You Don't Know JS: ES6 & Beyond

Chapter 4: Async Flow Control

It's no secret if you've written any significant amount of JavaScript that asynchronous programming is a required skill. The primary mechanism for managing asynchrony has been the function callback.

However, ES6 adds a new feature that helps address significant shortcomings in the callbacks-only approach to async: *Promises*. In addition, we can revisit generators (from the previous chapter) and see a pattern for combining the two that's a major step forward in async flow control programming in JavaScript.

Promises

Let's clear up some misconceptions: Promises are not about replacing callbacks. Promises provide a trustable intermediary -- that is, between your calling code and the async code that will perform the task -- to manage callbacks.

Another way of thinking about a Promise is as an event listener, on which you can register to listen for an event that lets you know when a task has completed. It's an event that will only ever fire once, but it can be thought of as an event nonetheless.

Promises can be chained together, which can sequence a series of asychronously completing steps. Together with higher-level abstractions like the `all(..)` method (in classic terms, a "gate") and the `race(..)` method (in classic terms, a "latch"), promise chains provide a mechanism for async flow control.

Yet another way of conceptualizing a Promise is that it's a *future value*, a time-independent container wrapped around a value. This container can be reasoned about identically whether the underlying value is final or not. Observing the resolution of a Promise extracts this value once available. In other words, a Promise is said to be the async version of a sync function's return value.

A Promise can only have one of two possible resolution outcomes: fulfilled or rejected, with an optional single value. If a Promise is fulfilled, the final value is called a fulfillment. If it's rejected, the final value is called a reason (as in, a "reason for rejection"). Promises can only be resolved (fulfillment or rejection) *once*. Any further attempts to fulfill or reject are simply ignored. Thus, once a Promise is resolved, it's an immutable value that cannot be changed.

Clearly, there are several different ways to think about what a Promise is. No single perspective is fully

sufficient, but each provides a separate aspect of the whole. The big takeaway is that they offer a significant improvement over callbacks-only async, namely that they provide order, predictability, and trustability.

Making and Using Promises

To construct a promise instance, use the `Promise(..)` constructor:

```
var p = new Promise( function pr(resolve,reject){
    // ..
} );
```

The `Promise(..)` constructor takes a single function (`pr(..)`), which is called immediately and receives two control functions as arguments, usually named `resolve(..)` and `reject(..)`. They are used as:

- If you call `reject(..)`, the promise is rejected, and if any value is passed to `reject(..)`, it is set as the reason for rejection.
- If you call `resolve(..)` with no value, or any non-promise value, the promise is fulfilled.
- If you call `resolve(..)` and pass another promise, this promise simply adopts the state -- whether immediate or eventual -- of the passed promise (either fulfillment or rejection).

Here's how you'd typically use a promise to refactor a callback-reliant function call. If you start out with an `ajax(..)` utility that expects to be able to call an error-first style callback:

```
function ajax(url,cb) {
    // make request, eventually call `cb(..)`
}

// ..

ajax( "http://some.url.1", function handler(err,contents){
    if (err) {
        // handle ajax error
    }
    else {
        // handle `contents` success
    }
} );
```

You can convert it to:

```
function ajax(url) {
    return new Promise( function pr(resolve,reject){
        // make request, eventually call
        // either `resolve(..)` or `reject(..)`
    } );
}

// ..

ajax( "http://some.url.1" )
.then(
    function fulfilled(contents){
        // handle `contents` success
    },
    function rejected(reason){
        // handle ajax error reason
    }
);
```

Promises have a `then(..)` method that accepts one or two callback functions. The first function (if present) is treated as the handler to call if the promise is fulfilled successfully. The second function (if present) is treated as the handler to call if the promise is rejected explicitly, or if any error/exception is caught during resolution.

If one of the arguments is omitted or otherwise not a valid function -- typically you'll use `null` instead -- a default placeholder equivalent is used. The default success callback passes its fulfillment value along and the default error callback propagates its rejection reason along.

The shorthand for calling `then(null,handleRejection)` is `catch(handleRejection)`.

Both `then(..)` and `catch(..)` automatically construct and return another promise instance, which is wired to receive the resolution from whatever the return value is from the original promise's fulfillment or rejection handler (whichever is actually called). Consider:

```
ajax( "http://some.url.1" )
.then(
    function fulfilled(contents){
        return contents.toUpperCase();
    },
    function rejected(reason){
        return "DEFAULT VALUE";
    }
)
.then( function fulfilled(data){
    // handle data from original promise's
    // handlers
} );
```

In this snippet, we're returning an immediate value from either `fulfilled(..)` or `rejected(..)`, which then is received on the next event turn in the second `then(..)`'s `fulfilled(..)`. If we instead return a new promise, that new promise is subsumed and adopted as the resolution:

```
ajax( "http://some.url.1" )
.then(
    function fulfilled(contents){
        return ajax(
            "http://some.url.2?v=" + contents
        );
    },
    function rejected(reason){
        return ajax(
            "http://backup.url.3?err=" + reason
        );
    }
)
.then( function fulfilled(contents){
    // `contents` comes from the subsequent
    // `ajax(..)` call, whichever it was
} );
```

It's important to note that an exception (or rejected promise) in the first `fulfilled(..)` will *not* result in the first `rejected(..)` being called, as that handler only responds to the resolution of the first original promise. Instead, the second promise, which the second `then(..)` is called against, receives that rejection.

