ES6 Promises in Depth

Welcome back to ES6 – "Dude, we already had those!" – in Depth series. If you've never been around here before, start with A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new classes sugar on top of prototypes, `let`, `const`, and the "Temporal Dead Zone", iterators, generators, Symbols, Maps, WeakMaps, Sets, and WeakSets, proxies, proxy traps, more proxy traps, reflection, `Number`, `Math`, `Array`, `Object`, `String`, and the module system. This morning is about the `Promise` API in ES6.

Nicolás Bevacqua

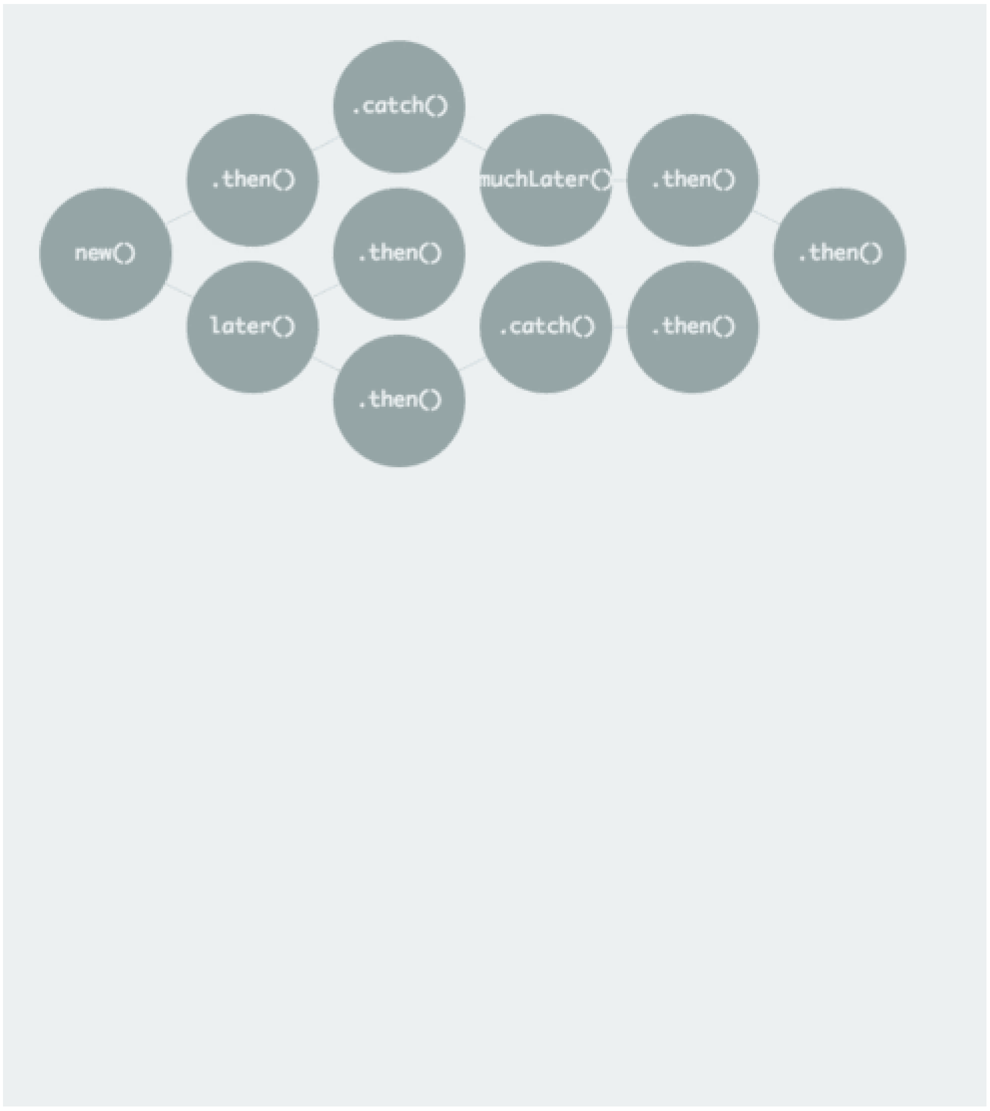
Published a year ago | 30 minute read | 21

Like I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for reading that, and let's go into *Promises* in ES6. Before reading this article you might want to read about arrow functions, as they're heavily used throughout the article; and generators, as they're somewhat related to the concepts discussed here.

I also wanted to mention *Promisees* – a promise visualization playground I made last week. It offers in-browser visualizations of how promises unfold. You can run those visualizations step by step, displaying how promises in any piece of code work. You can also record a gif of those visualizations, a few of which I'll be displaying here in the article. Hope it helps!



