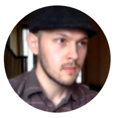
Taming the asynchronous beast with ES7 (https://github.com/ 03- 05- taming- the- async- beast- with- es7.md)



By: Nolan Lawson (https://twitter.com/nolanlawson)
Published: 05 March 2015

One of the trickiest aspects of PouchDB is that its API is asynchronous. I see no shortage of confused questions on Stack Overflow, Github, and IRC, and most often they stem from a misunderstanding of callbacks and promises.

We can't really help it. PouchDB is an abstraction over IndexedDB, WebSQL, LevelDB (in Node), and CouchDB (via Ajax). All of those APIs are asynchronous; hence PouchDB must be asynchronous.

When I think of elegant database APIs, however, I'm still struck by the simplicity of LocalStorage:

```
if (!localStorage.foo) {
  localStorage.foo = 'bar';
};
console.log(localStorage.foo);
```

To work with LocalStorage, you simply treat it like a magical JavaScript object that happens to persist your data. It uses the same synchronous toolset that you're already used to from working with JavaScript itself.

For all of LocalStorage's (http://www.html5rocks.com/en/tutorials/offline/quota-research/) faults (https://blog.mozilla.org/tglek/2012/02/22/psa-dom-local-storage-considered-harmful/) , the ergonomics of this API go a long way to explain its continuing popularity. People keep using LocalStorage, because it's simple and works exactly as expected.

Promises aren't a panacea

For PouchDB, we can try to mitigate the complexity of asynchronous APIs with promises, and that certainly helps us escape the pyramid of doom (https://medium.com/@wavded/managing-node-js-callback-hell-1fe03ba8baf) .

However, promisey code is still hard to read, because promises are basically a bolt-on replacement for language primitives like `try`, `catch`, and `return`:

```
var db = new PouchDB('mydb');
db.post({}).then(function (result) { // post a new doc
  return db.get(result.id);          // fetch the doc
}).then(function (doc) {
  console.log(doc);                  // log the doc
}).catch(function (err) {
  console.log(err);                  // log any errors
});
```

As JavaScript developers, we now have two parallel systems – sync and async – that we have to keep straight in our heads. And this gets even worse as our control flow becomes more complex, and we need to reach for APIs like `Promise.all()` and `Promise.resolve()`. Or maybe we just opt for one (https://github.com/petkaantonov/bluebird) of (https://github.com/tildeio/rsvp.js) the (https://github.com/caolan/async) many (https://github.com/kriskowal/q) helper (https://github.com/cujojs/when) libraries (https://msdn.microsoft.com/en-us/library/windows/apps/br211867.aspx) and pray we can understand the documentation.

Until recently, this was the best we could hope for. But all of that changes with ES7.

Enter ES7

What if I told you that, with ES7, you could rewrite the above code to look like this:

```
let db = new PouchDB('mydb');
try {
  let result = await db.post({});
  let doc = await db.get(result.id);
  console.log(doc);
} catch (err) {
  console.log(err);
}
```

And what if I told you that, thanks to tools like Babel.js (https://babeljs.io/) and Regenerator (http://facebook.github.io/regenerator/), you can transpile that down to ES5 and run it in a browser *today*?

(/static/img/orson_welles_clapping.gif)

Please ladies and gentlemen, hold your applause until the end of the blog post.

First, let's take a look at how ES7 is accomplishing this amazing feat.

Async functions

ES7 gives us a new kind of function, the `async function`. Inside of an `async function`, we have a new keyword, `await`, which we use to "wait for" a promise:

```
async function myFunction() {
  let result = await somethingThatReturnsAPromise();
  console.log(result); // cool, we have a result
}
```

If the promise resolves, we can immediately interact with it on the next line. And if it rejects, then an error is thrown. So `try`/`catch` actually works again!

```
async function myFunction() {
  try {
    await somethingThatReturnsAPromise();
  } catch (err) {
    console.log(err); // oh noes, we got an error
  }
}
```

This allows us to write code that looks synchronous on the surface, but is actually asynchronous under the hood. The fact that the API returns a promise instead of blocking the event loop is just an implementation detail.

