Understanding JavaScript's async await

Earlier this week we took a look at new features coming in ES2016. Today we'll learn about `async` / `await`.

Nicolás Bevacqua

Published a year ago | *16 minute read* | *21*

The `async` / `await` feature didn't make the cut for ES2016, but that doesn't mean it won't be coming to JavaScript. At the time of this writing, it's a *Stage 3* proposal, and actively being worked on. The feature is already in Edge, and *should it land in another browser* it'll reach Stage 4 – *paving its way for inclusion in the next Edition of the language (see also: TC39 Process).*

We've heard about this feature for a while, but let's drill down into it and see how it works. To be able to grasp the contents of this article, you'll need a solid understanding of promises and generators. These resources should help you out.

- ES6 Overview in 350 Bullet Points
- ES6 Promises in Depth
- ES6 Generators in Depth
- Asynchronous I/O with Generators & Promises

- Promisees Visualization Tool

Using Promises

Let’s suppose we had code like the following. Here I’m wrapping an HTTP request in a `Promise`. The promise fulfills with the `body` when successful, and is rejected with an `err` reason otherwise. It pulls the HTML for a random article from this blog every time.

```
var request = require('request');

function getRandomPonyFooArticle () {
  return new Promise((resolve, reject) => {
    request('https://ponyfoo.com/articles/random', (err, res, body) => {
      if (err) {
        reject(err); return;
      }
      resolve(body);
    });
  });
}
```

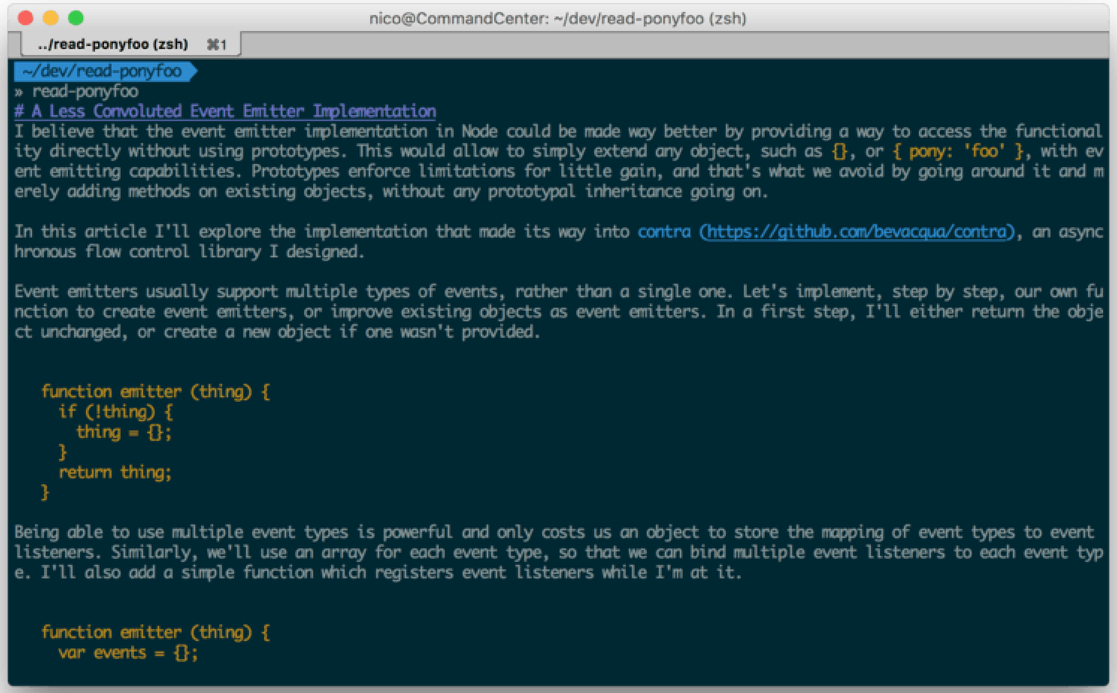
Typical usage of the promised code shown above is below. There, we build a promise chain transforming the HTML page into Markdown of a subset of its DOM, and then into Terminal-friendly output, to finally print it using `console.log`. Always remember to add `.catch` handlers to your promises.

```
var hget = require('hget');
var marked = require('marked');
var Term = require('marked-terminal');

printRandomArticle();
```

```
function printRandomArticle () {
  getRandomPonyFooArticle()
    .then(html => hget(html, {
      markdown: true,
      root: 'main',
      ignore: '.at-subscribe,.mm-comments,.de-sidebar'
    }))
    .then(md => marked(md, {
      renderer: new Term()
    }))
    .then(txt => console.log(txt))
    .catch(reason => console.error(reason));
}
```

