You're Missing the Point of Promises

Oct 14, 2012 | 33 Comments | Tweet | posted in promises

This post originally appeared as a gist. Since then, the development of Promises/A+ has made its emphasis on the Promises/A spec seem somewhat outdated.

Contrary to some mistaken statements on the internet, the problems with jQuery's promises explained here are not fixed in recent versions; as of 2.1 beta 1 they have all the same problems outlined here, and according to one jQuery core team member, they will forever remain broken, in the name of backward compatibility.

Promises are a software abstraction that makes working with asynchronous operations much more pleasant. In the most basic definition, your code will move from continuation-passing style:

```
getTweetsFor("domenic", function (err, results) {
  // the rest of your code goes here.
});
```

to one where your functions return a value, called a *promise*, which represents the eventual results of that operation.

```
var promiseForTweets = getTweetsFor("domenic");
```

This is powerful since you can now treat these promises as first-class objects, passing them around, aggregating them, and so on, instead of inserting dummy callbacks that tie together other callbacks in order to do the same.

I've talked about how cool I think promises are at length. This essay isn't about that. Instead, it's about a disturbing trend I am seeing in recent JavaScript libraries that have added promise support: *they completely miss the point of promises*.

Thenables and CommonJS Promises/A

When someone says "promise" in a JavaScript context, usually they mean—or at least *think* they mean—CommonJS Promises/A. This is one of the smallest "specs" I've seen. The meat of

it is entirely about specifying the behavior of a single function, `then`:

> *A promise is defined as an object that has a function as the value for the property `then`:*
>
> `then(fulfilledHandler, errorHandler, progressHandler)`
>
> *Adds a `fulfilledHandler`, `errorHandler`, and `progressHandler` to be called for completion of a promise. The `fulfilledHandler` is called when the promise is fulfilled. The `errorHandler` is called when a promise fails. The `progressHandler` is called for progress events. All arguments are optional and non-function values are ignored. The `progressHandler` is not only an optional argument, but progress events are purely optional. Promise implementors are not required to ever call a `progressHandler` (the `progressHandler` may be ignored), this parameter exists so that implementors may call it if they have progress events to report.*
>
> *This function should return a new promise that is fulfilled when the given `fulfilledHandler` or `errorHandler` callback is finished. This allows promise operations to be chained together. The value returned from the callback handler is the fulfillment value for the returned promise. If the callback throws an error, the returned promise will be moved to failed state.*

People mostly understand the first paragraph. It boils down to *callback aggregation*. You use `then` to attach callbacks to a promise, whether for success or for errors (or even progress). When the promise transitions state—which is out of scope of this very small spec!—your callbacks will be called. This is pretty useful, I guess.

What people don't seem to notice is the second paragraph. Which is a shame, since it's the most important one.

What Is the Point of Promises?

The thing is, promises are not *about* callback aggregation. That's a simple utility. Promises are about something much deeper, namely providing a direct correspondence between synchronous functions and asynchronous functions.

What does this mean? Well, there are two very important aspects of synchronous functions:

- They *return values*
- They *throw exceptions*

Both of these are essentially about composition. That is, you can feed the return value of one function straight into another, and keep doing this indefinitely. *More importantly*, if at any point that process fails, one function in the composition chain can throw an exception, which then

bypasses all further compositional layers until it comes into the hands of someone who can handle it with a `catch`.

Now, in an asynchronous world, you can no longer return values: they simply aren't ready in time. Similarly, you can't throw exceptions, because nobody's there to catch them. So we descend into the so-called "callback hell," where composition of return values involves nested callbacks, and composition of errors involves passing them up the chain manually, and oh by the way you'd better *never* throw an exception or else you'll need to introduce something crazy like domains.

The point of promises is to give us back functional composition and error bubbling in the async world. They do this by saying that your functions should return a promise, which can do one of two things:

- Become *fulfilled by a value*
- Become *rejected with an exception*

And, *if* you have a correctly implemented `then` function that follows Promises/A, then fulfillment and rejection will compose just like their synchronous counterparts, with fulfillments flowing up a compositional chain, but being interrupted at any time by a rejection that is only handled by someone who declares they are ready to handle it.

