

# DERICKBAILEY.COM

Trade Secrets Of A Developer / Entrepreneur

[ABOUT](#)[TWITTER](#)[G+](#)[RSS](#)[BLOG](#)[COURSES](#)[PRODUCTS](#)[NEWSLETTER](#)[PUBLICATIONS](#)[PODCASTS](#)[SPEAKING](#)

## Promises, Modal Dialogs and Resolve vs Reject

August 20, 2015 By [derickbailey](#)

Someone on twitter asked me a question about promises a while back. It was a good question about the general use of reject and resolve to manage a yes/no dialog box.

“ [@derickbailey](#) I'm thinking on how I would turn showing a simple modal into a promise. Better to call resolve on hide event or btn press?

— moniuchos (@moniuchos)

[August 1, 2015](#)

“ [@derickbailey](#) For confirmation dialogs (yes/no) would you make “no” call .reject or rather .resolve it with a false parameter?

— moniuchos (@moniuchos)

[August 1, 2015](#)

The short answer is always resolve with a status indicator.



But I think @moniuchos is looking for something more along the lines of *why* you would use resolve or reject – not just this one specific, terse answer for this one specific situation.

To understand the answer, there's some background to dig in to: managing the result of a modal dialog, and understanding reject vs resolve.

## Managing The Result Of A Modal Dialog

There's a thousand different modal dialog options in JavaScript. Personally, I've used various jQuery extensions

including Bootstrap, KendoUI and many others. The important part is not which library you're using, but how you manage the result of the dialog.

Generally speaking, my use of any view is handled with a mediator – [a workflow object](#) that manages the over-all process. In the example from that post, the “getEmployeeDetail” method could easily be modified to use a modal dialog instead of just displaying the form in a normal DOM element (using Bootstrap in this example):

```
1 orgChart = {  
2  
3     addNewEmployee: function(){  
4         var employeeDetail = this.getEmployeeDetail();  
5  
6         employeeDetail.on("cancel", () => {  
7             // the modal was cancelled...  
8             // go back to previous UI / step / whatever  
9         });  
10  
11         employeeDetail.on("complete", (detail) => {  
12             // the detail was entered. use it wherever  
13             // and then move the UI forward to the next step  
14         });  
15  
16     },  
17  
18     getEmployeeDetail: function(){  
19         var form = new EmployeeDetailForm();  
20         form.render();  
21  
22         // use a modal dialog, here  
23         $("#my-modal").modal(form.el);  
24  
25         return form;  
26     }  
27 }
```

1.js hosted with ❤ by GitHub

[view raw](#)

Notice, in this example, that the result of the employee detail modal dialog is split across two possible events: a “cancel” event or a “complete” event. Having these two events split apart makes it easy to write code for each specific path. But a promise doesn’t have a “cancel” and “complete”. It only has a “reject” or “resolve” – which are not equivalent.

To move the modal code toward something that the promise can better work with, a single “closed” event could be used with the modal dialog, passing a result object with a status back to the mediator:

```
1 orgChart = {  
2  
3     addNewEmployee: function(){  
4         var employeeDetail = this.getEmployeeDetail();  
5  
6         employeeDetail.on("close", (result) => {  
7             if (result.cancelled) {  
8  
9                 // the modal was cancelled...  
10                // go back to previous UI / step / whatever  
11  
12            } else {  
13  
14                // it was completed  
15                // pass any data needed as "result.data" and  
16                // use that data wherever it is needed  
17                // then move the UI forward to the next step  
18  
19            }  
20        });  
21  
22    },  
23  
24    getEmployeeDetail: function(){  
25        var form = new EmployeeDetailForm();  
26        form.render();  
27    }  
};
```

```

28   // use a modal dialog, here
29   $("#my-modal").modal(form.el);
30
31   return form;
32 }
33 }
```

2.js hosted with ❤ by GitHub

[view raw](#)

The change in this file is to have a single result object passed through a single “closed” event. You would then have to examine the result to see what should be done next.

Note that neither of these examples is the “right” or “wrong” way to do it. Which you would use when is a matter of preference and functional needs at any given point in the application. With the second example, though, it will be easier to work with the “resolve” method of a project. To understand why it will be easier this way, you need to understand the purpose of reject vs resolve in a promise.