An animation showing a complicated promises visualization

If that animation looks insanely complicated to you, read on!

Promises are a very involved paradigm, so we'll take it slow.

Here's a table of contents with the topics we'll cover in this article. Feel free to skip topics you're comfortable about.

Shall we?

What is a `Promise`?

Promises are usually vaguely defined as *"a proxy for a value that will eventually become available"*. They can be used for both synchronous and asynchronous code flows, although they make asynchronous flows easier to reason about – once you've mastered promises, that is.

Consider as an example the *upcoming* `fetch` API. This API is a simplification of `XMLHttpRequest`. It aims to be super simple to use for the most basic use cases: making a `GET` request against a resource relative to the current page over `http(s)` – it also provides a comprehensive API that caters to advanced use cases as well, but that's not our focus for now. In it's most basic incarnation, you can make a request for `GET foo` like so.

```
fetch('foo')
```

The `fetch('foo')` statement doesn't seem all that exciting. It makes a *"fire-and-forget"* `GET` request against `foo` relative to the resource we're currently on. The `fetch` method returns a `Promise`. You can chain a `.then` callback that will be executed once the `foo` resource finishes loading.

```
fetch('foo').then(response => /* do something */)
```

Promises offer an alternative to callbacks and events.

Callbacks and Events

If the `fetch` API used callbacks, you'd get one last parameter that then gets executed whenever fetching ends. Typical asynchronous code flow conventions dictate that we allocate the first parameter for errors *(that may or may not occur)* during the *fetching process*. The rest of the parameters can be used to pass in resulting data. Often, a single parameter is used.

```
fetch('foo', (err, res) => {
  if (err) {
    // handle error
  }
  // handle response
})
```

The callback wouldn't be invoked until the `foo` resource has been fetched, so its execution remains asynchronous and non-blocking. Note that in this model you could only specify **a single callback**, and that callback would be responsible for *all functionality* derived from the response.

Another option might have been to use an *event-driven* API model. In this model the object returned by `fetch` would be able to listen `.on` events, binding as many event handlers as needed for any events. Typically there's an `error` event for when things go awry and a `data` event that's called when the operation completes successfully.

```
fetch('foo')
  .on('error', err => {
    // handle error
  })
  .on('data', res => {
    // handle response
  })
```

In this case, errors usually end up in hard exceptions if no event listener is attached – but that depends on what event emitter implementation is used. Promises are a bit different.

Gist of a `Promise`

Instead of binding event listeners through `.on`, promises offer a slightly different API. The snippet of code shown below displays the actual API of the `fetch` method, which returns a `Promise` object. Much like with events, you can bind as many listeners as you'd like with both `.catch` and `.then`. Note how there's no need for an event type anymore with the declarative methods used by promises.

```
var p = fetch('foo')
p.then(res => {
  // handle response
})
p.catch(error => {
  // handle error
})
```

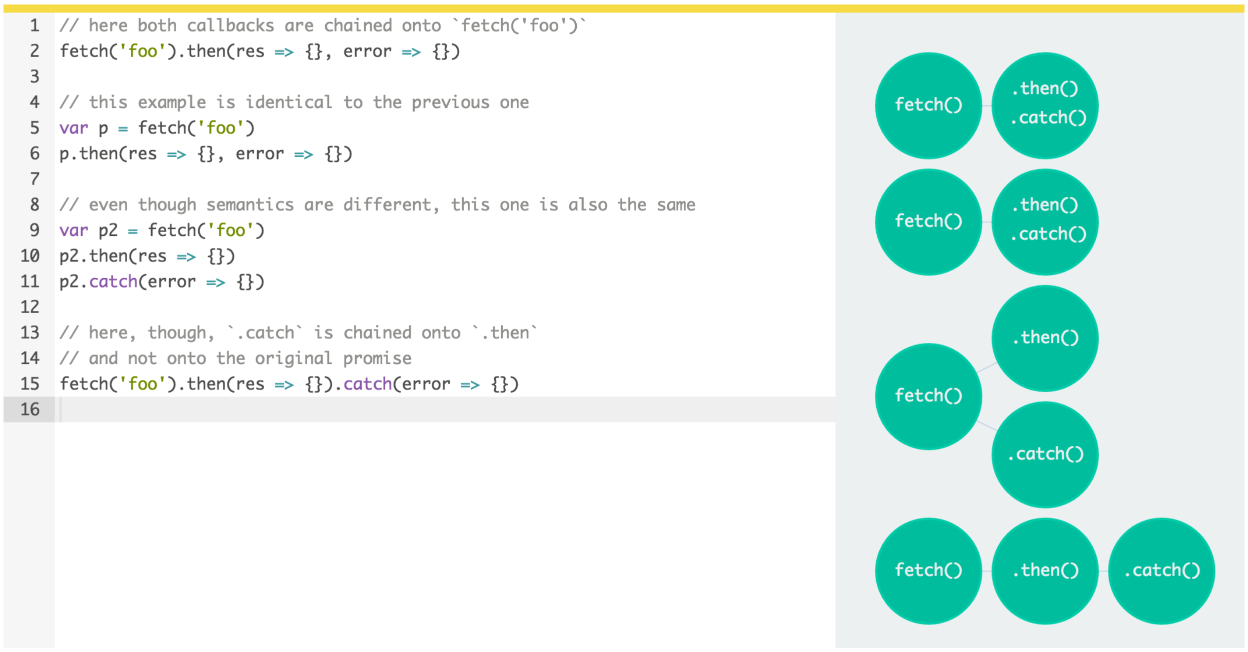
See [this example][(http://buff.ly/1KtWGUD)] on Promisees

