# JavaScript Promises Are Cool

**Keyhole Software**, 30 Jul 2014 CPOL

Rate:

5.00 (1 vote)

"And when I promise something, I never ever break that promise. Never." — Rapunzel Many languages have libraries of interesting schemes called promises, deferreds, or futures. Those help to tame the wild asynchronous into something more like the mundane sequential.



"And when I promise something, I never ever break that promise. Never." — Rapunzel

Many languages have libraries of interesting schemes called promises, deferreds, or futures. Those help to tame the wild asynchronous into something more like the mundane sequential. JavaScript promises can promote separation of concerns in lieu of tightly coupled interfaces. This article is about JavaScript promises of the Promises/A variety. [http://wiki.commonjs.org/wiki/Promises/A]

Promise use cases:

- Executing rules
- Multiple remote validations
- Handling timeouts
- Remote data requests
- Animation
- Decoupling event logic from app logic
- Eliminating callback triangle of doom
- Controlling parallel asynchronous operations

A JavaScript *promise* is an I.O.U. to return a value in the future. It is a data object having a well-defined behavior. A promise has one of three possible states:

1. Pending
2. Rejected
3. Resolved

A rejected or resolved promise is *settled*. A promise state can only move from pending to settled. Thereafter its state is immutable. A promise can be held long after its associated action has settled. At leisure, we can extract the result multiple times. We carry this out by calling *promise.then()*. That call won't return unless, or until, the associated action has been settled. We can fluidly chain promises. The chained "then" functions should each either return a promise, or let the original promise be the return value.

Through this paradigm we can write asynchronous code more as if it were synchronous code. The power lies in composing promise tasks:

- **Stacked** tasks: multiple *thens* scattered in the code, but on same promise.

- **Parallel** tasks: multiple promises return a single promise.
- **Sequential** tasks: *promise … then … promise*
- **Combinations** of these.

Why this extra layer? Why can't we just use raw callbacks?

# Problems with callbacks

Callbacks are fine for reacting to simple recurring events such as enabling a form value based on a click, or for storing the result of a REST call. Callbacks also entice one to code in a chain by having one callback make the next REST call, in turn supplying a callback that makes the next REST call, and so forth. This tends toward a *pyramid of doom* shown in Figure 1. There, code grows horizontally faster than it grows vertically. Callbacks seem simple … until we need a result, **right now**, to use in the next step of our code.

Figure 1: Pyramid of Doom anti-pattern + below code

Hide Copy Code

```
'use strict';
var i = 0;
function log(data) {console.log('%d %s', ++i, data); };

function validate() {
   log("Wait for it ...");
   // Sequence of four Long-running async activities
   setTimeout(function () {
      log('result first');
      setTimeout(function () {
         log('result second');
         setTimeout(function () {
            log('result third');
            setTimeout(function () {
               log('result fourth')
            }, 1000);
         }, 1000);
      }, 1000);
   }, 1000);

};
validate();
```

In Figure 1, I've used timeouts to mock asynchronous actions. The notion of managing exceptions there that could play controllably with downstream actions is painful. When we have to compose callbacks, then code organization becomes messy. Figure 2 shows a

mock validation flow that will run, when pasted into a NodeJS REPL. We will migrate it from the pyramid-of-doom pattern to a sequential promise rendition in the next section.

Hide Copy Code

```
'use strict';
var i = 0;
function log(data) {console.log('%d %s', ++i, data); };

// Asynchronous fn executes a callback result fn
function async(arg, callBack) {
   setTimeout(function(){
      log('result ' + arg);
      callBack();
   }, 1000);
};

function validate() {
   log("Wait for it ...");
   // Sequence of four Long-running async activities
   async('first', function () {
      async('second',function () {
         async('third', function () {
            async('fourth', function () {});
         });
      });
   });
};
validate();
```

Execution in a NodeJS REPL yields:

Hide Copy Code

```
$ node scripts/examp2b.js
1 Wait for it ...
2 result first
3 result second
4 result third
5 result fourth
$
```

I once had a dynamic validation situation in AngularJS where form values could be dynamically mandatory, depending on peer form values. A REST service ruled on the valid value of each mandatory item. I avoided nested callbacks by writing a dispatcher that operated on a stack of functions based on which values were required. The dispatcher would pop a function from the stack and execute it. That function's callback would finish by calling my dispatcher to repeat until the stack emptied. Each callback recorded any validation error returned from its remote validation call.

