Extracted from:

Async JavaScript

Build More Responsive Apps with Less Code

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic Express

Async JavaScript

Build More Responsive Apps with Less Code

Trevor Burnham

edited by Jacquelyn Carter

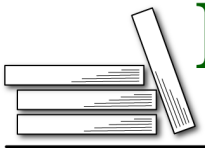
Async JavaScript

Build More Responsive Apps with Less Code

Trevor Burnham

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-27-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—November 2012

3.6 Binding to the Future with pipe

A big reason why performing a series of async tasks is often inconvenient in JavaScript is that you can't attach handlers to the second task until the first one is complete. As an example, let's GET data from one URL and then POST it to another.

```
var getPromise = $.get('/query');
getPromise.done(function(data) {
  var postPromise = $.post('/search', data);
});
// Now we'd like to attach handlers to postPromise...
```

Do you see what the problem is here? We can't bind callbacks to postPromise until our GET operation is done, because it doesn't exist yet! It's created by a $.post call that we can't make until we have the data that we're getting asynchronously from the $.get call.

That's why jQuery 1.6 added the pipe method to Promises. Essentially, pipe says this: "Give me a callback for this Promise, and I'll give you a Promise that represents the result of that callback."

```
var getPromise = $.get('/query');
var postPromise = getPromise.pipe(function(data) {
  return $.post('/search', data);
});
```

Looks like dark magic, right? Here's a breakdown: pipe takes one argument for each type of callback: done, fail, and progress. So, in this example, we just provided a callback that gets run when getPromise is resolved. The pipe method returns a new Promise that's resolved/rejected when the Promise returned from our callback is resolved/rejected.

Effectively, *pipe is a window into the future!*

You can also use pipe to "filter" a Promise by modifying callback arguments. If a pipe callback returns something other than a Promise/Deferred, then that value becomes the callback argument. For instance, if you have a Promise that emits progress notifications with a number between 0 and 1, you can use pipe to create an identical Promise that emits progress notifications with a human-readable string instead.

```
var promise2 = promise1.pipe(null, null, function(progress) {
  return Math.floor(progress * 100) + '% complete';
});
```

To summarize, there are two things you can do from a pipe callback.

- If you return a Promise, the Promise returned by pipe will mimic it.
- If you return a non-Promise value (or nothing), the Promise returned by pipe will immediately be resolved, rejected, or notified with that value, according to what just happened to the original Promise.

pipe's rule for whether something is a Promise is the same as $.when's: if it has a promise method, that method's return value is used as a Promise representing the original object. Again, promise.promise() === promise.

Pipe Cascading

pipe doesn't require you to provide every possible callback. In fact, you'll usually just want to write

```
var pipedPromise = originalPromise.pipe(successCallback);
```

or the following:

```
var pipedPromise = originalPromise.pipe(null, failCallback);
```

We've seen what happens when the original Promise succeeds in the first case, or fails in the second case, so that the piped Promise's behavior depends on the return value of successCallback or failCallback. But what about when we haven't given pipe a callback for what the original Promise does?

It's simple. The piped Promise mimics the original Promise in those cases. We can say that the original Promise's behavior *cascades* through the piped Promise. This cascading is very handy, because it allows us to define branching logic for async tasks with minimal effort. Suppose we have a three-step process.

```
var step1 = $.post('/step1', data1);
var step2 = step1.pipe(function() {
  return $.post('/step2', data2);
});
var lastStep = step2.pipe(function() {
  return $.post('/step3', data3);
});
```

Here, lastStep will resolve only if all three Ajax calls succeeded, and it'll be rejected if *any* of the three fail. If we care only about the process as a whole, we can omit the variable declarations for the earlier steps.

```
var posting = $.post('/step1', data1)
  .pipe(function() {
    return $.post('/step2', data2);
  })
    .pipe(function() {
      return $.post('/step3', data3);
    });
```

We could, equivalently, nest the second pipe inside of the other.

```
var posting = $.post('/step1', data1)
  .pipe(function() {
    return $.post('/step2', data2)
    .pipe(function() {
      return $.post('/step3', data3);
    });
  });
```

Of course, this brings us back to the Pyramid of Doom. You should be aware of this style, but as a rule, try to declare your piped Promises individually. The variable names may not be necessary, but they make the code far more self-documenting.