REMEMBER WHEN YOU COULD JUST USE "RETURN" AND "TRY/CATCH"?
PEPPERIDGE FARM REMEMBERS
(/static/img/pepperidge_farm_remembers.png)

And the best part is, we can use this *today* with any library that returns promises. PouchDB is such a library, so let's use it to test our theory.

Managing errors and return values

First, consider a common idiom in PouchDB: we want to `get()` a document by `_id` if it exists, or return a new document if it doesn't.

With promises, you'd have to write something like this:

```
db.get('docid').catch(function (err) {
  if (err.name === 'not_found') {
    return {}; // new doc
  }
  throw err; // some error other than 404
}).then(function (doc) {
  console.log(doc);
})
```

With async functions, this becomes:

```
let doc;
try {
  doc = await db.get('docid');
} catch (err) {
  if (err.name === 'not_found') {
    doc = {};
  } else {
    throw err; // some error other than 404
  }
}
console.log(doc);
```

Much more readable! This is almost the exact same code we would write if `db.get()` directly returned a document rather than a promise. The only difference is that we have to add the `await` keyword when we call any promise-returning function.

Potential gotchas

There are a few subtle issues that I ran into while playing with this, so it's good to be aware of them.

First off, anytime you `await` something, you need to be inside an async function. So if your code relies heavily on PouchDB, you may find that you write lots of async functions, but very few regular functions.

Another, more insidious problem is that you have to be careful to wrap your code in `try`/`catch`es, or else a promise might be rejected, in which case the error is silently swallowed. (!)

My advice is to ensure that your async functions are entirely surrounded by `try`/`catch`es, at least at the top level:

```
async function createNewDoc() {
  let response = await db.post({}); // post a new doc
  return await db.get(response.id); // find by id
}

async function printDoc() {
  try {
    let doc = await createNewDoc();
    console.log(doc);
  } catch (err) {
    console.log(err);
  }
}
```

Loops

Async functions get really impressive when it comes to iteration. For instance, Let's say that we want to insert some documents into the database, but *sequentially*. That is, we want the promises to execute one after the other, not concurrently.

Using standard ES6 promises, we'd have to roll our own promise chain:

```
var promise = Promise.resolve();
var docs = [{}, {}, {}];

docs.forEach(function (doc) {
  promise = promise.then(function () {
    return db.post(doc);
  });
});

promise.then(function () {
  // now all our docs have been saved
});
```

This works, but it sure is ugly. It's also error-prone, because if you accidentally do:

```
docs.forEach(function (doc) {
  promise = promise.then(db.post(doc));
});
```

Then the promises will actually execute *concurrently*, which can lead to unexpected results.

With ES7, though, we can just use a regular for-loop:

```
let docs = [{}, {}, {}];

for (let i = 0; i < docs.length; i++) {
  let doc = docs[i];
  await db.post(doc);
}
```

This (very concise) code does the same thing as the promise chain! We can make it even shorter by using `for...of`:

```
let docs = [{}, {}, {}];

for (let doc of docs) {
  await db.post(doc);
}
```

Note that you cannot use a `forEach()` loop here. If you were to naïvely write:

```
let docs = [{}, {}, {}];

// WARNING: this won't work
docs.forEach(function (doc) {
  await db.post(doc);
});
```

Then Babel.js will fail with a somewhat opaque error:

```
Error : /../script.js: Unexpected token (38:23)
> 38 |     await db.post(doc);
     |           ^
```

This is because you cannot use `await` from within a normal function. You have to use an async function.

However, if you try to use an async function, then you will get a more subtle bug:

```
let docs = [{}, {}, {}];

// WARNING: this won't work
docs.forEach(async function (doc, i) {
  await db.post(doc);
  console.log(i);
});
console.log('main loop done');
```

This will compile, but the problem is that this will print out:

```
main loop done
0
1
2
```

What's happening is that the main function is exiting early, because the `await` is actually in the sub-function. Furthermore, this will execute each promise *concurrently*, which is not what we intended.

The lesson is: be careful when you have any function inside your async function. The `await` will only pause its parent function, so check that it's doing what you actually think it's doing.