Also note that `.then` is able to register a reaction to rejections as its second argument. The above could be expressed as the following piece of code.

```
fetch('foo')
  .then(
    res => {
      // handle response
    },
    err => {
      // handle error
    }
  )
```

See [this example][(http://buff.ly/1V8xpHI)] on Promisees

Just like you can omit the error reaction in `.then(fulfillment)`, you can also omit the reaction to *fulfillment*. Using `.then(null, rejection)` is equivalent to `.catch(rejection)`. Note that `.then` and `.catch` return **a new promise every time**. That's important because chaining can have wildly different results depending on where you append a `.then` or a `.catch` call onto. See the following example to understand the difference.



```
// here both callbacks are chained onto `fetch('foo')`
fetch('foo').then(res => {}, error => {})

// this example is identical to the previous one
var p = fetch('foo')
p.then(res => {}, error => {})

// even though semantics are different, this one is also the same
var p2 = fetch('foo')
p2.then(res => {})
p2.catch(error => {})

// here, though, `.catch` is chained onto `.then`
// and not onto the original promise
fetch('foo').then(res => {}).catch(error => {})

```

Differences when chaining promises

We'll get more in depth into these two methods in a bit. Let's look at a brief history of promises before doing that.

Promises in Time

Promises aren't all that new. Like *most things in computer science*, the earliest mention of

Promises aren't all that new. Like *most things in computer science*, the earliest mention of Promises can be traced all the way back to the late seventies. According to the *Internet*, they made their first appearance in JavaScript in 2007 – in a library called `MochiKit`. Then `Dojo` adopted it, and `jQuery` followed shortly after that.

Then the Promises/A+ specification came out from the CommonJS group *(now famous for their CommonJS module specification)*. In its earliest incarnations, Node.js shipped with promises. Some time later, they were removed from core and everyone switched over to callbacks. Now, promises ship with the ES6 standard and V8 has already implemented them a while back.

The ES6 standard implements **Promises/A+** natively. In the latest versions of Node.js you can use promises without any libraries. They're also available on Chrome 32+, Firefox 29+, and Safari 7.1+.

Shall we go back to the `Promise` API?

Then, Again

Going back to our example – here's some of the code we had. In the simplest use case, this is all we wanted.

```
fetch('foo').then(res => {
  // handle response
})
```

What if an error happens in one of the reactions passed to `.then`? You can catch those with `.catch`. The example in the snippet below logs the error caught when trying to access `.prop`

`.catch`. The example in the snippet below logs the error caught when trying to access `prop` from the *undefined* `a` *property* in `res`.

```
fetch('foo')
  .then(res => res.a.prop.that.does.not.exist)
  .catch(err => console.error(err.message))
// <- 'Cannot read property "prop" of undefined'
```

Note that *where* you tack your reactions onto matters. The following example **won't** print the `err.message` twice – only once. That's because no errors happened in the first `.catch`, so the rejection branch for that promise wasn't executed. Check out the Promisee for a visual explanation of the code below.

```
fetch('foo')
  .then(res => res.a.prop.that.does.not.exist)
  .catch(err => console.error(err.message))
  .catch(err => console.error(err.message))
// <- 'Cannot read property "prop" of undefined'
```