When ran, that snippet of code produces output as shown in the following screenshot.



```
~/dev/read-ponyfoo
» read-ponyfoo
# A Less Convoluted Event Emitter Implementation
I believe that the event emitter implementation in Node could be made way better by providing a way to access the functional
ity directly without using prototypes. This would allow to simply extend any object, such as {}, or { pony: 'foo' }, with ev
ent emitting capabilities. Prototypes enforce limitations for little gain, and that's what we avoid by going around it and m
erely adding methods on existing objects, without any prototypal inheritance going on.

In this article I'll explore the implementation that made its way into contra (https://github.com/bevacqua/contra), an async
hronous flow control library I designed.

Event emitters usually support multiple types of events, rather than a single one. Let's implement, step by step, our own fu
nction to create event emitters, or improve existing objects as event emitters. In a first step, I'll either return the obje
ct unchanged, or create a new object if one wasn't provided.


   function emitter (thing) {
     if (!thing) {
       thing = {};
     }
     return thing;
   }

Being able to use multiple event types is powerful and only costs us an object to store the mapping of event types to event
listeners. Similarly, we'll use an array for each event type, so that we can bind multiple event listeners to each event typ
e. I'll also add a simple function which registers event listeners while I'm at it.


   function emitter (thing) {
     var events = {};
```

Screenshot

That code was *"better than using callbacks"*, when it comes to how sequential it feels to read the code.

Using Generators

We've already explored generators as a way of making the `html` available in a synthetic *"synchronous"* manner in the past. Even though the code is now somewhat synchronous, there's quite a bit of wrapping involved, and generators may not be the most straightforward way of accomplishing the results that we want, so we might end up sticking to Promises anyways.

```
function getRandomPonyFooArticle (gen) {
  var g = gen();
  request('https://ponyfoo.com/articles/random', (err, res, body) => {
    if (err) {
      g.throw(err); return;
    }
    g.next(body);
  });
}

getRandomPonyFooArticle(function* printRandomArticle () {
  var html = yield;
  var md = hget(html, {
    markdown: true,
    root: 'main',
    ignore: '.at-subscribe,.mm-comments,.de-sidebar'
  });
  var txt = marked(md, {
    renderer: new Term()
  });
  console.log(txt);
});
```

Keep in mind you should wrap the `yield` call in a `try` / `catch` block to preserve the error handling we had added when using promises.

Needless to say, using generators like this *doesn't scale well*. Besides involving an unintuitive syntax into the mix, your iterator code will be highly coupled to the generator function that's being consumed. That means you'll have to change it often as you add new `await` expressions to the generator. A better alternative is to use the upcoming **Async Function**.

Using `async` / `await`

When *Async Functions* finally hit the road, we'll be able to take our `Promise` -based implementation and have it take advantage of the synchronous-looking generator style. Another benefit in this approach is that you won't have to change `getRandomPonyFooArticle` at all, as long as it returns a promise, it can be awaited.

Note that `await` may only be used in functions marked with the `async` keyword. It works similarly to generators, suspending execution in your context until the promise settles. If the awaited expression isn't a promise, its casted into a promise.

```
read();

async function read () {
  var html = await getRandomPonyFooArticle();
  var md = hget(html, {
    markdown: true,
    root: 'main',
    ignore: '.at-subscribe,.mm-comments,.de-sidebar'
  });
  var txt = marked(md, {
    renderer: new Term()
  });
  console.log(txt);
}
```

Again, – and just like with generators – keep in mind that you should wrap `await` in `try / catch` so that you can capture and handle errors in awaited promises from within the `async` function.

Furthermore, an *Async Function* always returns a `Promise`. That promise is rejected in the case of uncaught exceptions, and it's otherwise resolved to the return value of the `async` function. This enables us to invoke an `async` function and mix that with regular promise-based continuation as well. The following example shows how the two may be combined *(see Babel REPL)*.

```
async function asyncFun () {
  var value = await Promise
    .resolve(1)
    .then(x => x * 3)
    .then(x => x + 5)
    .then(x => x / 2);
  return value;
}
asyncFun().then(x => console.log(`x: ${x}`));
// <- 'x: 4'
```

Going back to the previous example, that'd mean we could `return txt` from our `async read` function, and allow consumers to do continuation using promises or yet another *Async Function*. That way, your `read` function becomes only concerned with pulling terminal-readable Markdown from a random article on Pony Foo.

```
async function read () {
  var html = await getRandomPonyFooArticle();
  var md = hget(html, {
    markdown: true,
    root: 'main',
    ignore: '.at-subscribe,.mm-comments,.de-sidebar'
  });
  var txt = marked(md, {
    renderer: new Term()
  });
  return txt;
}
```

Then, you could further `await read()` in another *Async Function*.

```
  async function write () {
    var txt = await read();
    console.log(txt);
  }
```

Or you could just use promises for further continuation.

```
  read().then(txt => console.log(txt));
```

Fork in the Road

In asynchronous code flows, it is commonplace to execute two or more tasks concurrently. While **Async Functions** make it easier to write asynchronous code, they also lend themselves to code that is *serial*. That is to say: code that executes **one operation at a time**. A function with multiple `await` expressions in it will be suspended once at a time on each `await` expression until that `Promise` is settled, before unsuspending execution and moving onto the next `await` expression – *not unlike the case we observe with generators and `yield`.*

To work around that you can use `Promise.all` to create a single promise that you can `await` on. Of course, the biggest problem is getting in the habit of using `Promise.all` instead of leaving everything to run in a series, as it'll otherwise make a dent in your code's performance.

