Beginner's Guide to JavaScript promises

Created: Sept 21st, 2015

JavaScript Promises are a new addition to ECMAscript 6 that aims to provide a cleaner, more intuitive way to deal with the completion (or failure) of asynchronous tasks. Up until JavaScript Promises, that job is taken on by JavaScript event handlers (ie: `image.onload`) and callback functions, popularized by libraries such as jQuery and Node.js, with varying degrees of frustration. Event handlers work well with individual elements, but what if you wanted to, for example, be notified when a collection of images have all been loaded or the order in which they happened? Call back functions that you pass as the last parameter to methods that support it such as jQuery's `animate()` function perform their job admirably for running custom code when a task is complete, but what if the custom code in itself also needs to call `animate()` with another call back function, and so on? You end up with what's called "callback hell", or a growing stack of call back functions resembling the Tower of Babel.

JavaScript Promises provide a mechanism for tracking the state of an asynchronous task with more robustness and less chaos. But first thing's first.

JavaScript Promises support and Polyfill

JavaScript Promises are part of the ECMAscript 6 standards and should be supported by all browsers eventually. At the moment that promise is already realized in recent versions of Chrome, FF, Safari, and on mobile browsers with the exception of IE. Check out this page for the skinny. Due to the absence of IE including IE11 in the green column, you can use a Polyfill such as es6-promise.js to bridge the gap until IE catches up with the rest of the herd. Just download **es6-promise-min.js** and include it at the top of your page as an external JavaScript, and viola!

The syntax

Ok, lets get down to business now. At the heart of JavaScript Promises is the Promise constructor function, which is called like so:

```
var mypromise = new Promise(function(resolve, reject){
 // asynchronous code to run here
 // call resolve() to indicate task successfully completed
 // call reject() to indicate task has failed
})
```

It is passed an anonymous function with two parameters- a `resolve()` method that you call at some point to set the state of the promise to ***fulfilled***, and `reject()` to set it to ***rejected*** instead. A Promise object starts out with a state of ***pending***, to indicate the asynchronous code it's monitoring has neither completed (fulfilled) or failed (rejected). Lets get our first taste of JavaScript Promises in action with a function that dynamically loads an image based on the image URL:

```
function getImage(url){
    return new Promise(function(resolve, reject){
        var img = new Image()
        img.onload = function(){
            resolve(url)
        }
        img.onerror = function(){
            reject(url)
        }
        img.src = url
    })
}
```

The `getImage()` function returns a Promise object that keeps track of the state of the image load. When you call:

```
getImage('doggy.gif')
```

its promise object goes from the initial state of "pending" to either fulfilled or rejected eventually depending on the outcome of the image load. Notice how we've passed the URL of the image to both the `resolve()` and `reject()` method of Promise; this could be any data you wish to be processed further depending on the outcome of the task. More on this later.

Ok, at this point Promises may just seem like a pointless exercise to set some object's state to indicate the status of a task. But as you'll soon see, with this mechanism comes the ability to easily and intuitively define what happens next once the task is completed.

The then() and catch() methods

Whenever you instantiate a Promise object, two methods- `then()` and `catch()`- become available to decide what happens next after the conclusion of an asynchronous task. Take a look at the below:

```
getImage('doggy.jpg').then(function(successurl){
    document.getElementById('doggyplayground').innerHTML = '<img src="' + successurl + '" />'
})
```

Here as soon as "doggy.jpg" has loaded, we specify that the image be shown inside the "`doggyplayground`" DIV. The original `getImage()` function returns a Promise object, so we can call `then()` on it to specify what happens when the request has been **resolved**. The URL of the image we passed into the `resolve()` function when we created `getImage()` becomes available as the parameter inside the `then()` function.

