Lesson Monday

Intermediate JavaScript (/intermediate-javascript) / Asynchrony and APIs (/intermediate-javascript/asynchrony-and-apis) / Introduction to Promises

Text

ES2015 (also known as ES6) added a lot of great functionality to JavaScript. We've already learned about many of the most popular features, including `let` and `const`, classes, template literals, and destructuring. Now it's time to learn about promises, another great addition from ES6.

A promise allows us to wrap async code and then wait for the result of that code before moving on. In this lesson, we'll discuss why promises are so useful and how we can use them to tidy up our async code.

Promises have been a key concept in JavaScript development for quite some time, but up until ES6, they weren't native to JS. Instead, developers relied on promise libraries like Bluebird.js or used jQuery's `then()` method. Some developers still prefer to use promise libraries instead of ES6's native functionality because these libraries have more features or are potentially more efficient than ES6 promises.

Working with Promises

Before we dive into the async aspects of promises, it's important to cover some basic things about what a promise is.

First of all, a promise is just an object that inherits a few prototypal methods. Remember all that work we did just a few sections ago on JavaScript objects? Well, we can apply that knowledge now, which means we already know something about promises. At this point, we know that the first thing we have to do when we are working with JavaScript objects is create an instance of the object we want to work with, just like we did with the `XMLHttpRequest` object. Once we do that, we'll be able to use the object as well as any methods that are reserved for that type of object.

So on the most basic level, that's all a promise is: an object.

```
let promise = new Promise();
```

A promise generally takes a function as an argument. This function will hold our async code and it has two parameters, `resolve` and `reject`. So let's update our object a little bit.

```
let promise = new Promise(function(resolve, reject) {
  // async code goes here
});
```

We always need to include the `resolve` and `reject` parameters. Why? Well, we have to decide what determines whether a promise is resolved or rejected. JavaScript can't just look at a function that's making an API call and figure out what it's supposed to do with it. In the case of our API call, our promise needs to look something like this:

```
// Some pseudocode added... these variable names are just t
o demonstrate how a promise works.

let promise = new Promise(function(resolve, reject) {
  if (apiCallSuccessful) {
    resolve(data)
  } else {
    reject(message)
  }
});
```

In the example above, we've added a conditional. If `apiCallSuccessful` is `true`, our promise will `resolve`. We'll pass the `data` from the API call along. Otherwise, our promise will `reject` and we'll pass a `message` along.

Think of it as being a little bit like a gift. It won't be ready until your birthday. When it's opened, it will be resolved ("I like it!") or rejected ("I don't like it."). Our response will then be passed along to be used where it's needed.

So if we look at the example above, the `promise` variable is a `Promise` object that holds a function with two parameters: `resolve` and `reject`. When this function is actually called, the promise can be in one of three states:

- **Pending**: The object's initial state. A pending operation has been started but hasn't been completed yet.
- **Fulfilled**: A promise is fulfilled when the operation has been successfully completed.
- **Rejected**: A promise is rejected when the operation fails.

Once again, *we* need to determine what it means for the promise to succeed (fulfilled) or fail (rejected).

It's also important to remember that **a promise can only be resolved once**. Once it's rejected or fulfilled, there's no going back. For instance, you can't try to turn a rejected promise into a fulfilled one. Instead, you'd have to create another promise and try again.

Let's use one more analogy to explain the process. Imagine you're waiting to renew your driver's license at the DMV. When you go in, you get a piece of paper with a number on it. You wait until your number is called and then you go to the counter to renew your license.

That piece of paper is similar to a promise. It represents an appointment you'll have in the future, but that appointment doesn't exist yet. While you're waiting for your number to be called, the promise is **pending**. The promise will either be **fulfilled** (driver's license renewed...yay!) or **rejected** (you forgot to bring the right documents with which to renew your license...).

Once the promise is either fulfilled or rejected, it is complete and becomes immutable. An **immutable** value can't be changed. A promise is a one-off situation and we can't use it again. Returning to the analogy of getting a license, once that ticket is resolved, it can't be used again. If you want to do something else at the DMV, you'll have to get a new ticket (which represents a new promise).

So how do we actually access and deal with this data and determine whether it's fulfilled or rejected?

Promise objects have just three methods — we only need to know about two of them: `Promise.prototype.then()` and `Promise.prototype.catch()`. As you might guess, the latter method is used to handle errors that come up in promises.

We can do the following with our `promise` variable:

```
promise.then(function(response) {
  doSomething(response);
});
```

When we call `Promise.prototype.then()`, the function inside the promise is triggered and the promise is now in a **pending** state. Any code passed into `Promise.prototype.then()` will *not* run yet. That code won't be triggered until the promise is fulfilled or rejected. If our promise is resolved, our callback will be triggered, and the `doSomething` function will be called. As we can see here, we'll still be working with callbacks even when we use promises.

What happens if our promise is rejected? Well, we aren't handling that yet. We have to add an additional function to handle the rejection:

```
promise.then(function(success) {
  doSomething(success);
}, function(failure) {
  itFailed(failure);
});
```

Note that we put a `,` after the first closing curly bracket *and then* we add another function for rejection. The final closing parens then comes *after* the second function. Why does this look so weird?