## When To Reject Or Resolve A Promise

There are a lot of “promise” objects and libraries and specifications out there. But with ES6 (ES2015) being “done” and work to implement them in many browsers underway, I’m going to assume that the [ES6 Promise object/specification](#) is being used.

With that in mind, an understanding of reject vs resolve can be extrapolated from the [MDN documentation on ES6 Promises](#):

“ A Promise is in one of these states:

- *pending: initial state, not fulfilled or rejected.*
- *fulfilled: meaning that the operation completed successfully.*
- *rejected: meaning that the operation failed.*

The first state, pending, isn't really meaningful right now.

Fulfilled vs rejected is what we care about. In order to fulfill a promise, the “resolve” method is called. In order to reject a promise, we call “reject”

```

1 new Promise((resolve, reject) => {
2   // some work was done
3   resolve(result);
4
5   //or
6
7   // the work failed
8   reject(reason);
9 });

```

3.js hosted with ❤ by GitHub

[view raw](#)

The question, though, is what does it mean for an operation to “fail”? Is a “cancel” button or the little red (x) on a modal dialog a “failed” operation? Or is that simply a “cancelled” state for an operation that succeeds? The answer is found by looking at how a promise behaves when you reject it.

```

1 var p = new Promise(function(resolve, reject){
2   reject("some reason");
3 });
4
5 function onReject(reason){
6   console.log(reason);
7 }
8
9 p.then(undefined, onReject)

```

```
10 | p.catch(onReject);
```

4.js hosted with ❤ by GitHub

[view raw](#)

In this example, the output will be “some reasons” printed twice to the console. This happens because the rejection is being handled twice – once with the second parameter to the “then” call, and again with a “catch” call.

If you’ve ever done any error handling in JavaScript, C++, C#, Java, any of a number of other languages, you probably recognize the “catch” keyword as the way to handle an error condition. And indeed, this is what the “catch” method does on a promise, as stated in [the MDN documentation for the catch method](#):

*“ The catch method can be useful for error handling in your promise composition.*

You can also verify this by throwing an exception from within your promise, watching both the “catch” and “onReject” callbacks firing:

```
1 var p = new Promise(function(resolve, reject){
2   throw new Error("some error");
3 });
4
5 function onReject(reason){
6   console.log(reason);
7 }
8
9 p.then(undefined, onReject);
10 p.catch(onReject);
```

5.js hosted with ❤ by GitHub

[view raw](#)

All of this points to the conclusion that you call “reject” when the “failure” of the process is a truly exceptional state – something that was not anticipated. If an error is thrown, a state that cannot be handled is found, or some other

condition that cannot be handled through normal means occurs, this is when you “reject” the promise.

But, a “cancel” button or the little red (x) being clicked? That is certainly not an unexpected or exceptional state. That is something your code should handle under normal circumstances, not as an error condition with the equivalent of a try / catch block.

In other words, a “cancel” or “no” or click of a red (x) to close a dialog is something to be handled via the “resolve” method of a promise, not the reject method.

## Resolving The Close Of A Modal Dialog

With this new found knowledge of when to resolve vs reject, let’s go back to the modal dialog shown in the earlier example.

The second of the two code listings shows a single “closed” event that is used to deliver the status of the dialog box back to the mediator object that controls the higher-level flow. Since a promise has a single “resolve” method, the single “closed” event from the modal dialog makes life a bit easier.

