchronousValue).then(/* ... */);
```

This is also incredibly useful for catching any synchronous errors. It's so useful, that I've gotten in the habit of beginning nearly all of my promise-returning API methods like this:

```
function somePromiseAPI() {
  return Promise.resolve().then(function () {
    doSomethingThatMayThrow();
    return 'foo';
  }).then(/* ... */);
}
```

Just remember: any code that might `throw` synchronously is a good candidate for a nearly-impossible-to-debug swallowed error somewhere down the line. But if you wrap everything in `Promise.resolve()`, then you can always be sure to `catch()` it later.

Similarly, there is a `Promise.reject()` that you can use to return a promise that is immediately rejected:

```
Promise.reject(new Error('some awful error'));
```

Advanced mistake #2: `catch()` isn't exactly like `then(null, ...)`

I said above that `catch()` is just sugar. So these two snippets are equivalent:

```
somePromise().catch(function (err) {
  // handle error
});

somePromise().then(null, function (err) {
  // handle error
});
```

However, that doesn't mean that the following two snippets are equivalent:

```
somePromise().then(function () {
  return someOtherPromise();
}).catch(function (err) {
  // handle error
});

somePromise().then(function () {
  return someOtherPromise();
}, function (err) {
  // handle error
});
```

If you're wondering why they're not equivalent, consider what happens if the first function throws an error:

```
somePromise().then(function () {
  throw new Error('oh noes');
```

```
}).catch(function (err) {
  // I caught your error! :)
});

somePromise().then(function () {
  throw new Error('oh noes');
}, function (err) {
  // I didn't catch your error! :(
});
```

As it turns out, when you use the `then(resolveHandler, rejectHandler)` format, the `rejectHandler` *won't actually catch an error* if it's thrown by the `resolveHandler` itself.

For this reason, I've made it a personal habit to never use the second argument to `then()`, and to always prefer `catch()`. The exception is when I'm writing asynchronous Mocha tests, where I might write a test to ensure that an error is thrown:

```
it('should throw an error', function () {
  return doSomethingThatThrows().then(function () {
    throw new Error('I expected an error!');
  }, function (err) {
    should.exist(err);
  });
});
```

Speaking of which, Mocha and Chai are a lovely combination for testing promise APIs. The pouchdb-plugin-seed project has some sample teststhat can get you started.

Advanced mistake #3: promises vs promise factories

Let's say you want to execute a series of promises one after the other, in a sequence. That is, you want something like `Promise.all()`, but which doesn't execute the promises in parallel.

You might naïvely write something like this:

```
function executeSequentially(promises) {
  var result = Promise.resolve();
  promises.forEach(function (promise) {
    result = result.then(promise);
  });
  return result;
}
```

Unfortunately, this will not work the way you intended. The promises you pass in to `executeSequentially()` will *still* execute in parallel.

The reason this happens is that you don't want to operate over an array of promises at all. Per the promise spec, as soon as a promise is created, it begins executing. So what you really want is an array of *promise factories*:

```
function executeSequentially(promiseFactories) {
  var result = Promise.resolve();
  promiseFactories.forEach(function (promiseFactory) {
    result = result.then(promiseFactory);
  });
  return result;
}
```

I know what you're thinking: "Who the hell is this Java programmer, and why is he talking about factories?" A promise factory is very simple, though – it's just a function that returns a promise:

```
function myPromiseFactory() {
  return somethingThatCreatesAPromise();
}
```

Why does this work? It works because a promise factory doesn't create the promise until it's asked to. It works the same way as a `then` function – in fact, it's the same thing!

If you look at the `executeSequentially()` function above, and then imagine `myPromiseFactory` being substituted inside of `result.then(...)`, then hopefully a light bulb will click in your brain. At that moment, you will have achieved promise enlightenment.

Advanced mistake #4: okay, what if I want the result of two promises?

Often times, one promise will depend on another, but we'll want the output of both promises. For instance:

```
getUserByName('nolan').then(function (user) {
  return getUserAccountById(user.id);
}).then(function (userAccount) {
  // dangit, I need the "user" object too!
});
```

Wanting to be good JavaScript developers and avoid the pyramid of doom, we might just store the `user` object in a higher-scoped variable:

```
var user;
getUserByName('nolan').then(function (result) {
  user = result;
  return getUserAccountById(user.id);
}).then(function (userAccount) {
  // okay, I have both the "user" and the "userAccount"
});
```

This works, but I personally find it a bit kludgey. My recommended strategy: just let go of your preconceptions and embrace the pyramid:

```
getUserByName('nolan').then(function (user) {
  return getUserAccountById(user.id).then(function (userAccount) {
    // okay, I have both the "user" and the "userAccount"
  });
});
```

...at least, temporarily. If the indentation ever becomes an issue, then you can do what JavaScript developers have been doing since time immemorial, and extract the function into a named function:

```
function onGetUserAndUserAccount(user, userAccount) {
  return doSomething(user, userAccount);
}

function onGetUser(user) {
  return getUserAccountById(user.id).then(function (userAccount) {
    return onGetUserAndUserAccount(user, userAccount);
  });
}

getUserByName('nolan')
  .then(onGetUser)
  .then(function () {
  // at this point, doSomething() is done, and we are back to indentation 0
});
```

As your promise code starts to get more complex, you may find yourself extracting more and more functions into named functions. I find this leads to very aesthetically-pleasing code, which might look like this:

```
putYourRightFootIn()
  .then(putYourRightFootOut)
  .then(putYourRightFootIn)
  .then(shakeItAllAbout);
```

That's what promises are all about.

Advanced mistake #5: promises fall through

Finally, this is the mistake I alluded to when I introduced the promise puzzle above. This is a very esoteric use case, and it may never come up in your code, but it certainly surprised me.

What do you think this code prints out?

