Embracing Promises in JavaScript

*08 February 2015 by **Jack Franklin***

In this post we will look at how we can embrace promises to lead to much nicer code when working asynchronously with JavaScript. This post is not a full, in-depth exploration of Promises. For that, Jake Archibald's post on HTML5 Rocks has you covered. I highly recommend reading it.

Throughout this post I will be working using the es6-promise library, a polyfill for the native Promise implementation that will exist in ECMAScript 6. All my code examples will be run through Node.js, but they should behave identically when run in a browser environment. Whenever in the code you see `Promise`, this will be using the above polyfill, but if you're reading this in a world where promises are widely implemented in browsers, you should still find everything here works exactly the same.

Dealing with Errors

The first subject to tackle is that of error handling with promises. This was something that a lot of people have asked about and something that trips a lot of people up, understandably. Take a look at the below code. When I run this, what do you expect to be logged?

```
var someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // this will throw, x does not exist
    resolve(x + 2);
  });
};
```

```
someAsyncThing().then(function() {
  console.log('everything is great');
});
```

You might expect an error to be thrown, because x does not exist. That's what would happen if you wrote that code outside of a promise. However, running this code gives you absolutely nothing. Nothing is logged to the console, and no errors are thrown. Within a promise, any error that is thrown is swallowed up and treated as the promise rejecting. This means we have to catch the error to see it:

```
someAsyncThing().then(function() {
  console.log('everything is great');
}).catch(function(error) {
  console.log('oh no', error);
});
```

Now, running this gives:

```
oh no [ReferenceError: x is not defined]
```

You also need to be comfortable with how errors are caught in a chain of promises. Take the below example:

```
var someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // this will throw, x does not exist
    resolve(x + 2);
  });
};

var someOtherAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    reject('something went wrong');
  });
};

someAsyncThing().then(function() {
  return someOtherAsyncThing();
}).catch(function(error) {
```

```
  console.log('oh no', error);
});
```

Here we will still get `oh no [ReferenceError: x is not defined]`, because `someAsyncThing` rejected. However, if `someAsyncThing` resolves successfully, we'll still see the error when `someOtherAsyncThing` rejects:

```
var someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    var x = 2;
    resolve(x + 2);
  });
};

var someOtherAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    reject('something went wrong');
  });
};

someAsyncThing().then(function() {
  return someOtherAsyncThing();
}).catch(function(error) {
  console.log('oh no', error);
});
```

Now we get `oh no something went wrong`. When a promise rejects, the first `catch` in the chain following that is called.

Another important point is that there's nothing special about `catch`. It's just a method to register a handler for when a promise rejects. It doesn't stop further execution:

```
someAsyncThing().then(function() {
  return someOtherAsyncThing();
}).catch(function(error) {
  console.log('oh no', error);
}).then(function() {
  console.log('carry on');
});
```

Given the above code, once something rejects, `carry on` will then be logged to the screen. Of course, if the code within the `catch` throws an error, that is not the case:

```
someAsyncThing().then(function() {
  return someOtherAsyncThing();
}).catch(function(error) {
  console.log('oh no', error);
  // y is not a thing!
  y + 2;
}).then(function() {
  console.log('carry on');
});
```

Now the catch callback is executed, but `carry on` is not, because the catch callback threw an error. Notice again that there is no record of the error, it is not logged, nor is anything thrown visibly. If you were to add another `catch` onto the end, that `catch` function would run, because when a callback function throws, the next `catch` in the chain is called.

Chaining and Passing around Promises

This part is inspired by some work I did recently to add CSV exporting to our client-side application. In that case it was using the `$q` framework within an AngularJS application, but I have replicated it here so we can use it as an example.

The steps to export a CSV (the CSV itself is built in the browser using FileSaver) are as follows:

- Fetch the data from the API that will make up the CSV (this could mean multiple API requests)
- Pass that data into an object which does some editing of the data to make it CSV ready.
- Write the data to a CSV.
- Show the user a message confirming their CSV has been successfully created, or an error.

We won't go into the underlying specifics of how the code works, but I wanted to look at a high level how we used Promises to build a robust solution that handles errors too. In a complex operation like this, errors could easily occur at any stage

of the process (the API might be down, or the code parsing the data might throw an error, or the CSV might not save properly) and we found that with promises we could handle this really nicely, using a sensible combination of `then` and `catch`.

As you'll see we also end up chaining promises heavily. The chaining of promises is something that really makes them shine in my opinion, but it does take some getting used to - the way they work can be a little odd at first. Jake Archibald (yup, him again!) puts this best:

When you return something from a "then" callback, it's a bit magic. If you return a value, the next "then" is called with that value. However, if you return something promise-like, the next "then" waits on it, and is only called when that promise settles (succeeds/fails)

Again, for a real in-depth look at promises, I can't recommend this blog post highly enough.

Let's start with a really simple function that just returns some data. In a real application this would be a http call of some sort. In our case after 50ms, this promise will resolve with an array of users that we want to export to CSV:

```
var fetchData = function() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      resolve({
        users: [
          { name: 'Jack', age: 22 },
          { name: 'Tom', age: 21 },
          { name: 'Isaac', age: 21 },
          { name: 'Iain', age: 20 }
        ]
      });
    }, 50);
  });
}
```

Next, there's the function that prepares this data for the CSV. In this case all it actually does is immediately resolve with the data its given, but in a real application it would do more work:

```
var prepareDataForCsv = function(data) {
  return new Promise(function(resolve, reject) {
    // imagine this did something with the data
    resolve(data);
  });
};
```

There's something quite important to note here: in this example (and in the real app), none of the work `prepareDataForCsv` does is async. There's no need for this to be wrapped in a promise. But when a function exists as part of a larger chain, I've found it really beneficial to wrap it in a promise, because it means all your error handling can be done through promises. Else, you have to deal with error handling through promises in one area, but through good old `try {} catch` in another.