The code can be modified (with many details omitted in this example) with only a few changes to the workflow:

```
1 orgChart = {  
2  
3   addNewEmployee: function(){  
4     var employeeDetail = this.getEmployeeDetail();  
5  
6     employeeDetail.then((result) => {  
7       if (result.cancelled) {  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
5510  
5511  
5512  
5513  
5514  
5515  
5516  
5517  
5518  
5519  
5520  
5521  
5522  
5523  
5524  
5525  
5526  
5527  
5528  
5529  
5530  
5531  
5532  
5533  
5534  
5535  
5536  
5537  
5538  
5539  
55310  
55311  
55312  
55313  
55314  
55315  
55316  
55317  
55318  
55319  
55320  
55321  
55322  
55323  
55324  
55325  
55326  
55327  
55328  
55329  
55330  
55331  
55332  
55333  
55334  
55335  
55336  
55337  
55338  
55339  
55340  
55341  
55342  
55343  
55344  
55345  
55346  
55347  
55348  
55349  
55350  
55351  
55352  
55353  
55354  
55355  
55356  
55357  
55358  
55359  
55360  
55361  
55362  
55363  
55364  
55365  
55366  
55367  
55368  
55369  
55370  
55371  
55372  
55373  
55374  
55375  
55376  
55377  
55378  
55379  
55380  
55381  
55382  
55383  
55384  
55385  
55386  
55387  
55388  
55389  
55390  
55391  
55392  
55393  
55394  
55395  
55396  
55397  
55398  
55399  
553100  
553101  
553102  
553103  
553104  
553105  
553106  
553107  
553108  
553109  
553110  
553111  
553112  
553113  
553114  
553115  
553116  
553117  
553118  
553119  
5531100  
5531101  
5531102  
5531103  
5531104  
5531105  
5531106  
5531107  
5531108  
5531109  
55311010  
55311011  
55311012  
55311013  
55311014  
55311015  
55311016  
55311017  
55311018  
55311019  
553110100  
553110101  
553110102  
553110103  
553110104  
553110105  
553110106  
553110107  
553110108  
553110109  
553110110  
553110111  
553110112  
553110113  
553110114  
553110115  
553110116  
553110117  
553110118  
553110119  
5531101100  
5531101101  
5531101102  
5531101103  
5531101104  
5531101105  
5531101106  
5531101107  
5531101108  
5531101109  
5531101110  
5531101111  
5531101112  
5531101113  
5531101114  
5531101115  
5531101116  
5531101117  
5531101118  
5531101119  
55311011100  
55311011101  
55311011102  
55311011103  
55311011104  
55311011105  
55311011106  
55311011107  
55311011108  
55311011109  
55311011110  
55311011111  
55311011112  
55311011113  
55311011114  
55311011115  
55311011116  
55311011117  
55311011118  
55311011119  
553110111100  
553110111101  
553110111102  
553110111103  
553110111104  
553110111105  
553110111106  
553110111107  
553110111108  
553110111109  
553110111110  
553110111111  
553110111112  
553110111113  
553110111114  
553110111115  
553110111116  
553110111117  
553110111118  
553110111119  
5531101111100  
5531101111101  
5531101111102  
5531101111103  
5531101111104  
5531101111105  
5531101111106  
5531101111107  
5531101111108  
5531101111109  
5531101111110  
5531101111111  
5531101111112  
5531101111113  
5531101111114  
5531101111115  
5531101111116  
5531101111117  
5531101111118  
5531101111119  
55311011111100  
55311011111101  
55311011111102  
55311011111103  
55311011111104  
55311011111105  
55311011111106  
55311011111107  
55311011111108  
55311011111109  
55311011111110  
55311011111111  
55311011111112  
55311011111113  
55311011111114  
55311011111115  
55311011111116  
55311011111117  
55311011111118  
55311011111119  
553110111111100  
553110111111101  
553110111111102  
553110111111103  
553110111111104  
553110111111105  
553110111111106  
553110111111107  
553110111111108  
553110111111109  
553110111111110  
553110111111111  
553110111111112  
553110111111113  
553110111111114  
553110111111115  
553110111111116  
553110111111117  
553110111111118  
553110111111119  
5531101111111100  
5531101111111101  
5531101111111102  
5531101111111103  
5531101111111104  
5531101111111105  
5531101111111106  
5531101111111107  
5531101111111108  
5531101111111109  
5531101111111110  
5531101111111111  
5531101111111112  
5531101111111113  
5531101111111114  
5531101111111115  
5531101111111116  
5531101111111117  
5531101111111118  
5531101111111119  
55311011111111100  
55311011111111101  
55311011111111102  
55311011111111103  
55311011111111104  
55311011111111105  
55311011111111106  
55311011111111107  
55311011111111108  
55311011111111109  
55311011111111110  
55311011111111111  
55311011111111112  
55311011111111113  
55311011111111114  
55311011111111115  
55311011111111116  
55311011111111117  
55311011111111118  
55311011111111119  
553110111111111100  
553110111111111101  
553110111111111102  
553110111111111103  
553110111111111104  
553110111111111105  
553110111111111106  