In contrast, the snippet found below *will* print the `err.message` twice. It works by saving a reference to the promise returned by `.then`, and then tacking two `.catch` reactions onto it. The second `.catch` in the previous example was capturing errors produced in the promise returned from the first `.catch`, while in this case both `.catch` branch off of `p`.

```
var p = fetch('foo').then(res => res.a.prop.that.does.not.exist)
p.catch(err => console.error(err.message))
p.catch(err => console.error(err.message))
// <- 'Cannot read property "prop" of undefined'
// <- 'Cannot read property "prop" of undefined'
```

Here's another example that puts that difference the spotlight. The second catch is triggered this time because it's bound to the rejection branch on the first `.catch`.

```
fetch('foo')
  .then(res => res.a.prop.that.does.not.exist)
  .catch(err => { throw new Error(err.message) })
  .catch(err => console.error(err.message))
// <- 'Cannot read property "prop" of undefined'
```

If the first `.catch` call didn't return anything, then nothing would be printed.

```
fetch('foo')
  .then(res => res.a.prop.that.does.not.exist)
  .catch(err => {})
  .catch(err => console.error(err.message))
// nothing happens
```

We should observe, then, that promises can be chained "arbitrarily", that is to say: as we just saw, you can save a reference to any point in the promise chain and then tack more promises on top of it. This is one of the fundamental points to understanding promises.

You can save a reference to any point in the promise chain.

In fact, the last example can be represented as shown below. This snippet makes it much easier to understand what we've discussed so far. Glance over it and then I'll give you some bullet points.

```
var p1 = fetch('foo')
var p2 = p1.then(res => res.a.prop.that.does.not.exist)
var p3 = p2.catch(err => {})
var p4 = p3.catch(err => console.error(err.message))
```

Good boy! Have some bullet points. Or you could just look at the Promisees visualization.

1. `fetch` returns a **brand new** `p1` promise
2. `p1.then` returns a **brand new** `p2` promise
3. `p2.catch` returns a **brand new** `p3` promise
4. `p3.catch` returns a **brand new** `p4` promise
5. When `p1` is settled *(fulfilled)*, the `p1.then` reaction is executed
6. After that `p2` , which is awaiting the pending result of `p1.then` is settled
7. Since `p2` was *rejected*, `p2.catch` reactions are executed *(instead of the `p2.then` branch)*
8. The `p3` promise from `p2.catch` is *fulfilled*, even though it doesn't produce any value nor an error
9. Because `p3` succeeded, `p3.catch` is never executed – the `p3.then` branch would've been used instead

You should think of promises as **a tree structure**. It all starts with a single promise, which we'll later see how to construct. You then add a branch with `.then` or `.catch` . You can tack as many `.then` or `.catch` calls as you want onto each branch, creating new branches, and so on.

Creating a Promise From Scratch

You should now understand how promises work like a tree where you can add branches where you need them, as you need them. But how do you create a promise from scratch? Writing these kinds of `Promise` tutorials is hard because its a chicken and egg situation. People hardly have a need to create a promise from scratch, since libraries usually take care of that. In this article, for instance, I purposely started explaining things using `fetch`, which internally creates a new promise object. Then, each call to `.then` or `.catch` on the promise created by fetch also creates a promise internally, and those promises depend on their parent when it comes to deciding whether the fulfillment branch or the rejection branch should be executed.

Promises can be created from scratch by using `new Promise(resolver)`. The `resolver` parameter is a method that will be used to resolve the promise. It takes two arguments, a `resolve` method and a `reject` method. These promises are fulfilled and rejected, respectively, on the next tick – as seen on Promisees.

```
new Promise(resolve => resolve()) // promise is fulfilled
new Promise((resolve, reject) => reject()) // promise is rejected
```

Resolving and rejecting promises without a value isn't that useful, though. Usually promises will resolve to some `result`, like the response from an AJAX call as we saw with `fetch`. Similarly, you'll probably want to state the `reason` for your rejections – typically using an `Error` object. The code below codifies what you've just read (see the visualization, too).

```
new Promise(resolve => resolve({ foo: 'bar' }))
  .then(result => console.log(result))
  // <- { foo: 'bar' }

new Promise((resolve, reject) =>
  reject(new Error('failed to deliver on my promise to you')))
  .catch(reason => console.log(reason))
  // <- Error: failed to deliver on my promise to you
```

As you may have guessed, there’s nothing inherently synchronous about promises. Fulfillment and rejection can both be completely asynchronous. That’s the whole point of promises! The promise below is fulfilled after two seconds elapse.

```
new Promise(resolve => setTimeout(resolve, 2000))
```

It’s important to note that only the first call made to either of these methods will have an impact – once a promise is settled, it’s result can’t change. The example below creates a promise that’s fulfilled in the alloted time or rejected after a generous timeout (visualization).