The following example shows how you could `await` on three different promises that could be resolved concurrently. Given that `await` suspends your `async` function and the `await Promise.all` expression ultimately resolves into a `results` array, we can use destructuring to pull individual results out of that array.

```
async function concurrent () {
  var [r1, r2, r3] = await Promise.all([p1, p2, p3]);
}
```

At some point, there was an `await*` alternative to the piece of code above, where you didn't have to wrap your promises with `Promise.all` . *Babel 5* still supports it, but **it was dropped from the spec** (and from Babel 6) – *because reasons*.

```
async function concurrent () {
  var [r1, r2, r3] = await* [p1, p2, p3];
}
```

You could still do something like `all = Promise.all.bind(Promise)` to obtain a terse alternative to using `Promise.all` . An upside of this is that you could do the same for `Promise.race` , which didn't have an equivalent to `await*` .

```
const all = Promise.all.bind(Promise);
async function concurrent () {
  var [r1, r2, r3] = await all([p1, p2, p3]);
}
```

Error Handling

Note that **errors are swallowed *"silently"*** within an `async` function – *just like inside normal Promises.* Unless we add `try` / `catch` blocks around `await` expressions, uncaught exceptions – regardless of whether they were raised in the body of your `async` function or while its suspended during `await` – will reject the promise returned by the `async` function.

Naturally, this can be seen as a strength: you're able to leverage `try` / `catch` conventions, something you were unable to do with callbacks – and *somewhat* able to with Promises. In this sense, *Async Functions* are akin to generators, where you're also able to leverage `try` / `catch` thanks to function execution suspension turning asynchronous flows into synchronous code.

Furthermore, you're able to catch these exceptions from outside the `async` function, simply by adding a `.catch` clause to the promise they return. While this is a flexible way of combining the `try` / `catch` error handling flavor with `.catch` clauses in *Promises*, it can also lead to confusion and ultimately cause to errors going unhandled.

```
read()
  .then(txt => console.log(txt))
  .catch(reason => console.error(reason));
```

We need to be careful and educate ourselves as to the different ways in which we can notice exceptions and then handle, log, or prevent them.

Using `async` / `await` Today

One way of using *Async Functions* in your code today is through Babel. This involves a series of modules, but you could always come up with a module that wraps all of these in a single one if you prefer that. I included `npm-run` as a helpful way of keeping everything in locally installed packages.

```
npm i -g npm-run
npm i -D \
  browserify \
  babelify \
  babel-preset-es2015 \
  babel-preset-stage-3 \
  babel-runtime \
  babel-plugin-transform-runtime

echo '{
  "presets": ["es2015", "stage-3"],
  "plugins": ["transform-runtime"]
}' > .babelrc
```

The following command will compile `example.js` through `browserify` while using `babelify` to enable support for **Async Functions**. You can then pipe the script to `node` or save it to disk.

```
npm-run browserify -t babelify example.js | node
```

Further Reading

The specification draft for **Async Functions** is surprisingly short, and should make up for an interesting read if you're keen on learning more about this upcoming feature.

I've pasted a piece of code below that's meant to help you understand how `async` functions will work internally. Even though we can't polyfill new keywords, its helpful in terms of understanding what goes on behind the curtains of `async` / `await` .

Namely, it should be useful to learn that *Async Functions* internally leverage both **generators and promises**.

First off, then, the following bit shows how an `async function` declaration could be dumbed down into a regular `function` that returns the result of feeding `spawn` with a generator function – *where we'll consider `await` as the syntactic equivalent for `yield` .*

```
async function example (a, b, c) {
  example function body
}

function example (a, b, c) {
  return spawn(function* () {
    example function body
  }, this);
}
```

In `spawn` , a promise is wrapped around code that will step through the generator function – *made out of user code* – in series, forwarding values to your *"generator" code* (the `async` function's body). In this sense, we can observe that *Async Functions* really **are syntactic sugar** on top of generators and promises, which makes it important that you understand how each of these things work in order to get a better understanding into how you can mix, match, and combine these different flavors of asynchronous code flows together.

```
function spawn (genF, self) {
  return new Promise(function (resolve, reject) {
    var gen = genF.call(self);
    step(() => gen.next(undefined));
    function step (nextF) {
      var next;
      try {
        next = nextF();
      } catch(e) {
        // finished with failure, reject the promise
        reject(e);
        return;
      }
      if (next.done) {
        // finished with success, resolve the promise
        resolve(next.value);
        return;
      }
      // not finished, chain off the yielded promise and `step` again
      Promise.resolve(next.value).then(
        v => step(() => gen.next(v)),
        e => step(() => gen.throw(e))
      );
    }
  });
}
```

The highlighted bits of code should aid you in understanding how the `async` / `await` algorithm iterates over the generator sequence *(of `await` expressions)*, wrapping each item in the sequence in a promise and then chaining that with the next step in the sequence. When the **sequence is over or one of the promises is rejected**, the promise returned by the *underlying generator function* is settled.

Special thanks to @ljharb, @jaydson, @calvinf, @ericclemmons, @sherman3ero, @matthewmolnar3, and @rauschma for reviewing drafts of this article.