I consider my contraption to be an anti-pattern. Had I used the promise alternative offered by Angular's $http call, my thinking for the entire validation would have resembled a linear form resembling synchronous programming. A flattened promise chain is readable. Read on ...

# Using Promises

Figure 3 shows my contrived validation recast into a promise chain. It uses the kew promise library. The Q library works equally well. To try it, first use npm to import the kew library into NodeJS, and then load the code into the NodeJS REPL.

Hide Shrink ▲ Copy Code

```
'use strict';
var Q = require('kew');
var i = 0;

function log(data) {console.log('%d %s', ++i, data); };

// Asynchronous fn returns a promise
function async(arg) {
    var deferred = Q.defer();
    setTimeout(function () {
        deferred.resolve('result ' + arg);\
    }, 1000);
    return deferred.promise;
```

```
};

// Flattened promise chain
function validate() {
    log("Wait for it ...");
    async('first').then(function(resp){
        log(resp);
        return async('second');
    })
    .then(function(resp){
        log(resp);
        return async('third')
    })
    .then(function(resp){
        log(resp);
        return async('fourth');
    })
    .then(function(resp){
        log(resp);
    }).fail(log);
};
validate();
```

The output is the same as that of the nested callbacks:

Hide Copy Code

```
$ node scripts/examp2-pflat.js
1 Wait for it ...
2 result first
3 result second
4 result third
5 result fourth
$
```

The code is slightly "taller," but I think it is easier to understand and modify. Adding rational error handling is easier. The fail call at the end of the chain catches errors within the chain, but I could have also supplied a reject handler at any *then* to deal with a rejection in its action.

# Server or Browser

Promises are useful in a browser as well as in a NodeJS server. The following URL, http://jsfiddle.net/mauget/DnQDx/, points to a JSFiddle that shows how to use a single promise in a web page. All the code is modifiable in the JSFiddle. One variation of the browser output is in Figure 4. I rigged the action to randomly reject. You could try it several times to get an opposite result. It would be straightforward to extend it to a multiple promise chain, as in the previous NodeJS example.

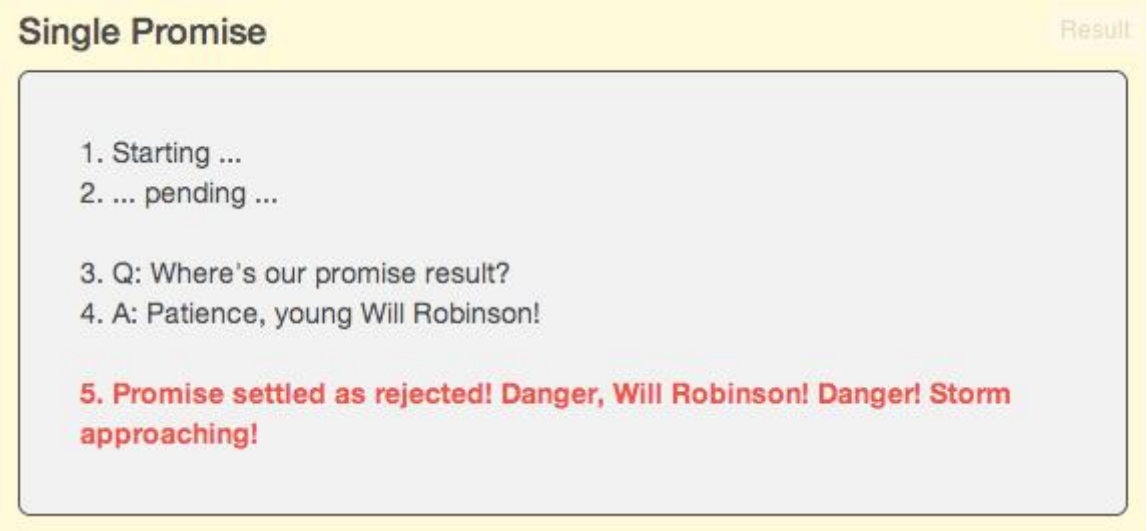


Figure 4: Single promise

# Parallel Promises

Consider an asynchronous operation feeding another asynchronous action. Let the latter consist of three parallel asynchronous actions that, in turn, feed a final action. It settles only when all parallel child requests settle. See Figure 5. This is inspired from a favorable encounter with a chain of twelve MongoDB operations. Some were eligible to operate parallel. I implemented the flow with promises.