In this previous snippet, we are not listening for that rejection, which means it will be silently held onto for future observation. If you never observe it by calling a `then(..)` or `catch(..)`, then it will go unhandled. Some browser developer consoles may detect these unhandled rejections and report them, but this is not

reliably guaranteed; you should always observe promise rejections.

Note: This was just a brief overview of Promise theory and behavior. For a much more in-depth exploration, see Chapter 3 of the *Async & Performance* title of this series.

Thenables

Promises are genuine instances of the `Promise(..)` constructor. However, there are promise-like objects called *thenables* that generally can interoperate with the Promise mechanisms.

Any object (or function) with a `then(..)` function on it is assumed to be a thenable. Any place where the Promise mechanisms can accept and adopt the state of a genuine promise, they can also handle a thenable.

Thenables are basically a general label for any promise-like value that may have been created by some other system than the actual `Promise(..)` constructor. In that perspective, a thenable is generally less trustable than a genuine Promise. Consider this misbehaving thenable, for example:

```
var th = {
    then: function thener( fulfilled ) {
        // call `fulfilled(..)` once every 100ms forever
        setInterval( fulfilled, 100 );
    }
};
```

If you received that thenable and chained it with `th.then(..)`, you'd likely be surprised that your fulfillment handler is called repeatedly, when normal Promises are supposed to only ever be resolved once.

Generally, if you're receiving what purports to be a promise or thenable back from some other system, you shouldn't just trust it blindly. In the next section, we'll see a utility included with ES6 Promises that helps address this trust concern.

But to further understand the perils of this issue, consider that *any* object in *any* piece of code that's ever been defined to have a method on it called `then(..)` can be potentially confused as a thenable -- if used with Promises, of course -- regardless of if that thing was ever intended to even remotely be related to Promise-style async coding.

Prior to ES6, there was never any special reservation made on methods called `then(..)`, and as you can imagine there's been at least a few cases where that method name has been chosen prior to Promises ever showing up on the radar screen. The most likely case of mistaken thenable will be async libraries that use `then(..)` but which are not strictly Promises-compliant -- there are several out in the wild.

The onus will be on you to guard against directly using values with the Promise mechanism that would be

incorrectly assumed to be a thenable.

`Promise` API

The `Promise` API also provides some static methods for working with Promises.

`Promise.resolve(..)` creates a promise resolved to the value passed in. Let's compare how it works to the more manual approach:

```
var p1 = Promise.resolve( 42 );

var p2 = new Promise( function pr(resolve){
    resolve( 42 );
} );
```

`p1` and `p2` will have essentially identical behavior. The same goes for resolving with a promise:

```
var theP = ajax( .. );

var p1 = Promise.resolve( theP );

var p2 = new Promise( function pr(resolve){
    resolve( theP );
} );
```

Tip: `Promise.resolve(..)` is the solution to the thenable trust issue raised in the previous section. Any value that you are not already certain is a trustable promise -- even if it could be an immediate value -- can be normalized by passing it to `Promise.resolve(..)`. If the value is already a recognizable promise or thenable, its state/resolution will simply be adopted, insulating you from misbehavior. If it's instead an immediate value, it will be "wrapped" in a genuine promise, thereby normalizing its behavior to be async.

`Promise.reject(..)` creates an immediately rejected promise, the same as its `Promise(..)` constructor counterpart:

```
var p1 = Promise.reject( "Oops" );

var p2 = new Promise( function pr(resolve,reject){
    reject( "Oops" );
} );
```

While `resolve(..)` and `Promise.resolve(..)` can accept a promise and adopt its state/resolution, `reject(..)` and `Promise.reject(..)` do not differentiate what value they receive. So, if you reject

with a promise or thenable, the promise/thenable itself will be set as the rejection reason, not its underlying value.

`Promise.all([ .. ])` accepts an array of one or more values (e.g., immediate values, promises, thenables). It returns a promise back that will be fulfilled if all the values fulfill, or reject immediately once the first of any of them rejects.

Starting with these values/promises:

```
var p1 = Promise.resolve( 42 );
var p2 = new Promise( function pr(resolve){
    setTimeout( function(){
        resolve( 43 );
    }, 100 );
} );
var v3 = 44;
var p4 = new Promise( function pr(resolve,reject){
    setTimeout( function(){
        reject( "Oops" );
    }, 10 );
} );
```

Let's consider how `Promise.all([ .. ])` works with combinations of those values:

```
Promise.all( [p1,p2,v3] )
.then( function fulfilled(vals){
    console.log( vals );            // [42,43,44]
} );

Promise.all( [p1,p2,v3,p4] )
.then(
    function fulfilled(vals){
        // never gets here
    },
    function rejected(reason){
        console.log( reason );      // Oops
    }
);
```

While `Promise.all([ .. ])` waits for all fulfillments (or the first rejection), `Promise.race([ .. ])` waits only for either the first fulfillment or rejection. Consider:

```
// NOTE: re-setup all test values to
// avoid timing issues misleading you!

Promise.race( [p2,p1,v3] )
.then( function fulfilled(val){
    console.log( val );             // 42
} );

Promise.race( [p2,p4] )
.then(
    function fulfilled(val){
        // never gets here
    },
    function rejected(reason){
        console.log( reason );      // Oops
    }
);
```

Warning: While `Promise.all([])` will fulfill right away (with no values), `Promise.race([])` will hang forever. This is a strange inconsistency, and speaks to the suggestion that you should never use these methods with empty arrays.