553110111111111107  
553110111111111108  
553110111111111109  
553110111111111110  
553110111111111111  
553110111111111112  
553110111111111113  
553110111111111114  
553110111111111115  
553110111111111116  
553110111111111117  
553110111111111118  
553110111111111119  
5531101111111111100  
5531101111111111101  
5531101111111111102  
5531101111111111103  
5531101111111111104  
5531101111111111105  
5531101111111111106  
5531101111111111107  
5531101111111111108  
5531101111111111109  
5531101111111111110  
5531101111111111111  
5531101111111111112  
5531101111111111113  
5531101111111111114  
5531101111111111115  
5531101111111111116  
5531101111111111117  
5531101111111111118  
5531101111111111119  
55311011111111111100  
55311011111111111101  
55311011111111111102  
55311011111111111103  
55311011111111111104  
55311011111111111105  
55311011111111111106  
55311011111111111107  
55311011111111111108  
55311011111111111109  
55311011111111111110  
55311011111111111111  
55311011111111111112  
55311011111111111113  
55311011111111111114  
55311011111111111115  
55311011111111111116  
55311011111111111117  
55311011111111111118  
55311011111111111119  
553110111111111111100  
553110111111111111101  
553110111111111111102  
553110111111111111103  
553110111111111111104  
553110111111111111105  
553110111111111111106  
553110111111111111107  
553110111111111111108  
553110111111111111109  
553110111111111111110  
553110111111111111111  
553110111111111111112  
553110111111111111113  
553110111111111111114  
553110111111111111115  
553110111111111111116  
553110111111111111117  
553110111111111111118  
553110111111111111119  
5531101111111111111100  
5531101111111111111101  
5531101111111111111102  
5531101111111111111103  
5531101111111111111104  
5531101111111111111105  
5531101111111111111106  
5531101111111111111107  
5531101111111111111108  
5531101111111111111109  
5531101111111111111110  
5531101111111111111111  
5531101111111111111112  
5531101111111111111113  
5531101111111111111114  
5531101111111111111115  
5531101111111111111116  
5531101111111111111117  
5531101111111111111118  
5531101111111111111119  
55311011111111111111100  
55311011111111111111101  
55311011111111111111102  
55311011111111111111103  
55311011111111111111104  
55311011111111111111105  
55311011111111111111106  
55311011111111111111107  
55311011111111111111108  
55311011111111111111109  
55311011111111111111110  
55311011111111111111111  
55311011111111111111112  
55311011111111111111113  
55311011111111111111114  
55311011111111111111115  
55311011111111111111116  
55311011111111111111117  
55311011111111111111118  
55311011111111111111119  
553110111111111111111100  
553110111111111111111101  
553110111111111111111102  
553110111111111111111103  
553110111111111111111104  
553110111111111111111105  
553110111111111111111106  
553110111111111111111107  
553110111111111111111108  
553110111111111111111109  
553110111111111111111110  
553110111111111111111111  
553110111111111111111112  
553110111111111111111113  
553110111111111111111114  
553110111111111111111115  
553110111111111111111116  
553110111111111111111117  
553110111111111111111118  
553110111111111111111119  
5531101111111111111111100  
5531101111111111111111101  
5531101111111111111111102  
5531101111111111111111103  
5531101111111111111111104  
5531101111111111111111105  
5531101111111111111111106  
5531101111111111111111107  
5531101111111111111111108  
5531101111111111111111109  
5531101111111111111111110  
5531101111111111111111111  
5531101111111111111111112  
5531101111111111111111113  
5531101111111111111111114  
5531101111111111111111115  
5531101111111111111111116  
5531101111111111111111117  
5531101111111111111111118  
5531101111111111111111119  
55311011111111111111111100  
55311011111111111111111101  
55311011111111111111111102  
55311011111111111111111103  
55311011111111111111111104  
55311011111111111111111105  
55311011111111111111111106  
55311011111111111111111107  
55311011111111111111111108  
55311011111111111111111109  
55311011111111111111111110  
55311011111111111111111111  
55311011111111111111111112  
55311011111111111111111113  
55311011111111111111111114  
55311011111111111111111115  
553110111111111111111
```

```

  9   // the modal was cancelled...
 10   // go back to previous UI / step / whatever
 11
 12 } else {
 13
 14   // it was completed
 15   // pass any data needed as "result.data" and
 16   // use that data wherever it is needed
 17   // then move the UI forward to the next step
 18
 19 }
 20 });
 21 },
 22
 23 getEmployeeDetail: function(){
 24   var formP = ... // some code to get a promise from th
 25   return formP;
 26 }
 27 }
```