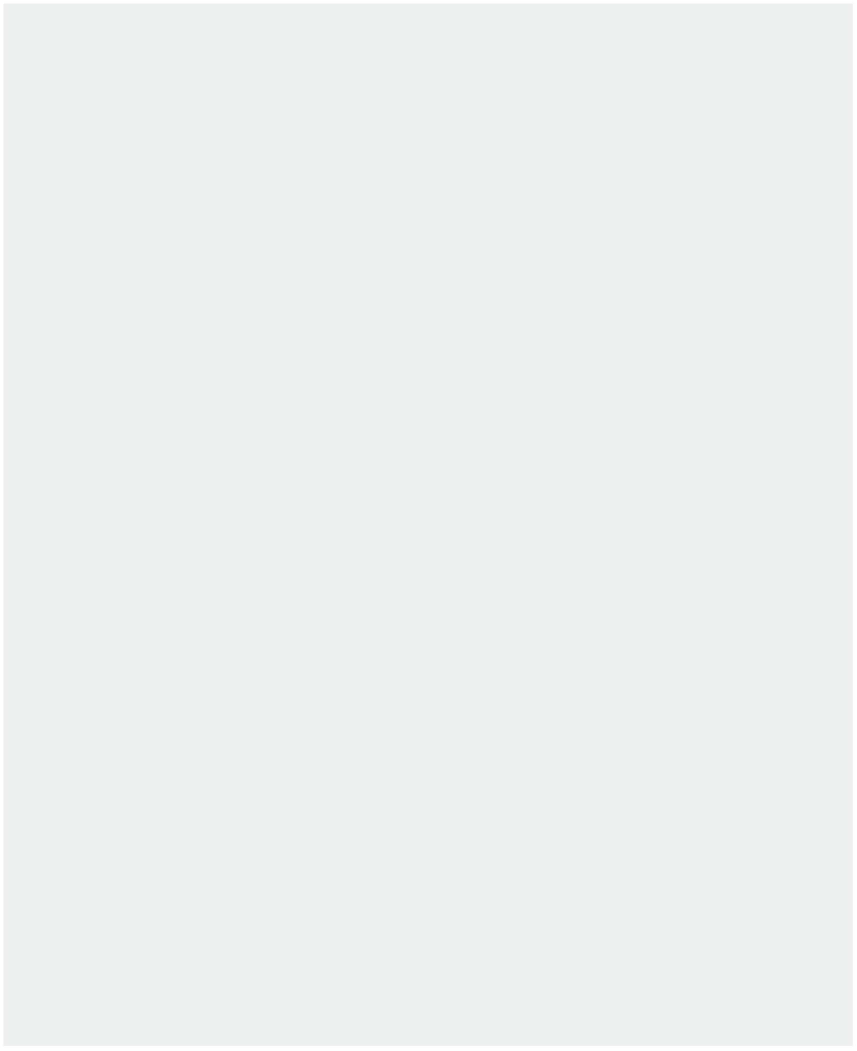
```
function resolveUnderThreeSeconds (delay) {
  return new Promise(function (resolve, reject) {
    setTimeout(resolve, delay)
    setTimeout(reject, 3000)
  })
}
resolveUnderThreeSeconds(2000) // resolves!
resolveUnderThreeSeconds(7000) // fulfillment took so long, it was rejected.
```

See the promises unfold with this animation.

Besides returning resolution values, you could also resolve with *another promise*. What happens in those cases? In the following snippet we create a promise `p` that will be rejected in three seconds. We also create a promise `p2` that will be resolved with `p` in a second. Since `p` is still two seconds out, resolving `p2` won't have an immediate effect. Two seconds later, when `p` is rejected, `p2` will be rejected as well, with the same rejection reason that was provided to `p` .

```
var p = new Promise(function (resolve, reject) {
  setTimeout(() => reject(new Error('fail')), 3000)
})
var p2 = new Promise(function (resolve, reject) {
  setTimeout(() => resolve(p), 1000)
})
p2.then(result => console.log(result))
p2.catch(error => console.log(error))
// <- Error: fail
```

In the animation shown below we can observe how `p2` becomes blocked – *marked in yellow* – waiting for a settlement in `p` .



Animation of a promise blocking another one.

Note that this behavior is only possible for fulfillment branches using `resolve`. If you try to replicate the same behavior with `reject` you'll find that the `p2` promise is just rejected with the `p` promise as the rejection `reason`.