Generators + Promises

It *is* possible to express a series of promises in a chain to represent the async flow control of your program. Consider:

```
step1()
.then(
    step2,
    step1Failed
)
.then(
    function step3(msg) {
        return Promise.all( [
            step3a( msg ),
            step3b( msg ),
            step3c( msg )
        ] )
    }
)
.then(step4);
```

However, there's a much better option for expressing async flow control, and it will probably be much more preferable in terms of coding style than long promise chains. We can use what we learned in Chapter 3 about generators to express our async flow control.

The important pattern to recognize: a generator can yield a promise, and that promise can then be wired to resume the generator with its fulfillment value.

Consider the previous snippet's async flow control expressed with a generator:

```
function *main() {

    try {
        var ret = yield step1();
    }
    catch (err) {
        ret = yield step1Failed( err );
    }

    ret = yield step2( ret );

    // step 3
    ret = yield Promise.all( [
        step3a( ret ),
        step3b( ret ),
        step3c( ret )
    ] );

    yield step4( ret );
}
```

On the surface, this snippet may seem more verbose than the promise chain equivalent in the earlier snippet. However, it offers a much more attractive -- and more importantly, a more understandable and reason-able -- synchronous-looking coding style (with `=` assignment of "return" values, etc.) That's especially true in that `try..catch` error handling can be used across those hidden async boundaries.

Why are we using Promises with the generator? It's certainly possible to do async generator coding without Promises.

Promises are a trustable system that uninverts the inversion of control of normal callbacks or thunks (see the *Async & Performance* title of this series). So, combining the trustability of Promises and the synchronicity of code in generators effectively addresses all the major deficiencies of callbacks. Also, utilities like `Promise.all([ .. ])` are a nice, clean way to express concurrency at a generator's single `yield` step.

So how does this magic work? We're going to need a *runner* that can run our generator, receive a `yield`ed promise, and wire it up to resume the generator with either the fulfillment success value, or throw an error into the generator with the rejection reason.

Many async-capable utilities/libraries have such a "runner"; for example, `Q.spawn(..)` and my asynquence's `runner(..)` plug-in. But here's a stand-alone runner to illustrate how the process works:

```
function run(gen) {
    var args = [].slice.call( arguments, 1), it;

    it = gen.apply( this, args );

    return Promise.resolve()
        .then( function handleNext(value){
            var next = it.next( value );

            return (function handleResult(next){
                if (next.done) {
                    return next.value;
                }
                else {
                    return Promise.resolve( next.value )
                        .then(
                            handleNext,
                            function handleErr(err) {
                                return Promise.resolve(
                                    it.throw( err )
                                )
                                .then( handleResult );
                            }
                        );
                }
            })( next );
        } );
}
```

Note: For a more prolifically commented version of this utility, see the *Async & Performance* title of this series. Also, the run utilities provided with various async libraries are often more powerful/capable than what we've shown here. For example, asynquence's `runner(..)` can handle `yield`ed promises, sequences, thunks, and immediate (non-promise) values, giving you ultimate flexibility.

So now running `*main()` as listed in the earlier snippet is as easy as:

```
run( main )
.then(
    function fulfilled(){
        // `*main()` completed successfully
    },
    function rejected(reason){
        // Oops, something went wrong
    }
);
```

Essentially, anywhere that you have more than two asynchronous steps of flow control logic in your program, you can *and should* use a promise-yielding generator driven by a run utility to express the flow control in a synchronous fashion. This will make for much easier to understand and maintain code.

This yield-a-promise-resume-the-generator pattern is going to be so common and so powerful, the next version of JavaScript after ES6 is almost certainly going to introduce a new function type that will do it automatically without needing the run utility. We'll cover `async function`s (as they're expected to be called) in Chapter 8.

Review

As JavaScript continues to mature and grow in its widespread adoption, asynchronous programming is more and more of a central concern. Callbacks are not fully sufficient for these tasks, and totally fall down the more sophisticated the need.

Thankfully, ES6 adds Promises to address one of the major shortcomings of callbacks: lack of trust in predictable behavior. Promises represent the future completion value from a potentially async task, normalizing behavior across sync and async boundaries.

But it's the combination of Promises with generators that fully realizes the benefits of rearranging our async flow control code to de-emphasize and abstract away that ugly callback soup (aka "hell").

Right now, we can manage these interactions with the aide of various async libraries' runners, but JavaScript is eventually going to support this interaction pattern with dedicated syntax alone!