In other words, the following asynchronous code:

```
getTweetsFor("domenic") // promise-returning function
  .then(function (tweets) {
    var shortUrls = parseTweetsForUrls(tweets);
    var mostRecentShortUrl = shortUrls[0];
    return expandUrlUsingTwitterApi(mostRecentShortUrl); // promise-returning fur
  })
  .then(httpGet) // promise-returning function
  .then(
    function (responseBody) {
      console.log("Most recent link text:", responseBody);
    },
    function (error) {
      console.error("Error with the twitterverse:", error);
    }
  );
```

parallels* the synchronous code:

```
try {
  var tweets = getTweetsFor("domenic"); // blocking
  var shortUrls = parseTweetsForUrls(tweets);
  var mostRecentShortUrl = shortUrls[0];
  var responseBody = httpGet(expandUrlUsingTwitterApi(mostRecentShortUrl)); // bl
  console.log("Most recent link text:", responseBody);
} catch (error) {
  console.error("Error with the twitterverse: ", error);
}
```

Note in particular how errors flowed from any step in the process to our `catch` handler, without explicit by-hand bubbling code. And with the upcoming ECMAScript 6 revision of JavaScript, plus some party tricks, the code becomes not only parallel but almost identical.

That Second Paragraph

All of this is essentially enabled by that second paragraph:

> *This function should return a new promise that is fulfilled when the given `fulfilledHandler` or `errorHandler` callback is finished. This allows promise operations to be chained together. The value returned from the callback handler is the fulfillment value for the returned promise. If the callback throws an error, the returned promise will be moved to failed state.*

In other words, `then` is *not* a mechanism for attaching callbacks to an aggregate collection. It's a mechanism for *applying a transformation* to a promise, and yielding a *new* promise from that transformation.

This explains the crucial first phrase: "this function should return a new promise." Libraries like jQuery (before 1.8) don't do this: they simply mutate the state of the existing promise. That means if you give a promise out to multiple consumers, they can interfere with its state. To realize how ridiculous that is, consider the synchronous parallel: if you gave out a function's return value to two people, and one of them could somehow change it into a thrown exception! Indeed, Promises/A points this out explicitly:

> *Once a promise is fulfilled or failed, the promise's value MUST not be changed, just as a values in JavaScript, primitives and object identities, can not change (although objects themselves may always be mutable even if their identity isn't).*

Now consider the last two sentences. They inform how this new promise is created. In short:

- If either handler returns a value, the new promise is fulfilled with that value.

- If either handler throws an exception, the new promise is rejected with that exception.

This breaks down into four scenarios, depending on the state of the promise. Here we give their synchronous parallels so you can see why it's crucially important to have semantics for all four:

1. Fulfilled, fulfillment handler returns a value: simple functional transformation
2. Fulfilled, fulfillment handler throws an exception: getting data, and throwing an exception in response to it
3. Rejected, rejection handler returns a value: a `catch` clause got the error and handled it
4. Rejected, rejection handler throws an exception: a `catch` clause got the error and re-threw it (or a new one)

Without these transformations being applied, you lose all the power of the synchronous/asynchronous parallel, and your so-called "promises" become simple callback aggregators. This is the problem with jQuery's current "promises": they only support scenario 1 above, omitting entirely support for scenarios 2–4. This was also the problem with Node.js 0.1's `EventEmitter`-based "promises" (which weren't even `then`able).

Furthermore, note that by catching exceptions and transforming them into rejections, we take care of both intentional and unintentional exceptions, just like in sync code. That is, if you write `aFunctionThatDoesNotExist()` in either handler, your promise becomes rejected and that error will bubble up the chain to the nearest rejection handler just as if you had written `throw new Error("bad data")`. Look ma, no domains!

So What?

Maybe you're breathlessly taken by my inexorable logic and explanatory powers. More likely, you're asking yourself why this guy is raging so hard over some poorly-behaved libraries.

Here's the problem:

> *A promise is defined as an object that has a function as the value for the property* `then`

As authors of Promises/A-consuming libraries, we would like to assume this statement to be true: that something that is "thenable" actually behaves as a Promises/A promise, with all the power that entails.

If you can make this assumption, you can write very extensive libraries that are entirely agnostic to the implementation of the promises they accept! Whether they be from Q, when.js, or even WinJS, you can use the simple composition rules of the Promises/A spec to build on promise behavior. For example, here's a generalized retry function that works with any Promises/A implementation.