Using `Promise.resolve` and `Promise.reject`

Sometimes you want to create a Promise but you don't want to go through the trouble of using the constructor. The following statement creates a promise that's fulfilled with a result of `'foo'`.

```
new Promise(resolve => resolve('foo'))
```

If you already know the value a promise should be fulfilled with, you can use `Promise.resolve` instead. The following statement is equivalent to the previous one.

```
Promise.resolve('foo')
```

Similarly, if you already know the rejection reason, you can use `Promise.reject` . The next statement creates a promise that's going to settle into a rejection, with `reason` .

```
Promise.reject(reason)
```

What else should we know about settling a promise?

Settling a Promise

Promises can exist in three states: pending, fulfilled, and rejected. Pending is the default state. From there, a promise can be *"settled"* into either fulfillment or rejection. Once a promise is settled, all reactions that are waiting on it are evaluated. Those on the correct branch – *`.then` for fulfillment and `.catch` for rejections* – are executed.

From this point on, the promise is *settled*. If at a later point in time another reaction is chained onto the settled promise, the appropriate branch for that reaction is executed in the next tick of the program. In the example below, `p` is resolved with a value of `100` after two seconds. Then, `100` is printed onto the screen. Two seconds later, another `.then` branch is added onto `p` , but since `p` has already fulfilled, the new branch gets executed right away.

```
var p = new Promise(function (resolve, reject) {
  setTimeout(() => resolve(100), 2000)
})
p.then(result => console.log(result))
// <- 100

setTimeout(() => p.then(result => console.log(result * 20)), 4000)
// <- 2000
```

A promise can return another promise – this is what enables and powers most of their asynchronous behavior. In the previous section, when creating a promise from scratch, we saw that we can `resolve` with another promise. We can also return promises when calling `.then`.

Paying a Promise with another Promise

The example below shows how we use a promise and `.then` another promise that will only be settled once the returned promise also settles. Once that happens, we get back the response from the wrapped promise, and we use the `res.url` to figure out what random article we were graced with.

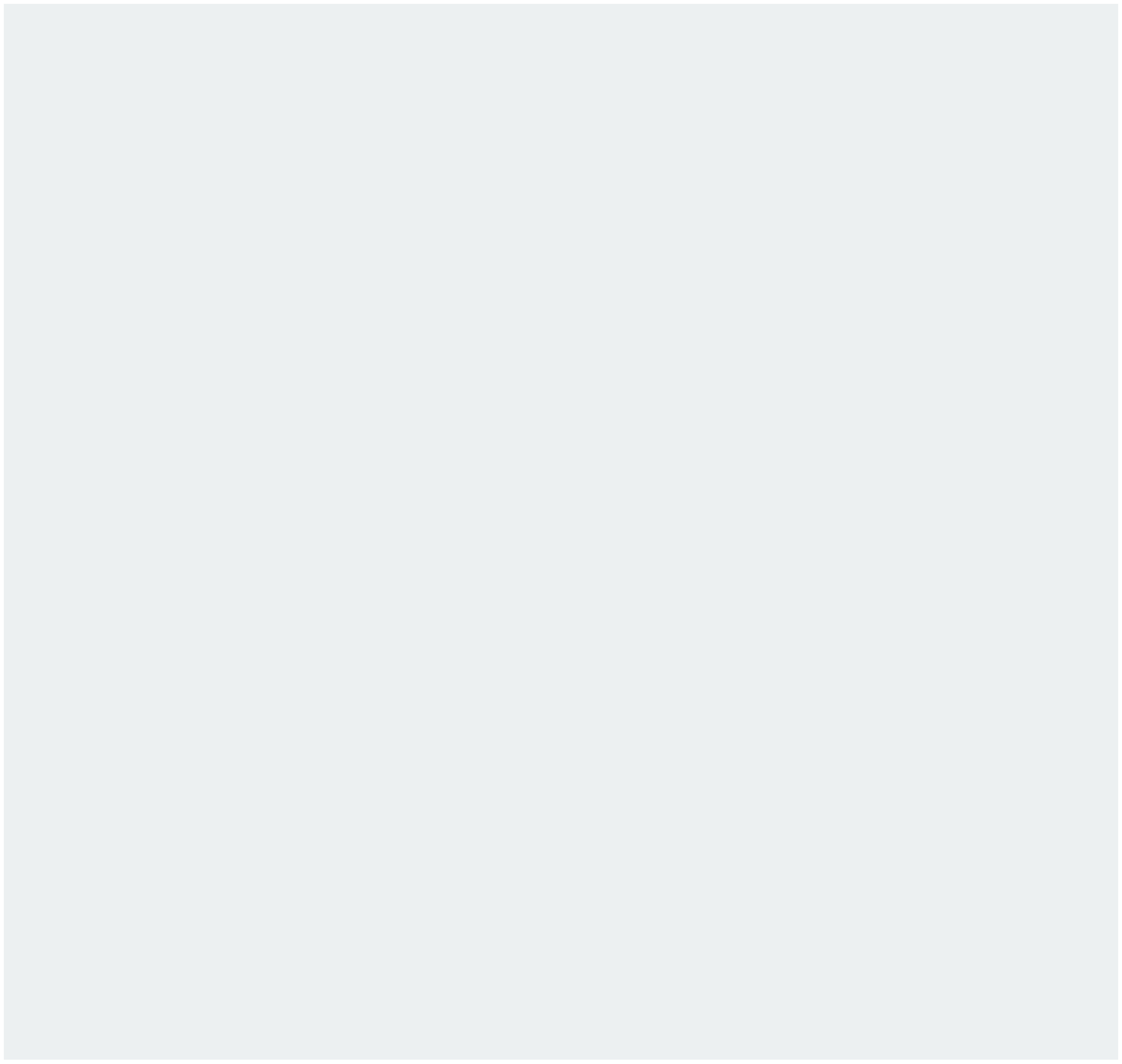
```
fetch('foo')
  .then(response => fetch('/articles/random'))
  .then(response => console.log(response.url))
// <- 'http://ponyfoo.com/articles/es6-symbols-in-depth'
```

