# 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):

```
orgChart = {

  addNewEmployee: function(){
    var employeeDetail = this.getEmployeeDetail();

    employeeDetail.on("cancel", () => {
      // the modal was cancelled...
      // go back to previous UI / step / whatever
    });

    employeeDetail.on("complete", (detail) => {
      // the detail was entered. use it wherever
      // and then move the UI forward to the next step
    });

  },

  getEmployeeDetail: function(){
    var form = new EmployeeDetailForm();
    form.render();

    // use a modal dialog, here
    $("#my-modal").modal(form.el);

    return form;
  }
}
```

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:

```
orgChart = {

  addNewEmployee: function(){
    var employeeDetail = this.getEmployeeDetail();

    employeeDetail.on("close", (result) => {
      if (result.cancelled) {

        // the modal was cancelled...
        // go back to previous UI / step / whatever

      } else {

        // it was completed
        // pass any data needed as "result.data" and
        // use that data wherever it is needed
        // then move the UI forward to the next step

      }
    });

  },

  getEmployeeDetail: function(){
    var form = new EmployeeDetailForm();
    form.render();

```

```
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"

```
new Promise((resolve, reject) => {
  // some work was done
  resolve(result);

  //or

  // the work failed
  reject(reason);
});
```

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.

```
var p = new Promise(function(resolve, reject){
  reject("some reason");
});

function onReject(reason){
  console.log(reason);
}

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           // 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.



---

**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