Unfortunately, libraries like jQuery break this. This necessitates ugly hacks to detect the presence of objects masquerading as promises, and who call themselves in their API documentation promises, but aren't really Promises/A promises. If the consumers of your API start trying to pass you jQuery promises, you have two choices: fail in mysterious and hard-to-decipher ways when your compositional techniques fail, or fail up-front and block them from using your library entirely. This sucks.

The Way Forward

So this is why I want to avoid an unfortunate callback aggregator solution ending up in Ember. That's why I wrote this essay. And that's why, in the hours following writing the original version of this essay, I worked up a general Promises/A compliance suite that we can all use to get on the same page in the future.

Since the release of that test suite, great progress has been made in promise interoperability and understanding. One library, rsvp.js, was released with the explicit goal of providing these features of Promises/A. Others followed suit. But the most exciting result was the formation of the Promises/A+ organization, a loose coalition of implementors who have produced the Promises/A+ specification extending and clarifying the prose of the original Promises/A spec into something unambiguous and well-tested.

There's still work to be done, of course. Notably, at current time of writing, the latest jQuery version is 1.9.1, and its promises implementation is completely broken with regard to the error handling semantics. Hopefully, with the above explanation to set the stage and the Promises/A+ spec and test suite in place, this problem can be corrected in jQuery 2.0.

In the meantime, here are the libraries that conform to Promises/A+, and that I can thus unreservedly recommend:

- Q by Kris Kowal and myself: a full-featured promise library with a large, powerful API surface, adapters for Node.js, progress support, and preliminary support for long stack traces.
- RSVP.js by Yehuda Katz: a very small and lightweight, but still fully compliant, promise library.
- when.js by Brian Cavalier: an intermediate library with utilities for managing collections of eventual tasks, as well as support for both progress and cancellation.

If you are stuck with a crippled "promise" from a source like jQuery, I recommend using one of the above libraries' assimilation utilities (usually under the name `when`) to convert to a real promise as soon as possible. For example:

```
var promise = Q.when($.get("https://github.com/kriskowal/q"));
// aaaah, much better
```

Comments

33 Comments Hidden Variables 1 Login

Recommend 20 Share Sort by Best

Join the discussion…

larry • 4 years ago
Just wanted to say that I truly enjoyed your writing. You are one of the best out there. And I can tell you've been around a while (as have I) by your "Look ma...!" reference. An oldie but a goodie. Anyway, I think you make excellent points, and I see that jQuery's Deferred now has a promise() method that returns an immutable Deferred, I wonder if he read your post. Keep up the great writing, it's clear, it's smart, and it teaches.
15 • Reply • Share ›

willlma larry • 3 years ago
As a side-note. Hands in French is mains, so, intentional or not, "no domains" is punny.
3 • Reply • Share ›

regardless larry • a year ago
According to the docs, deferred.promise() was added in jQuery 1.5.
https://api.jquery.com/deferre...
• Reply • Share ›

bergus • 4 years ago
In my opinion you didn't got the point of promises either. Of course, it is about functional composition in an asynchronous environment. But it is not limited to a mechanism for "catching" the asynchronous counterpart of "thrown exceptions", it can model any control flow like if-else statements as it represents a task/computation with an error state. That can include an async GUI-`confirm` mechanism, which is in no way an Exception when rejected. Yet, your very nice `retry`-function would apply to it as well.
7 • Reply • Share ›

Guest • 3 years ago
I think you actually misunderstand that spec. The spec defines promise as an an object whose "then" property maps to a function. The function that should return the promise is the "then" function, not any of your attached callbacks. Whether or not the spec further defines what your callbacks should return is another discussion but I seriously think this is much ado about

nothing; albeit an interesting concept, it's not what they meant which is why jQuery implemented theirs that way, correctly.

4 ▲ ▼ • Reply • Share ›

Domenic Denicola Mod → Guest • 2 years ago

As the author of the spec (both Promises/A+ and ES6), you seem to have misread them :)

9 ▲ ▼ • Reply • Share ›

Peter L → Domenic Denicola • 5 months ago

LOL

1 ▲ ▼ • Reply • Share ›

Devin → Guest • 2 years ago

I disagree, The point of the then (as spec'ed) is to allow mutation of the promise chain. As implemented like you suggest that is not the case. This is why every promise library I know of wraps thenables to protect you from the whack-a-do world of jQuery and into the sane world of promises.