Obviously, in the real world, your second `fetch` would probably depend on the response from the first one. Here's another example of returning a promise, where we randomly `fulfill` or `reject` after a second.

```
var p = Promise.resolve()
  .then(data => new Promise(function (resolve, reject) {
    setTimeout(Math.random() > 0.5 ? resolve : reject, 1000)
  }))

p.then(data => console.log('okay!'))
p.catch(data => console.log('boo!'))
```

The animation for this one is super fun!



Animation of the code shown right above

Okay it's **not** *that fun*. I did have fun making the Promisees tool itself!

Transforming Values in Promises

You’re not just limited to returning other promises from your `.then` and `.catch` callbacks. You could also return values, transforming what you had. The example below first creates a promise fulfilled with `[1, 2, 3]` and then has a fulfillment branch on top of that which maps thoes values into `[2, 4, 6]`. Calling `.then` on that branch of the promise will produce the doubled values.

```
Promise.resolve([1, 2, 3])
  .then(values => values.map(value => value * 2))
  .then(values => console.log(values))
  // <- [2, 4, 6]
```

Note that you can do the same thing in rejection branches. An interesting fact that may catch your eye is that if a `.catch` branch goes smoothly without errors, then it will be fulfilled with the returned value. That means that if you still want to have an error for that branch, you should `throw` again. The following piece of code takes an internal error and **masks it** behind a generic *“Internal Server Error”* message as to not leak off potentially dangerous information to its clients (visualization).

```
Promise.reject(new Error('Database ds.214.53.4.12 connection timeout!'))
  .catch(error => { throw new Error('Internal Server Error') })
  .catch(error => console.info(error))
  // <- Error: Internal Server Error
```

Mapping promise results is particularly useful when dealing with multiple concurrent promises. Let’s see how that looks like.

Leveraging Promise.all and Promise.race

A tremendously common scenario – even more so for those used to Node.js – is to have a dependency on things A and B before being able to do thing C. I'll proceed that lousy description of the scenario with multiple code snippets. Suppose you wanted to pull the homepage for both Google and Twitter, and then print out the length of each of their responses. Here's how that looks in the most näive approach possible, with a hypothetical `request(url, done)` method.

```
request('https://google.com', function (err, goog) {
  request('https://twitter.com', function (err, twit) {
    console.log(goog.length, twit.length)
  })
})
```

Of course, that's going to run in series you say! Why would we wait on Google's response before pulling Twitter's? The following piece fixes the problem. It's also ridiculously long, though, right?

```
var results = {}
request('https://google.com', function (err, goog) {
  results.goog = goog
  done()
})
request('https://twitter.com', function (err, twit) {
  results.twit = twit
  done()
})
function done () {
  if (Object.keys(results).length < 2) {
    return
  }
  console.log(results.goog.length, results.twit.length)
}
```

Since nobody wants to be writing code like that, utility libraries like `async` and `contra` make this much shorter for you. You can use `contra.concurrent` to run these methods at the same time and execute a callback once they all ended. Here's how that'd look like.

```
contra.concurrent({
  goog: function (next) {
    request('https://google.com', next)
  }
  twit: function (next) {
    request('https://twitter.com', next)
  }
}, function (err, results) {
  console.log(results.goog.length, results.twit.length)
})
```