6.js hosted with ❤ by GitHub

[view raw](#)

The callback function for the promises “then” hasn’t changed all, from the callback for the “closed” event. The major difference is found in how that callback is executed. Rather than being event driven, it is now promise driven.

## Handling Additional Results

It might seem trivial to handle a form close as a “reject” in a promise, even when looking at the above reasoning and examples. But if you did that, you would quickly run in to some rather serious limitations and complexities.

For example, if a modal dialog needs to change from simply yes/no or closed/complete to a more complex set of results, you would be in trouble with “resolve” as yes and “reject” as no. Say you’re implementing a wizard style UI with promises. What do you do when the wizard has a “next”, and a “previous” event, as well as a “cancel” and

“complete” event? You’ve only got two states you can model this within, if you’re using resolve and reject as positive and negative responses.

Even if you’re only dealing with yes/no results, modeling no as a reject is dangerous because of the way exceptions are handled. You saw in the promise example that throws an exception, how the exception is handled in the “catch” or “onReject” callbacks. If you tried to model a “no” as reject, you would end up with logic in your “onReject” callback to check if you’re dealing with an exception or simply a negative response. This kind of logic gets complicated, quickly. It makes the code brittle and difficult to work with.

All that being said, having everything modeled in the “resolve” of a promise isn’t always the best way to move forward, either.

## It’s All About Tradeoffs

[My original workflow blog post](#) doesn’t use promises or single return values with status codes for a very specific reason: explicit handling of state changes tends to reduce complexity in code.

For every state / result that can be produced by a given object, a single “closed” or “resolved” handler would require you to add yet another branch of logic to your if-statement or switch-statement. With only two states to handle, this may not be a problem, but it will quickly get out of hand.

By modeling each return status as a separate event from the form, it will be easier to add / remove / change the number of possible results without adding complexity to the code. Rather than having a series of if-statements or a

long switch statement, each result will be facilitated by an explicit callback for that result.

The downside to the explicit callback per result pattern, is added complexity in managing all of the possible states. You need good documentation and probably a fair amount of testing to make sure you have handled all of the necessary results callbacks.

## In The End ...

If you're already dealing with a promise-based API, you can use resolve to manage the result of a modal dialog or any other code.

If you don't have an event-based API on which you can model a specific event per result, or if doing that would add complexity in managing the object life-cycles, a promise can be a good way to handle things.

Personally, I prefer event based systems for the functionality I've shown here. I do use promises on a regular basis, though. They are a powerful tool with a lot of great features for making our lives easier. And I'm glad to see promises becoming a standard part of JavaScript.

Tweet

---

## RELATED POST

[10 Myths About Docker That Stop Developers Cold](#)

[Docker Recipes Update: Speed Up npm install In Mou...](#)

[Update and a Bonus Recipe for the Docker Recipes e...](#)

[A Sneak Peak at Docker Recipes for Node.js Develop...](#)

[Docker Recipes for Node.js: Pre-sale Dates & ...](#)

---

Filed Under: [Backbone](#), [Design](#), [JavaScript](#), [Patterns](#), [Promises](#), [Workflow](#)



### About derickbailey

Derick Bailey is a developer, entrepreneur, author, speaker and technology leader in central Texas (north of Austin). He's been a professional developer since the late 90's, and has been writing code since the late 80's. In his spare time, he gets called a spamming marketer by people on Twitter, and blurts out all of the stupid / funny things he's ever done in his career on [his email newsletter](#).

## DERICK BAILEY AROUND THE WEB

Twitter: [@derickbailey](#)

Google+: [DerickBailey](#)

Screencasts: [WatchMeCode.net](#)

eBook: [Building Backbone Plugins](#)

Copyright © 2016 [Muted Solutions, LLC](#). All Rights Reserved · [Log in](#)