That concludes our tour of jQuery Promises. Now let's take a quick look at the major alternative: the CommonJS Promises/A specification and its flagship implementation, Q.js.

3.7 jQuery vs. Promises/A

In terms of capabilities, jQuery Promises and Promises/A are nearly identical. Q.js, the most popular Promises/A library, even offers methods that can work with jQuery Promises. The differences are superficial; they use the same words to mean different things.

As previously mentioned in Section 3.2, *Making Promises*, on page ?, jQuery uses the term *resolve* as the opposite of *fail*, whereas Promises/A uses *fulfill*.

Under Promises/A, a Promise is said to be "resolved" when it's either fulfilled or failed.

Up until the release of 1.8, jQuery's then method was just a shorthand for invoking done, fail, and progress simultaneously, while Promises/A's then acted more like jQuery's pipe. jQuery 1.8 corrected this by making then a synonym for pipe. However, any further reconciliation with Promises/A is unlikely because of backward compatibility concerns.

There are other, subtler differences as well. For instance, in Promises/A, whether a Promise returned by then is fulfilled or rejected depends on whether the invoked callback returns a value or throws an error. (Throwing errors from jQuery Promise callbacks is a bad idea because they'll go uncaught.)

Because of these issues, you should try to avoid interacting with multiple Promise implementations in the same project. If you're just getting Promises from jQuery methods, use jQuery Promises. If you're using another library that gives you CommonJS Promises, adopt Promises/A. Q.js makes it easy to "assimilate" jQuery Promises.

```
var qPromise = Q.when(jqPromise);
```

As long as these two standards remain divergent, this is the best way to make them play nice together. For more information, see the Q.js docs.[5]

3.8 Replacing Callbacks with Promises

In a perfect world, every function that started an async task would return a Promise. Unfortunately, most JavaScript APIs (including the native functions available in all browsers and in Node.js) are callback-based, not Promise-based. In this section, we'll see how Promises can be used with callback-based APIs.

The most straightforward way to use Promises with a callback-based API is to create a Deferred and pass its trigger function(s) as the callback argument(s). For example, with a simple async function like setTimeout, we'd pass our Deferred's resolve method.

```
var timing = new $.Deferred();
setTimeout(timing.resolve, 500);
```

In cases where an error could occur, we'd write a callback that conditionally routes to either resolve or reject. For example, here's how we'd work with a Node-style callback:

5. https://github.com/kriskowal/q

```
var fileReading = new $.Deferred();
fs.readFile(filename, 'utf8', function(err) {
  if (err) {
    fileReading.reject(err);
  } else {
    fileReading.resolve(Array.prototype.slice.call(arguments, 1));
  };
});
```

(Yes, you can use jQuery from Node. Just `npm install jquery` and use it like any other module. There's also a self-contained implementation of jQuery-style Promises, simply called Standalone Deferred.[6])

Writing this out routinely would be a drag, so why not make a utility function to generate a Node-style callback from any given Deferred?

```
deferredCallback = function(deferred) {
  return function(err) {
    if (err) {
      deferred.reject(err);
    } else {
      deferred.resolve(Array.prototype.slice.call(arguments, 1));
    };
  };
}
```

With that, we can write the previous example as follows:

```
var fileReading = new $.Deferred();
fs.readFile(filename, 'utf8', deferredCallback(fileReading));
```

In Q.js, Deferreds come with a `node` method for this right out of the box.

```
var fileReading = Q.defer();
fs.readFile(filename, 'utf8', fileReading.node());
```

As Promises become more popular, more and more JavaScript libraries will follow jQuery's lead and return Promises from their async functions. Until then, it takes only a few lines of code to turn any async function you want to use into a Promise generator.

3.9 What We've Learned

In my opinion, Promises are one of the most exciting features to be added to jQuery in years. Not only are they a big help in smoothing out the callback spaghetti that characterizes typical Ajax-rich apps, but they also make it much easier to coordinate async tasks of all kinds.

6. https://github.com/Mumakil/Standalone-Deferred

Using Promises takes some practice, especially when using `pipe`, but it's a habit well worth developing. You'll be peering into the future of JavaScript. The more APIs return Promises, the more compelling they become.

Microsoft has announced that Windows 8's Metro environment will have a Promise-based JavaScript API.[7] Where hipster developers and Microsoft both go, the rest of the world is bound to follow.

7. http://msdn.microsoft.com/en-us/library/windows/apps/br211867.aspx