For the very common *"I just want a method that appends that magical `next` parameter at the end"* use case, there's also `contra.curry` *(equivalent of `async.apply`)* to make the code even shorter.

```
contra.concurrent({
  goog: contra.curry(request, 'https://google.com'),
  twit: contra.curry(request, 'https://twitter.com')
}, function (err, results) {
  console.log(results.goog.length, results.twit.length)
})
```

Promises already make the "run this after this other thing in series" use case very easy, using `.then` as we saw in several examples earlier. For the *"run these things concurrently"* use case, we can use `Promise.all` (visualization here).

```
Promise.all([
  fetch('/'),
  fetch('foo')
])
  .then(responses => responses.map(response => response.statusText))
  .then(status => console.log(status.join(', ')))
  // <- 'OK, Not Found'
```

Note that even if a single dependency is rejected, the `Promise.all` method will be rejected entirely as well.

```
Promise.all([
  Promise.reject(),
  fetch('/'),
  fetch('foo')
])
  .then(responses => responses.map(response => response.statusText))
  .then(status => console.log(status.join(', ')))
  // nothing happens
```

In summary, `Promise.all` has two possible outcomes.

- Settle with *a single* rejection `reason` as soon as one of its dependencies is rejected
- Settle with *all* fulfillment `results` as soon as all of its dependencies are fulfilled

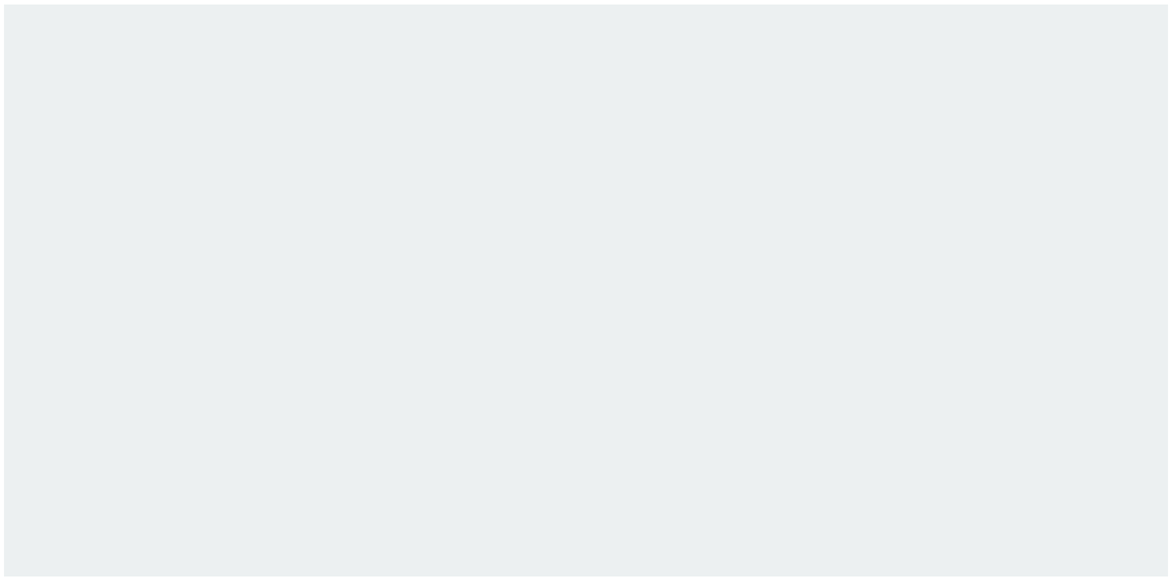
Then there's `Promise.race`. This is a similar method to `Promise.all`, except the first promise to settle will "win" the race, and its value will be passed along to branches of the race. If you run the visualization for the following piece of code a few times, you'll notice that this race doesn't have a clear winner. It depends on the server and the network!

```
Promise.race([
  fetch('/'),
  fetch('foo')
])
  .then(response => console.log(response.statusText))
  // <- 'OK', or maybe 'Not Found'.
```

Rejections will also finish the race, and the race promise will be rejected. As a closing note we may indicate that this could be useful for scenarios where we want to time out a promise we otherwise have no control over. For instance, the following race does make sense.

```
var p = Promise.race([
  fetch('/resource-that-may-take-a-while'),
  new Promise(function (resolve, reject) {
    setTimeout(() => reject(new Error('request timeout')), 5000)
  })
])
p.then(response => console.log(response))
p.catch(error => console.log(error))
```

To close this article, I'll leave you with a visualization. It shows the race between a resource and a timeout as shown in the code above.



Race between a resource and a timeout

Here’s hoping I didn’t make promises even harder to understand for you!