Promises

by Forbes Lindesay

Motivation

Consider the following synchronous JavaScript function to read a file and parse it as JSON. It is simple and easy to read, but you wouldn't want to use it in most applications as it is blocking. This means that while you are reading the file from disk (a slow operation) nothing else can happen.

```
function readJSONSync(filename) {
  return JSON.parse(fs.readFileSync(filename, 'utf8'));
}
```

To make our application performant and responsive, we need to make all the operations that involve IO be asynchronous. The simplest way to do this would be to use a callback. However, a naive implementation will probably go wrong:

```
function readJSON(filename, callback){
  fs.readFile(filename, 'utf8', function (err, res){
    if (err) return callback(err);
    callback(null, JSON.parse(res));
  });
}
```

- The extra `callback` parameter confuses our idea of what is input and what is the return value.
- It doesn't work at all with control flow primitives.
- It doesn't handle errors thrown by `JSON.parse`

We need to handle errors thrown by `JSON.parse` but we also need to be careful not to handle errors thrown by the `callback` function. By the time we've done all of this our code is a mess of error handling:

```
function readJSON(filename, callback){
  fs.readFile(filename, 'utf8', function (err, res){
    if (err) return callback(err);
    try {
      res = JSON.parse(res);
    } catch (ex) {
      return callback(ex);
    }
    callback(null, res);
  });
}
```

Despite all this mess of error handling code, we are still left with the problem of the extra `callback` parameter hanging around. Promises help you naturally handle errors, and write cleaner code by not having `callback` parameters, and without modifying the underlying architecture (i.e. you can implement (/implementing/) them in pure JavaScript and use them to wrap existing asynchronous operations).

What is a promise?

The core idea behind promises is that a promise represents the result of an asynchronous operation. A promise is in one of three different states:

- pending - The initial state of a promise.
- fulfilled - The state of a promise representing a successful operation.
- rejected - The state of a promise representing a failed operation.

Once a promise is fulfilled or rejected, it is immutable (i.e. it can never change again).

Constructing a promise

Once all of the APIs return promises, it should be relatively rare that you need to construct one by hand. In the meantime, we need a way to polyfill existing APIs. For example:

```
function readFile(filename, enc){
  return new Promise(function (fulfill, reject){
    fs.readFile(filename, enc, function (err, res){
      if (err) reject(err);
      else fulfill(res);
    });
  });
}
```

We use `new Promise` to construct the promise. We give the constructor a factory function which does the actual work. This function is called immediately with two arguments. The first argument fulfills the promise and the second argument rejects the promise. Once the operation has completed, we call the appropriate function.

Awaiting a promise

In order to use a promise, we must somehow be able to wait for it to be fulfilled or rejected. The way to do this is using `promise.done` (see warning at the end of this section if attempting to run these samples).

With this in mind, it's easy to re-write our earlier `readJSON` function to use promises:

```
function readJSON(filename){
  return new Promise(function (fulfill, reject){
    readFile(filename, 'utf8').done(function (res){
      try {
        fulfill(JSON.parse(res));
      } catch (ex) {
        reject(ex);
      }
    }, reject);
  });
}
```

This still has lots of error handling code (we'll see how we can improve on that in the next section) but it's a lot less error prone to write, and we no longer have a strange extra parameter.

> Non Standard
>
> Note that `promise.done` (used in the examples in this section) has not been standardised. It is supported by most major promise libraries though, and is useful both as a teaching aid and in production code. I recommend using it along with the following polyfill (minified (/polyfills/promise-done-7.0.4.min.js) / unminified (/polyfills/promise-done-7.0.4.js)): undefined

Transformation / Chaining

Following our example through, what we really want to do is transform the promise via another operation. In our case, this second operation is synchronous, but it might just as easily have been an asynchronous operation. Fortunately, promises have a (fully standardised, except jQuery (#jquery)) method for transforming promises and chaining operations.

Put simply, `.then` is to `.done` as `.map` is to `.forEach`. To put that another way, use `.then` whenever you're going to do something with the result (even if that's just waiting for it to finish) and use `.done` whenever you aren't planning on doing anything with the result.

Now we can re-write our original example as simply:

```
function readJSON(filename){
  return readFile(filename, 'utf8').then(function (res){
    return JSON.parse(res)
  })
}
```

Since `JSON.parse` is just a function, we could re-write this as:

```
function readJSON(filename){
  return readFile(filename, 'utf8').then(JSON.parse);
}
```

This is very close to the simple synchronous example we started out with.

Implementations / Polyfills

p Promises are useful both in node.js and the browser

jQuery

This feels like a good time to warn you that what jQuery calls a promise is in fact totally different to what everyone else calls a promise. jQuery's promises have a poorly thought out API that will likely just confuse you. Fortunately, instead of using jQuery's strange version of a promise, you can just convert it to a really simple standardised promise:

```
var jQueryPromise = $.ajax('/data.json');
var realPromise = Promise.resolve(jQueryPromise);
//now just use `realPromise` however you like.
```

Browser

Promises are currently only supported by a pretty small selection of browsers (see kangax compatibility tables (http://kangax.github.io/es5-compat-table/es6/#Promise)). The good news is that they're extremely easy to polyfill (minified (/polyfills/promise-7.0.4.min.js) / unminified (/polyfills/promise-7.0.4.js)):

```
<script src="https://www.promisejs.org/polyfills/promise-7.0.4.min.js"></script>
```

None of the browsers currently support `Promise.prototype.done` so if you want to use that feature, and you are not including the polyfill above, you must at least include this polyfill (minified (/polyfills/promise-done-7.0.4.min.js) / unminified (/polyfills/promise-done-7.0.4.js)):

```
<script src="https://www.promisejs.org/polyfills/promise-done-7.0.4.min.js"></script>
```

Node.js

It's generally not seen as good practice to polyfill things in node.js. Instead, you're better off just requiring the library wherever you need it.

To install promise (https://github.com/then/promise) run:

```
npm install promise --save
```

Then you can load it into a local variable using `require`

```
var Promise = require('promise');
```

The "promise" library also provides a couple of really useful extensions for interacting with node.js

```
var readFile = Promise.denodeify(require('fs').readFile);
// now `readFile` will return a promise rather than
// expecting a callback

function readJSON(filename, callback){
  // If a callback is provided, call it with error as the
  // first argument and result as the second argument,
  // then return `undefined`. If no callback is provided,
  // just return the promise.
  return readFile(filename, 'utf8')
    .then(JSON.parse)
    .nodeify(callback);
}
```

Further Reading

- Patterns (/patterns/) - patterns of promise use, introducing lots of helper methods that will save you time.
- MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) - The mozilla developer network has great documentation on promises.
- YouTube (https://www.youtube.com/watch?v=qbKWsbJ76-s) - A video of my JSConf.eu talk that discusses many of the same things as appear in this article.

API Reference → (/api/)