What happens if the image failed to load? The `then()` method can accept a 2nd function to deal with the **rejected** state of the Promise object:

```
getImage('doggy.jpg').then(
    function(successurl){
        document.getElementById('doggyplayground').innerHTML = '<img src="' + successurl + '" />'
    },
    function(errorurl){
        console.log('Error loading ' + errorurl)
    }
)
```

In such a construct, if the image loads, the first function inside `then()` is run, if it fails, the 2nd instead. We can also handle errors using the `catch()` method instead:

```
getImage('doggy.jpg').then(function(successurl){
    document.getElementById('doggyplayground').innerHTML = '<img src="' + successurl + '" />'
}).catch(function(errorurl){
    console.log('Error loading ' + errorurl)
})
```

Calling `catch()` is equivalent to calling `then(undefined, function)`, so the above is the same as:

```
getImage('doggy.jpg').then(function(successurl){
    document.getElementById('doggyplayground').innerHTML = '<img src="' + successurl + '" />'
}).then(undefined, function(errorurl){
    console.log('Error loading ' + errorurl)
})
```

Using recursion to load and display images sequentially

Lets say we have an array of images we want to load and display sequentially- that is to say, first load and show image1, and once that's complete, go on to image2, and so on. We'll talk about **chaining promises** together further below to accomplish this, but one approach is just to use recursion to go through the list of images, calling our `getImage()` function each time with a `then()` method that shows the current image before calling `getImage()` again until all of the images have been processed. Here is the code:

```
var doggyplayground = document.getElementById('doggyplayground')
var doggies = ['dog1.png', 'dog2.png', 'dog3.png', 'dog4.png', 'dog5.png']

function displayimages(images){
    var targetimage = images.shift() // process doggies images one at a time
    if (targetimage){ // if not end of array
        getImage(targetimage).then(function(url){ // load image then...
            var dog = document.createElement('img')
            dog.setAttribute('src', url)
            doggyplayground.appendChild(dog) // add image to DIV
            displayimages(images) // recursion- call displayimages() again to process next image/doggy
        }).catch(function(url){ // handle an image not loading
            console.log('Error loading ' + url)
            displayimages(images) // recursion- call displayimages() again to process next image/doggy
        })
    }
}

displayimages(doggies)
```

Demo (fetch and display images sequentially):

Get images

The `displayimages()` function takes an array of images and sequentially goes through each image, by calling `images.shift()`. For each image, we first call `getImage()` to fetch the image, then the returned Promise object's `then()` method to specify what happens next, in this case, add the image to the `doggyplayground` DIV before calling `displayimages()` again. In the case of an image failing to load, the `catch()` method handles those instances. The recursion stops when the doggies array is empty, after `Array.shift()` has gone through all of its elements.

Using recursion with JavaScript Promises is one way to sequentially process a series of asynchronous tasks. Another more versatile method is by learning the art of chaining promises. Lets see what that's all about now.

Chaining Promises

We already know that the `then()` method can be invoked on a Promise instance to specify what happens after the completion of a task. However, we can in fact chain multiple `then()` methods together, in turn chaining multiple promises together, to specify what happens after each promise has been resolved, in sequence. Using our trusted `getImage()` function to illustrate, the following fetches one image before fetching another:

```
getImage('dog1.png').then(function(url){
    console.log(url + ' fetched!')
    return getImage('dog2.png')
}).then(function(url){
```

```
    console.log(url + ' fetched!')
})

//Console log:
// dog1.png fetched
// dog2.png fetched!
```

So what's going on here? Notice inside the first `then()` method, the line:

```
return getImage('dog2.png')
```

This fetches "dog2.png" and returns a Promise object. By returning a Promise object inside `then()`, the next `then()` waits for that promise to resolve before running, accepting as its parameter the data passed on by the new Promise object. **This is the key to chaining multiple promises together- by returning another promise inside the `then()` method.**

Note that we can also simply return a static value inside `then()`, which would simply be carried on and executed immediately by the next `then()` method as its parameter value.