▲ ▼ • Reply • Share ›

wizardwerdna • 3 years ago

I thought I'd join in with the kudos, Dom. This missive has now become the iconic explanation of why you did what you did, and where will be the future of promises. It is also a simple, clear statement of the virtues of promises for developers. Your leadership has had a fundamental impact on the promises community, and ultimately on the upcoming DOM Futures standards. Keep up the great work.

3 ▲ ▼ • Reply • Share ›

obiwanginobli • 3 years ago

"More likely, you're asking yourself why this guy is raging so hard over some poorly-behaved libraries"

i LOL'd

3 ▲ ▼ • Reply • Share ›

jotabe net • a year ago

Finally jQuery has decided to implement Promises/A+ specs. I've made a little test with 3.0 alpha1, and the chaining, and the exception handling works fine. Later on, I've found an official quote, in this blog post: http://blog.jquery.com/2015/07... where you can find a heading qhich reads "jQuery.Deferred is now Promises/A+ compatible".

2 ▲ ▼ • Reply • Share ›

Ben Nadel • 2 years ago

Outstanding article. I love promises; but, find them to get quite heady. Every time I use them for something complex, I have to sort of remind myself how they work. I liked all your arguments. It

makes sense to me.

2 ^ v • Reply • Share ›

joelmoss • 3 years ago

We discovered a real problem with Promises this week, which we think makes jQuery the best option. Would love to know what you think https://codio.com/s/blog/2013/...

2 ^ v • Reply • Share ›

cweekly ➚ joelmoss • 3 years ago

TLDR answer for that codio post and its discussion:
use of Q and other good promises libs' "done" (vs "then") solves the problem.

5 ^ v • Reply • Share ›

gabalafou ➚ joelmoss • 2 years ago

Hey **@Domenic Denicola** -- I saw that you commented on this thread a couple of days ago. I realize this blog post is pretty old (2012), so if you or somebody you trust has already answered the question I'm about to ask, if you'd be willing to just link me to a discussion, I would greatly appreciate it.

I've already read to some extent conversations on jQuery message boards about why jQuery's authors chose not to handle uncaught exceptions in promise callbacks and bubble them up the promise chain. But as of yet, I still haven't seen anyone propose an alternative for "real-world" development.

You see, I think **@joelmoss** brings up a really good point. As someone who has used promises extensively in production code deployed across millions of web sites (I work for Disqus), I can say that I have run into the very same problem described in the Codio blog post and it's been a very real problem, and while I am absolutely seduced by your inexorable logic—and the isomorphism between blocking/synchronous code and spec-compliant promise implementations—I still find myself unable to get on board with "swallowing" uncaught exceptions -- i.e.:

```
    var deferred = Q.defer();
```

see more

^ v • Reply • Share ›

gabalafou ➚ gabalafou • 2 years ago

Haha, well, right after writing this I realized I should have tried using native Promises in my browsers. What I found was interesting. Given this code:

```
new Promise(function (resolve) { // or new ES6Promise.Promise if using polyfi
    resolve()
}).then(function () {
    console.log('Will the browser catch the following exception?'); // the an
    throw 'snafu';
});
```

Firefox v35 was able to find the uncaught exception. So was Chrome v41, although it wasn't as helpful in the console log as it usually is finding the line number. But Safari v8 did not catch the error.

And as far as I can tell, Jake Archibald's ES6 polyfill (ES6Promise.Promise) does not let you know about the uncaught exception.

Reply • Share ›

Blake Miner ➚ gabalafou • a year ago

@gabalafou - This is technically an unhandled rejection, not an uncaught exception. Good grief! There's all sort of debate right now about how to handle unhandled rejections: https://github.com/promises-ap...

Reply • Share ›

Cheers • a year ago

Great stuff. I remember reading http://tom.preston-werner.com/... and going aha about Git. This article is something similar for promises. I have been searching for something like this on promises for long. Most people talk about the syntax and things like that (same for Git). Its more subtle in the case of Promises, the main goal is to make the code flow natural. Otherwise callbacks do everything you need. Promise is a design pattern and not a new feature isn't it?

1 • Reply • Share ›

Sébastien Lorber • 3 years ago

Nice blog

As a Scala developer this seems quite obvious that the mutable JQuery promise is a bad idea in the first place. It is like having the "map" operator without the "flatMap" which permits to chain the promise calls, ie avoiding to nest promises. This is quite usual when using monads.