Finally, we have the function for writing to a CSV too:

```
var writeToCsv = function(data) {
  return new Promise(function(resolve, reject) {
    // write to CSV
    resolve();
  });
};
```

And now we can put them all together:

```
fetchData().then(function(data) {
  return prepareDataForCsv(data);
}).then(function(data) {
  return writeToCsv(data);
}).then(function() {
  console.log('your csv has been saved');
});
```

That's pretty succinct, and I think reads really well. It's clear what's going on and the order in which things happen. We can also tidy it up further though. If you have a function that just takes one argument, you can pass that directly to `then` rather than calling it from a callback function:

```
fetchData().then(prepareDataForCsv).then(writeToCsv).then(function() {
  console.log('your csv has been saved');
});
```

Bearing in mind how complex the underlying code is (at least, in the real application), the high level API reads really nicely. This is something I've come to really appreciate with promises, once you get used to writing them and working with them, you can end up with some really nice looking code that's easy to follow.

However, right now we don't have any error handling, but we can add it all with one extra piece of code:

```
fetchData().then(prepareDataForCsv).then(writeToCsv).then(function() {
  console.log('your csv has been saved');
}).catch(function(error) {
  console.log('something went wrong', error);
});
```

Because of how the chaining of promises and errors work, as discussed earlier, it means that just one `catch` at the end of the chain is guaranteed to catch any errors thrown along the way. This makes error handling really straight forward.

To demonstrate this, I'll change `prepareDataForCsv` so it rejects:

```
var prepareDataForCsv = function(data) {
  return new Promise(function(resolve, reject) {
    // imagine this did something with the data
    reject('data invalid');
  });
};
```

And now running the code logs the error. That's pretty awesome - `prepareDataForCsv` is right in the middle of our promise chain but we didn't have to do any extra work or trickery to deal with the error. Plus, the `catch` will not only catch errors that we trigger by making the promise reject, but also any that are thrown unexpectedly. This means even if a really unexpected edge case triggers a JS exception, the user will still have their error handled as expected.

Another approach that we've found to be very powerful is changing functions that expect some data to instead take a promise that will resolve to some data. Let's take `prepareDataForCsv` as the example:

```
var prepareDataForCsv = function(dataPromise) {
  return dataPromise().then(function(data) {
    return data;
  });
};
```

We've found this to be quite a nice pattern for tidying up code and keeping it more generic - it's often easier in an application where most of the work is async to pass promises around rather than waiting for them to resolve and pass the data.

With the above change, the new code looks like so:

```
prepareDataForCsv(fetchData).then(writeToCsv).then(function() {
  console.log('your csv has been saved');
}).catch(function(error) {
  console.log('something went wrong', error);
});
```

The beauty of this is that the error handling hasn't changed. `fetchData` could reject in some form, and the error will still be dealt with in the last catch. Once it clicks in your mind, you'll find promises really nice to work with and even nicer to handle errors with.

Recursion in Promises

One of the problems we had to deal with was that sometimes to fetch the data from our API, you might have to make multiple requests. This is because we paginate all our API requests, so if you need to get more data than can fit in one response, you need to make multiple. Thankfully our API tells you if there is more data to fetch, and in this section I'll explain how we used recursion in conjunction with promises to load all this data.

```
var count = 0;

var http = function() {
  if(count === 0) {
    count++;
    return Promise.resolve({ more: true, user: { name: 'jack', age: 22 } });
  } else {
    return Promise.resolve({ more: false, user: { name: 'isaac', age: 21 } });
  }
};
```

Firstly, we have `http`, which will serve as the fake HTTP calls to our API. (`Promise.resolve` just creates a promise that immediately resolves with whatever you give it). The first time I make a request, it's going to respond with a user but also the `more` flag set to true, which indicates there is more data to fetch (this isn't how the real life API responds, but it will do for the purposes of this post). The second time the request is made, it responds with a user but with the `more` flag set to `false`. Therefore to fetch all the data needed, we need to make two API calls. Let's write a function `fetchData` that can deal with this:

```
var fetchData = function() {
  var goFetch = function(users) {
    return http().then(function(data) {
      users.push(data.user);
      if(data.more) {
        return goFetch(users);
      } else {
        return users;
      }
    });
  }
```

```
  return goFetch([]);
};
```

`fetchData` itself does very little except define and then call another function, `goFetch`. `goFetch` takes an array of users in (the initial call to `goFetch` passes an empty array), and then calls `http()`, which resolves with some data. The new user that is returned is pushed onto the array of users, and then the function looks at the `data.more` field. If it's true, it calls itself again, passing in the new array of users. If it's false, and there is no more data to get, it just returns the array of users. The most important thing here and the reason this works is that at every stage something is `returned`. `fetchData` returns `goFetch`, which either returns itself or an array of users. It's the fact that everything returns itself that allows this recursive promise chain to be built up.

Conclusion

Promises are not going anywhere, and are going to become the standard approach for dealing with large amounts of asynchronous operations. However, I've found them to generally offer a lot of benefits when working on complex sequences of operations where some are sync, and others async. If you've not tried them yet I'd really recommend it on your next project.

If you'd like to discuss this post further or ask any questions, please tweet me.

LATEST POSTS

Using the HTML Webpack Plugin for generating HTML files

Setting up CSS Modules with React and Webpack

Screencast: Creating a React and Webpack Project

ABOUT THE SITE

The JavaScript Playground is a blog dedicated to producing top quality JavaScript tutorials on a number of subjects.

To see all past posts, take a look at the **archives**, or find out how you could **contribute**. You can also subscribe to the **RSS feed**, or **follow the site on Twiter**.