With the above example we still want to account for an image not loading, so we'll include the `catch()` method as well:

```
getImage('baddog1.png').then(function(url){
    console.log(url + ' fetched!')
}).catch(function(url){
    console.log(url + ' failed to load!')
}).then(function(){
    return getImage('dog2.png')
}).then(function(url){
    console.log(url + ' fetched!')
}).catch(function(url){
    console.log(url + ' failed to load!')
})

//Console log:
// baddog1.png failed to load!
// dog2.png fetched!
```

Recall that `catch()` is synonymous with `then(undefined, functionref)`, so after `catch()` the next `then()` will still be executed. Notice the organization of the `then()` and `catch()` methods- we put the return of the next promise object (or link in the chain) inside its own `then()` method, after the outcome of the previous promise is completely accounted for via the `then()` and `catch()` method proceeding it.

If you wanted to load 3 images in succession, for example, we could just add another set of `then()` `then()` `catch()` to the above code.

Creating a sequence of Promises

Ok, so we know the basic idea of chaining promises together is to return another promise inside the `then()` method. But manually chaining promises together can quickly become unmanageable. For longer chains, what we need is a way to start with an empty Promise object and programmatically pile on the desired `then()` and `catch()` methods to form the final sequence of promises. In JavaScript Promises, we can create a blank Promise object that's **resolved** to begin with with the line:

```
var resolvedPromise = Promise.resolve()
```

There is also `Promise.reject()` to create a blank Promise object that's already in the rejected state. So why would we want a new Promise object that's already resolved you may ask? Well, it makes for a perfect Promise object to chain additional promises together, since an already resolved Promise object will automatically jump to the first `then()` method added to it, and kick start the chain of events.

We can use a resolved Promise object to create a sequence of promises, by piling on `then()` and `catch()` methods to it. For example:

```
var sequence = Promise.resolve()
var doggies = ['dog1.png', 'dog2.png', 'dog3.png', 'dog4.png', 'dog5.png']

doggies.forEach(function(targetimage){
    sequence = sequence.then(function(){
        return getImage(targetimage)
    }).then(function(url){
        console.log(url + ' fetched!')
    }).catch(function(err){
        console.log(err + ' failed to load!')
    })
})

//Console log:
// dog1.png fetched
// dog2.png fetched!
// dog3.png fetched!
// dog4.png fetched!
// dog5.png fetched!
```

First we create a resolved Promise object called `sequence`, then go through each element inside the `doggies[]` array with `forEach()`, adding to `sequence` the required `then()` and `catch()` methods to handle each image after it's loaded. The result is a series of `then()` and `catch()` methods attached to `sequence`, creating the desired timeline of loading each image one at a time.

In case you're wondering, instead of using `forEach()` to cycle through the image array, you can also use a simple `for` loop instead, though the result may be more than you had bargained for:

```
var sequence = Promise.resolve()
var doggies = ['dog1.png', 'dog2.png', 'dog3.png', 'dog4.png', 'dog5.png']

for (var i=0; i<doggies.length; i++){
    (function(){ // define closure to capture i at each step of loop
        var capturedindex = i
        sequence = sequence.then(function(){
            return getImage(doggies[capturedindex])
        }).then(function(url){
            console.log(url + ' fetched!')
        }).catch(function(err){
            console.log('Error loading ' + err)
        })
    }()) // invoke closure function immediately
}

//Console log:
// dog1.png fetched
// dog2.png fetched!
// dog3.png fetched!
// dog4.png fetched!
// dog5.png fetched!
```

Inside the `for` loop, to properly get the value of `i` at each step and pass it into `then()`, we need to create an outer closure to capture each value of `i`. Without the outer closure, the value of `i` passed into `then()` each time will simply be the value of `i` when it's reached the end of the loop, or `doggies.length-1`. If all of this confounds you, the article JavaScript closures in for loops should help clear the air.