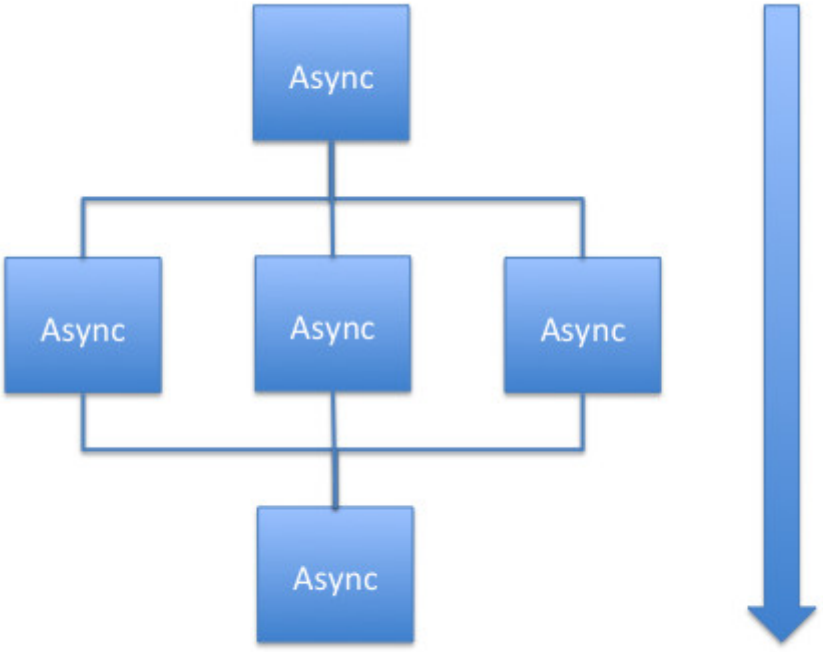


Figure 5: Composition of asynchronous actions

How would we model those parallel promises at the center row of that diagram? The key is that most promise libraries have an *all* function that produces a parent promise of child promises held in an array. When all the child promises resolve, the parent promise resolves. If one of the child promises rejects, the parent promise rejects.

Figure 6 shows a code fragment that makes ten literals into a promise of ten parallel promises. The *then* at the end completes only when all ten children resolve or if any child rejects.

Hide Copy Code

```
var promiseVals = ['To ', 'be, ', 'or ',
    'not ', 'to ', 'be, ', 'that ',
    'is ', 'the ', 'question.'];

var startParallelActions = function (){
    var promises = [];

    // Make an asynchronous action from each literal
    promiseVals.forEach(function(value){
        promises.push(makeAPromise(value));
    });

    // Consolidate all promises into a promise of promises
    return Q.all(promises);
};

startParallelActions ().then( . . .
```

The following URL, http://jsfiddle.net/mauget/XKCy2/, targets a JSFiddle that runs 10 parallel promises in a browser, rejecting or resolving at random. The complete code is there for inspection and what-if changes. Rerun until you get an opposite completion. Figure 7 shows the positive result.