Check this: http://stackoverflow.com/quest...

Domenic what I don't see in your post (maybe it is out of scope) is how a library developer is supposed to deal with promises if he wants to offer a promise-based library?
Because it seems the spec is pretty small so this means that as a lib developer I can't use in my lib advanced features of Q without coupling the lib to the promise implementor.

Is there the minimum promise implementation that fulfill the spec so that we can run the tests of the library against this minimal implementation? In order to avoid having a dependency to a specific implementation? Maybe you can answer here: http://stackoverflow.com/quest...

1 • Reply • Share ›

Domenic Denicola Mod ➚ Sébastien Lorber • 3 years ago

Indeed, that's a common and good question. The medium-term in answer is that, now that promises are in the JavaScript language (and shipping in Chrome and Firefox!), you should be able to just use those. In the short term, you can use any one of the many compliant implementations, or even allow pluggable promise constructors, e.g.

```
myLibrary.setPromiseConstructor(Q.Promise).
```

2 • Reply • Share ›

Fagner Brack • 8 months ago

Late to the party but I guess this can be useful for someone: https://medium.com/@fagnerbrac...

• Reply • Share ›

Rodrigo Rodriguez • a year ago

In the non-asynchrony version, why is shortUrls is being returned as a value instead of a Promise, even though it is not within the asynchrony scope?

• Reply • Share ›

nottRobin ➚ Rodrigo Rodriguez • a year ago

I think the non-async version basically amounts to pseudo-code. He's just saying "if this were a synchronous operation, the code might looks like this - see how closely the new way of writing async with promises resembles it."

• Reply • Share ›

nadimtuhin • a year ago

Great article

• Reply • Share ›

onstottj • 2 years ago

Does jQuery 1.8 follow the spec more closely? See http://api.jquery.com/deferred.... I could be wrong.

• Reply • Share ›

Domenic Denicola Mod ➚ onstottj • 2 years ago

No; see the note at the top of the post, in italics.

• Reply • Share ›

GrandTurion ➚ Domenic Denicola • a year ago

As per the note in italics, you should amend that phrase from the article: "This explains the crucial first phrase: “this function should return a new promise.” Libraries like jQuery (before 1.8) don’t do this: "

• Reply • Share ›

danjah ➚ GrandTurion • 5 months ago

That was what I thought, but then thought otherwise, better people continue reading and learn, than skip the lesson because they already know that their version of jQuery is <1.8.

• Reply • Share ›

onstotti ➚ Domenic Denicola • 2 years ago

Ah I see (I should learn to read). That's too bad.

Reply • Share ›

Alex • 3 years ago

Great explanations here. Thanks.

Reply • Share ›

Justin Abrahms • 3 years ago

Thanks for this. I was having some difficulty with a promises API for webdriver (specifically how to construct the page object pattern) but this triggered some ideas and I think it'll work now. :)

Reply • Share ›

Riley Eynon-Lynch • 3 years ago

Thanks for the post! I came across this while I was putting together my own writeup of promises, and it clarified a lot for me. I especially liked your observation that promises give us back functional composition - I quoted you. If you have time to check it out I'd love any feedback you have! http://larkolicio.us/blog/?p=1...

Thanks again!

Reply • Share ›

Alex Mills • a year ago

Or we could just use events and return an event emitter. it's all the same soup. personally I like error-first callbacks and the async library more than anything else. that's my opinion and I am sticking to it.

Furthermore, most libraries that return callbacks have hidden the try/catch from you. For example:

```
function doSomeRealWork(input, callback){

var foo = null;
try{
foo = input.parse();
}
catch(err){
return callback(err);
}
return foo;
}
```

so I don't see the point of promises, sorry, my loss I guess :)

Reply • Share ›

ALSO ON **HIDDEN VARIABLES**

Reading From Files ES6 Iterators, Generators, and Iterables

Reading From Files

14 comments • 2 years ago•

Mudit — Yep, you have indeed just reinvented a (primitive, probably buggy) version of the GC algorithm. So you've lost

ES6 Iterators, Generators, and Iterables

10 comments • 2 years ago•

Domenic Denicola — For anyone who comes along later and wonders what this was referring to, there used to be a neat trick for

---

Hidden Variables
Source at
domenic/blog.domenic.me

domenic
domenic

Copyright © 2016 Domenic Denicola
d@domenic.me