Creating an array of promises

Instead of chaining promises together, we can also create an array of promises. This makes it easy to do something after all of the asynchronous tasks have completed, instead of after each task. For example, the following uses `getImage()` to fetch two images and store them as an array of promises:

```
var twodoggypromises = [getImage('dog1.png'), getImage('dog2.png')]
```

Since `getImage()` when called returns a promise, `twodoggypromises` now contains two Promise objects. So what can we do with an array of promises? Well, we can then use the static method `Promise.all()` to do something after all promises inside the array have resolved:

```
Promise.all(twodoggypromises).then(function(urls){
    console.log(urls) // logs ['dog1.png', 'dog2.png']
}).catch(function(urls){ // if any image fails to load, then() is skipped and catch is called
    console.log(urls) // returns array of images that failed to load
})
```

`Promise.all()` takes an iterable (array or array-like list) of promise objects, and waits until all of those promises have been fulfilled before moving on to any `then()` method attached to it. The `then()` method is passed an array of returned values from each promise.

So what happens if one of the promises inside the array doesn't resolve (is rejected)? In that case the entire `then()` portion is ignore, and `catch()` is executed instead. So in the above scenario, if one or more of the images fails to load, it only logs an array of images that failed to load inside `catch()`.

Displaying images when they have all been fetched

It's high time now to see an example of showing off all the doggies when they have been fetched, instead of one at a time. We'll use `Array.map()` to ease the pain in creating a promise array:

```
var doggies = ['dog1.png', 'dog2.png', 'dog3.png', 'dog4.png', 'dog5.png']
var doggypromises = doggies.map(getImage) // call getImage on each array element and return array of promises

Promise.all(doggypromises).then(function(urls){
    for (var i=0; i<urls.length; i++){
        var dog = document.createElement('img')
        dog.setAttribute('src', urls[i])
        doggyplayground.appendChild(dog)
    }
}).catch(function(urls){
    console.log("Error fetching some images: " + urls)
})
```

`Array.map()` iterates through the original array and calls `getImage()` on each element, returning a new array using the return value of `getImage()` at each step, or a promise. The result is twofold- each image gets fetched, and in turn we get back an array of corresponding promises. Then, we put `Promise.all()` to work, passing in `doggypromises` to show all the images at once:

Demo (fetch and display images at all once):

Get images

Fetch images all at once, but display then in sequence as each one becomes ready

Finally, as if the dogs haven't been paraded enough, lets introduce them to the doggy park in an optimized manner, not one by one, not all at once, but the best of both words. We'll fetch all of the images at once to take advantage of parallel downloading in browsers, but show them in sequence as each one becomes available (fetched). This minimizes the time the doggies show up while still showing them in orderly sequence.

To do this, we just have to do two things we already know- create an array of promises to fetch all images at once (in parallel), then create a sequence of promises to actually show each image one at a time:

```
var doggies = ['dog1.png', 'dog2.png', 'dog3.png', 'dog4.png', 'dog5.png']
var doggypromises = doggies.map(getImage) // call getImage on each array element and return array of promises
var sequence = Promise.resolve()

doggypromises.forEach(function(curPromise){ // create sequence of promises to act on each one in succession
    sequence = sequence.then(function(){
        return curPromise
    }).then(function(url){
        var dog = document.createElement('img')
        dog.setAttribute('src', url)
        doggyplayground.appendChild(dog)
    }).catch(function(err){
        console.log(err + ' failed to load!')
    })
})
```

Demo (fetch images all at once in parallel, but show them sequentially):

Get images

Note that to create our sequence of promises this time, we iterate through the array of promises generated by `Array.map()`, and not the images array directly. This allows us to create the chain of promises without having to call `getImage()` each time again, which was done when we decided to fetch all images at once using `Array.map()` already.

In conclusion

JavaScript Promises offers an additional way to handle asynchronous tasks at a time where such tasks are becoming intimately woven into the fabric of any modern web site. Used in conjunction with a Polyfill, JavaScript Promises can be put to work today to make the whole affair more intuitive and manageable. Hopefully this tutorial has opened the doors to showing you how just to do that. Check out the following additional resources for more helpful info on the new feature:

- JavaScript Promises @ HTML5 Rocks
- JavaScript Promises Reference