Concurrent loops

If we do want to execute multiple promises concurrently, though, then this is pretty easy to accomplish with ES7.

Recall that with ES6 promises, we have `Promise.all()`. Let's use it to return an array of values from an array of promises:

```
var docs = [{}, {}, {}];

return Promise.all(docs.map(function (doc) {
  return db.post(doc);
})).then(function (results) {
  console.log(results);
});
```

In ES7, we can do this is a more straightforward way:

```
let docs = [{}, {}, {}];
let promises = docs.map((doc) => db.post(doc));

let results = [];
for (let promise of promises) {
  results.push(await promise);
}
console.log(results);
```

The most important parts are 1) creating the `promises` array, which starts invoking all the promises immediately, and 2) that we are `await`ing those promises within the main function. If we tried to use `Array.prototype.map`, then it wouldn't work:

```
let docs = [{}, {}, {}];
let promises = docs.map((doc) => db.post(doc));

// WARNING: this doesn't work
let results = promises.map(async function(promise) {
  return await promise;
});

// This will just be a list of promises :(
console.log(results);
```

The reason this doesn't work is because we are `await`ing inside of the sub-function, and not the main function. So the main function exits before we are really done waiting.

If you don't mind using `Promise.all`, you can also use it to tidy up the code a bit:

```
let docs = [{}, {}, {}];
let promises = docs.map((doc) => db.post(doc));

let results = await Promise.all(promises);
console.log(results);
```

Presumably this could look even nicer if we used array comprehesions (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Array_comprehensions) . However, the spec isn't final yet, so it's currently not supported (https://github.com/facebook/regenerator/issues/182) by Regenerator.

Caveats

ES7 is still very bleeding-edge. Async functions aren't supported in either Node.js or io.js, and you have to set some experimental flags to even get Babel to consider it. Officially, the async/await spec (https://github.com/lukehoban/ecmascript-asyncawait#status-of-this-proposal) is still in the "proposal" stage.

Also, you'll need to include the Regenerator runtime and ES6 shims in your transpiled code for this to work in ES5 browsers. For me, that added up to about 60KB, minified and gzipped. For many developers, that's just way too much to ship down the wire.

However, all of these new tools are very fun to play with, and they paint a bright picture of what working with asynchronous libraries will look like in the sunny ES7 future.

So if you want to play with it yourself, I've put together a small demo library (https://github.com/nolanlawson/async-functions-in-pouchdb) . To get started, just check out the code, run `npm install && npm run build`, and you're good to go. And for more about ES7, check out this talk by Jafar Husain (https://www.youtube.com/watch?v=DqMFX91ToLw).

Conclusion

Async functions are an empowering new concept in ES7. They give us back our lost `return`s and `try`/`catch`es, and they reward the knowledge we've already gained from writing synchronous code with new idiioms that look a lot like the old ones, but are much more performant.

Most importantly, async functions make APIs like PouchDB's a lot easier to work with. So hopefully this will lead to fewer user errors and confusion, as well as more elegant and readable code.

And who knows, maybe folks will finally abandon LocalStorage, and opt for a more modern client-side database.

(https://twitter.com/pouchdb) (https://github.com/rvagg/node-levelup) (https://github.com/pouchdb/pouchdb)

(https://travis-ci.org/pouchdb/pouchdb) (http://couchdb.apache.org/) (https://saucelabs.com)

Learn

Getting Started (/getting-started.html)
API Guide (/api.html)
Wiki (https://github.com/pouchdb/pouchdb/wiki)

Discuss

Mailing List (https://groups.google.com/forum/#!forum/pouchdb)
IRC (irc://freenode.net/#pouchdb)
Slack (http://slack.pouchdb.com/)
Twitter (http://twitter.com/pouchdb)
StackOverflow (http://stackoverflow.com/questions/tagged/pouchdb)

Contribute

Contributing (https://github.com/pouchdb/pouchdb/blob/master/CONTRIBUTING.md)
Source (https://github.com/pouchdb/pouchdb)
Issues (https://github.com/pouchdb/pouchdb/issues)
Apache License (https://github.com/pouchdb/pouchdb/blob/master/LICENSE)