Well, `Promise.prototype.then()` takes up to two callback functions as arguments. This is a really cool thing about JavaScript and something that can be hard for beginners to wrap their heads around. Functions are *first class citizens*, which means they can be passed around as variables and arguments. We will be discussing that in greater detail once we get to the React course.

So just remember that the first argument to `Promise.prototype.then()` is the function that runs if the promise is fulfilled and the second (optional) argument is the function that runs if the promise is rejected. You might wonder why the second function is optional — well, as a developer, you might want to do nothing if the promise is rejected. In this course section, we'll always include some form of error handling.

Another important thing to note about the example above: `success` and `failure` are just parameters of the functions and we can call them whatever we want. JavaScript doesn't care. We could call them both `response` but that's not very descriptive. It's common to call the parameter for a fulfilled promise `response` and the parameter for a rejected promise `error` — because when a promise is rejected, it often means there was an error. But once again, remember that you get to choose what they're called — just make sure that parameters, like other variables, have concise, descriptive names.

A Demonstration

So now that we know the basics of what a promise looks like, let's write a very silly one to illustrate exactly how they work. The function in this promise will generate a random number — either 0 or 1. If the value is 1, the promise will be resolved. If the value is 0, the promise will be rejected.

This is a silly use case because we are using a promise to handle synchronous code. In the real world, we'd never do that — promises are for handing async code. However, the point of this example is to demonstrate how promises work before we introduce more complex async code. This way, we can see exactly how to create a promise, use it, and then handle its response.

Try out the following example in the DevTools console:

```
let promise = new Promise(function(resolve, reject) {
    const value = Math.floor(Math.random() * Math.floor(2))
    if (value === 1) {
        resolve("The value is 1!");
    } else {
        reject("The value is 0.");
    }
});

promise.then(function(resolvedResponse) {
    console.log("resolved!")
    console.log(resolvedResponse);
}, function(rejectedResponse) {
    console.log("rejected!");
    console.log(rejectedResponse);
});
```

First, we create a new promise and store it in a `promise` variable. `Math.floor(Math.random() * Math.floor(2))` just randomly generates either the value 0 or 1. Then, we use a conditional to determine when the promise should be considered resolved or rejected. As we can see, we determine the exact conditions. It doesn't matter if it's a simple synchronous example like the one above or really complex async code.

Next, we use `Promise.prototype.then()` to determine how to handle both a `resolvedResponse` and a `rejectedResponse`. Note the descriptive parameter names — maybe not as concise as we'd like but solid for educational purposes.

Next, we use `console.log()` to let us know whether the promise was resolved or rejected. We also log the message that's passed from the promise into `Promise.prototype.then()`.

Try it out in the console. If the promise is resolved, we'll get the following:

```
resolved!
The value is 1!
Promise {<fulfilled>: undefined}
```

If it's rejected, we'll get this:

```
rejected!
The value is 0.
Promise {<fulfilled>: undefined}
```

But what's that third line? We didn't `console.log()` that, did we? `Promise.prototype.then()` returns another promise — this is the *return* of our method. This is absolutely essential because it allows us to chain multiple promises together, waiting until one resolves before starting the next one.

Once again, don't ever use promises for synchronous code — it's bad practice. We've already stated this but it's worth emphasizing. The example above is for learning purposes only.

One other thing — if you are trying the code snippet above multiple times without refreshing the console, you might be wondering what exactly is happening. Can't a promise be resolved only once? Yes. But each time we call the code snippet above, we are creating a *new* promise variable with a new promise in it — `let` allows us to do that. However, if we were just to run this part of the code again:

```
promise.then(function(resolvedResponse) {
    console.log("resolved!")
    console.log(resolvedResponse);
}, function(rejectedResponse) {
    console.log("rejected!");
    console.log(rejectedResponse);
});
```

The promise will be fulfilled once, then each time we run the code again, we'd get the exact same response — the already fulfilled promise.

Summary

Now that we've gone through an example and covered promises in detail, let's summarize how they work:

- We can wrap our async code in a promise.
- Next, we can tell our function when it should resolve or reject, giving us fine-grained control over how JavaScript should handle our async code.
- Finally, we can use `Promise.prototype.then()` to run code once the async operation is complete — no need to keep tabs on the promise! JavaScript will do that for us.
- `Promise.prototype.then()` takes up to two functions as arguments — the first (required) function determines what happens if a promise is fulfilled while the second (optional) function determines what happens if the promise is rejected.
- We can even chain promises together because `Promise.prototype.then()` itself returns a promise. We'll learn more about that later in this section, though it's not required to do any chaining for the independent project.

There are a few other useful methods related to promises as well, none of which you need to use in this section. However, it's good to know about them and you may want to try them out during a class

project. For instance, we can use `Promise.all()` to work with multiple promises:

```
Promise.all([promise1, promise2, promise3]);
```

`Promise.all()` wraps the enclosed promises into a single giant promise that only resolves after each of the included promises are resolved.

There are a few other methods such as `Promise.race()` and `Promise.catch()`. Check out Mozilla's documentation on promises (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) to learn more.

Now that we've explained the basics of how a promise works, we're ready to add one to our code. In the next lesson, we'll use a promise to handle the results of an API call.

Previous (/intermediate-javascript/asynchrony-and-apis/tools-for-handling-async-code)
Next (/intermediate-javascript/asynchrony-and-apis/promises-with-api-calls)