```
Promise.resolve('foo').then(Promise.resolve('bar')).then(function (result) {
  console.log(result);
});
```

If you think it prints out `bar`, you're mistaken. It actually prints out `foo`!

The reason this happens is because when you pass `then()` a non-function (such as a promise), it

actually interprets it as `then(null)`, which causes the previous promise's result to fall through. You can test this yourself:

```
Promise.resolve('foo').then(null).then(function (result) {
  console.log(result);
});
```

Add as many `then(null)`s as you want; it will still print `foo`.

This actually circles back to the previous point I made about promises vs promise factories. In short, you *can* pass a promise directly into a `then()` method, but it won't do what you think it's doing. `then()` is supposed to take a function, so most likely you meant to do:

```
Promise.resolve('foo').then(function () {
  return Promise.resolve('bar');
}).then(function (result) {
  console.log(result);
});
```

This will print `bar`, as we expected.

So just remind yourself: always pass a function into `then()`!

Solving the puzzle

Now that we've learned everything there is to know about promises (or close to it!), we should be able to solve the puzzle I originally posed at the start of this post.

Here is the answer to each one, in graphical format so you can better visualize it:

Puzzle #1

```
doSomething().then(function () {
  return doSomethingElse();
}).then(finalHandler);
```

Answer:

```
doSomething
|-----------------|
                  doSomethingElse(undefined)
                  |-----------------|
                                    finalHandler(resultOfDoSomethingElse)
                                    |-----------------|
```

Puzzle #2

```
doSomething().then(function () {
  doSomethingElse();
```

```
}).then(finalHandler);
```

Answer:

```
doSomething
|----------------|
                 doSomethingElse(undefined)
                 |-----------------|
                 finalHandler(undefined)
                 |-----------------|
```

Puzzle #3

```
doSomething().then(doSomethingElse())
  .then(finalHandler);
```

Answer:

```
doSomething
|----------------|
doSomethingElse(undefined)
|--------------------------------|
                 finalHandler(resultOfDoSomething)
                 |-----------------|
```

Puzzle #4

```
doSomething().then(doSomethingElse)
  .then(finalHandler);
```

Answer:

```
doSomething
|----------------|
                 doSomethingElse(resultOfDoSomething)
                 |-----------------|
                                   finalHandler(resultOfDoSomethingElse)
                                   |-----------------|
```

If these answers still don't make sense, then I encourage you to re-read the post, or to define the `doSomething()` and `doSomethingElse()` methods and try it out yourself in your browser.

> **Clarification:** for these examples, I'm assuming that both `doSomething()` and `doSomethingElse()` return promises, and that those promises represent something done outside of the JavaScript event loop (e.g. IndexedDB, network, `setTimeout`), which is

And for more advanced uses of promises, check out my promise protips cheat sheet.

why they're shown as being concurrent when appropriate. Here's a JSBin to demonstrate.

Final word about promises

Promises are great. If you are still using callbacks, I strongly encourage you to switch over to promises. Your code will become smaller, more elegant, and easier to reason about.

And if you don't believe me, here's proof: a refactor of PouchDB's map/reduce module to replace callbacks with promises. The result: 290 insertions, 555 deletions.

Incidentally, the one who wrote that nasty callback code was… me! So this served as my first lesson in the raw power of promises, and I thank the other PouchDB contributors for coaching me along the way.

That being said, promises aren't perfect. It's true that they're better than callbacks, but that's a lot like saying that a punch in the gut is better than a kick in the teeth. Sure, one is preferable to the other, but if you had a choice, you'd probably avoid them both.

While superior to callbacks, promises are still difficult to understand and error-prone, as evidenced by the fact that I felt compelled to write this blog post. Novices and experts alike will frequently mess this stuff up, and really, it's not their fault. The problem is that promises, while similar to the patterns we use in synchronous code, are a decent substitute but not quite the same.

In truth, you shouldn't have to learn a bunch of arcane rules and new APIs to do things that, in the synchronous world, you can do perfectly well with familiar patterns like `return`, `catch`, `throw`, and for-loops. There shouldn't be two parallel systems that you have to keep straight in your head at all times.

Awaiting async/await

That's the point I made in "Taming the asynchronous beast with ES7", where I explored the ES7 `async`/`await` keywords, and how they integrate promises more deeply into the language. Instead of having to write pseudo-synchronous code (with a fake `catch()` method that's kinda like `catch`, but not really), ES7 will allow us to use the real `try`/`catch`/`return` keywords, just like we learned in CS 101.

This is a huge boon to JavaScript as a language. Because in the end, these promise anti-patterns will still keep cropping up, as long as our tools don't tell us when we're making a mistake.

To take an example from JavaScript's history, I think it's fair to say thatJSLint and JSHint did a greater service to the community than *JavaScript: The Good Parts*, even though they effectively contain the same information. It's the difference between being told *exactly the mistake you just made in your code*, as opposed to reading a book where you try to understand other people's mistakes.

The beauty of ES7 `async`/`await` is that, for the most part, your mistakes will reveal themselves as syntax/compiler errors rather than subtle runtime bugs. Until then, though, it's good to have a grasp of what promises are capable of, and how to use them properly in ES5 and ES6.

So while I recognize that, like *JavaScript: The Good Parts*, this blog post can only have a limited impact, it's hopefully something you can point people to when you see them making these same mistakes. Because there are still way too many of us who just need to admit: "I have a problem with promises!"

Update: it's been pointed out to me that Bluebird 3.0 will print out warnings that can prevent many of the

mistakes I've identified in this post. So using Bluebird is another great option while we wait for ES7!