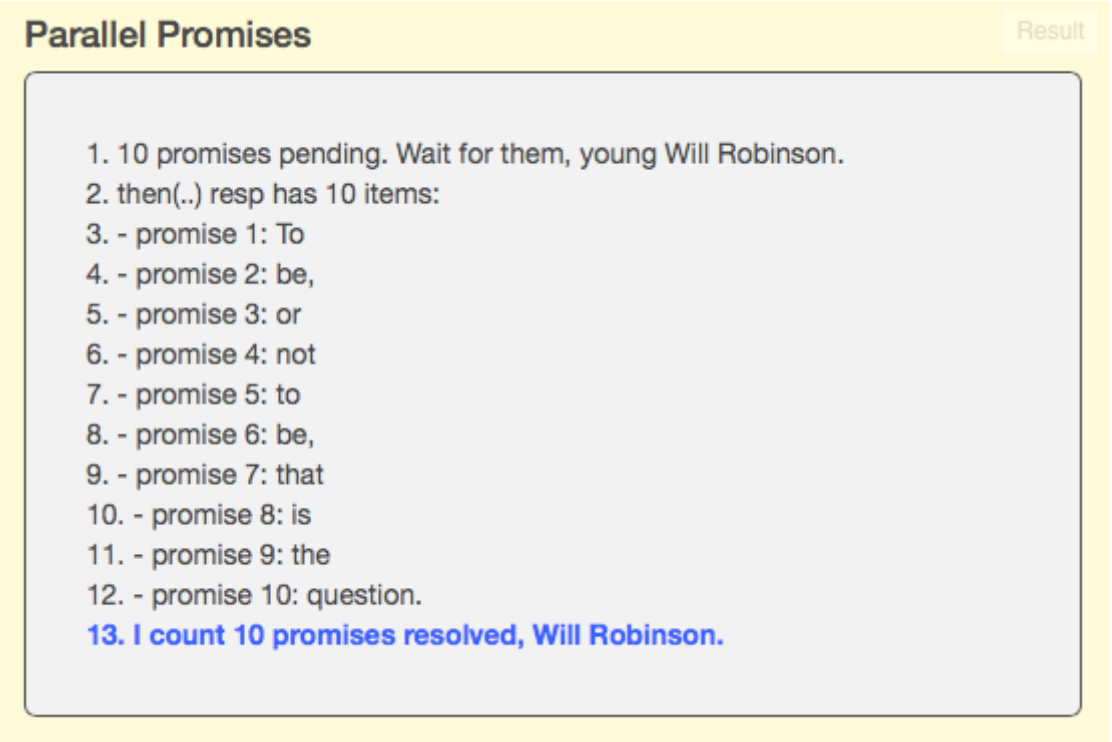


Figure 7: JSFiddle demo of parallel promises

# Birthing a Promise

Many APIs return a promise having a then function – they're *thenable*. Normally I could just chain a *then* to a *thenable* function's result. Otherwise, the $q, mpromise, Q, and kew libraries have a simple API used to create, reject, or resolve a promise. There are links to API documentation for each library in the references section. I've not usually needed to construct a promise, except for wrapping promise-ignorant literals and timeout functions for this article. See the examples where I created promises.

# Promise Library Interoperation

Most JavaScript promise libraries interoperate at the *then* level. You can create a promise from a foreign promise because a promise can wrap any kind of value. This works across libraries that support *then*. There are disparate promise functions aside from *then*. If you need a function that your library doesn't include, you can wrap a promise from your library in a new promise from a library that has your desired function. For example, JQuery promises are sometimes maligned in the literature. You could wrap each right away in a Q, $q, mpromise, or kew promise to operate in that library.

# Finally

I wrote this article as someone who one year ago hesitated to embrace promises. I was simply trying to get a job done. I didn't want to learn a new API or to chance breaking my code due to misunderstanding promises. Was I ever wrong! When I got off the dime, I easily achieved gratifying results.

In this article, I've given simplistic examples of a single promise, a promise chain, and a parallel promise of promises. Promises are not hard to use. If I can use them, anybody can. To flesh out the concept, I encourage you to click up the supplied references written by promise experts. Begin at the *Promises/A* reference, the de-facto standard for JavaScript promises.

If you have not directly used promises, give them a try. Resolved: you'll have a good experience. I promise!

CodeProject

– Lou Mauget, asktheteam@keyholesoftware.com

## Reference Links


- http://wiki.commonjs.org/wiki/Promises/A

- https://github.com/bellbind/using-promise-q/
- https://github.com/Medium/kew
- https://docs.angularjs.org/api/ng/service/$q
- https://github.com/aheckmann/mpromise
- http://blog.mediumequalsmessage.com/promise-deferred-objects-in-javascript-pt2-practical-use
- https://gist.github.com/domenic/3889970
- http://sitr.us/2012/07/31/promise-pipelines-in-javascript.html
- http://dailyjs.com/2014/02/20/promises-in-detail/
- https://www.promisejs.org/
- http://solutionoptimist.com/2013/12/27/javascript-promise-chains-2/
- http://www.erights.org/elib/distrib/pipeline.html
- http://zeroturnaround.com/rebellabs/monadic-futures-in-java8/

# License



# Share



# About the Author


## Keyhole Software

Keyhole Software
United States


Keyhole is a software development and consulting firm with a tight-knit technical team. We work primarily with Java, .NET, and Mobile technologies, specializing in application development. We love the challenge that comes in consulting and blog often regarding some of the technical situations and technologies we face. Kansas City, St. Louis and Chicago.

Group type: Organisation

3 members

Apply to join this group

# Comments and Discussions