asynquence: The Promises You Don't Know Yet (Part 1)

By Kyle Simpson on June 17, 2014 4

This is a multi-part blog post series highlighting the capabilities of asynquence, a promises-based flow-control abstraction utility.

`on("before", start)`

Normally, my blog posts (and training workshops, for that matter!) are intended to teach something, and in the process I highlight projects that I've written to explore and experiment in that area. I find that to be an effective aid to teaching.

However, this blog post series is going to be, unapologetically, quite a bit more obviously a promotion of one of my most important and ambitious projects: asynquence. The topic underlying? Promises and async flow-control.

But I've already written a detailed multi-part blog post series that teaches all about promises and the async issues they solve. I strongly suggest you read those posts first, if you're looking for deeper understanding of the topic, before you indulge my present ramblings on *asynquence*.

Why am I hard-promoting *asynquence* here in such an obvious self-horn-tooting way? Because I think it uniquely provides an accessibility to the topic of async flow-control and promises that you didn't realize you needed.

asynquence isn't rockstar popular or talked about by all the cool kids in-crowd. It doesn't have thousands of stars on github or millions of npm downloads. But I passionately believe if you spend some time digging into what it can do, **and how it does it**, you will find some missing clarity and relief from the tedium that sets in with other async utilities.

This is a long post, and there's more than one post in this series. There's a whole lot to show off. Make sure to take some time to digest everything I'm about to show you. Your code will thank you... **eventually**.

At a max size of well under **5k** (minzipped) for everything (including optional plugins!), I think you'll see *asynquence* packs quite a punch for its modest byte count.

Promise Or Abstraction?

The first thing to note is that despite some API similarities, *asynquence* creates an abstraction layer on top of promises, which I call **sequences**. That's where the weird name comes from: **async** + **sequence** = **asynquence**.

A sequence is a series of automatically created *and* chained promises. The promises are hidden under the API surface, so that you don't have to create or chain them in the general/simple cases. That's so that you can take advantage of promises with much less boilerplate cruft.

Of course, to make integration of *asynquence* into your project easier, a sequence can both consume a standard thenable/promise from some other vending, and it can also vend a standard ES6 promise at any step of a sequence. So you have ultimate freedom to sling promises around or enjoy the simplicity of the sequence abstractions.

Each step of a sequence can be arbitrarily simple, like an immediately fulfilled promise, or arbitrarily complex, like a nested tree of sequences, etc. *asynquence* provides a wide array of abstraction helpers to invoke at each step, like `gate(..)` (the same as native Promises `Promise.all(..)`), which runs 2 or more "segments" (sub-steps) in parallel, and waits for all of them to complete (in any order) before proceeding on the main sequence.

You construct the async flow-control expression for a particular task in your program by chaining together however many steps in the sequence as are applicable. Just like with promises, each step can either succeed (and pass along any number of success messages) or it can fail (and pass along any number of reason messages).

In this blog post, I detail a whole host of limitations implied when *all* you have are promises, and make the case for the power and utility of abstractions. I make the claim there that *asynquence* frees you from all these limitations, so this blog post series proves such a claim.

Basics

You're certainly more interested in seeing code than reading me ramble on about code. So, let's start by illustrating the basics of *asynquence*:

```
ASQ(function step1(done){
    setTimeout(function(){
        done( "Hello" );
    },100);
})
.then(function step2(done,msg){
    setTimeout(function(){
        done( msg.toUpperCase()) ;
    },100);
})
.gate(
    // these two segments '3a' and '3b' run in parallel!
    function step3a(done,msg) {
        setTimeout(function(){
            done( msg + " World" );
            // if you wanted to fail this segment,
            // you would call `done.fail(..)` instead
        },500);
    },
    function step3b(done,msg) {
        setTimeout(function(){
            done( msg + " Everyone" );
        },300);
    }
)
.then(function step4(done,msg1,msg2){
    console.log(msg1,msg2); // "Hello World"  "Hello Everyone"
})
.or(function oops(err){
    // if any error occurs anywhere in the sequence,
    // you'll get notified here
});
```

With just that snippet, you see a pretty good depiction of what *asynquence* was originally designed to do. For each step, a promise is created for you, and you are provided with the trigger (which I like to always call `done` for simplicity), which you just need to call now or at some point later.

If an error occurs, or if you want to fail a step by calling `done.fail(..)`, the rest of the sequence path is abandoned and any error handlers are notified.

Errors Not Lost

With promises, if you fail to register an error handler, the error stays silently buried inside the promise for some future consumer to observe. This along with how promise-chaining works leads to all manner of confusion and nuance.

If you read those discussions, you'll see I make the case that promises have an "opt-in" model for error handling, so if you forget to opt-in, you fail silently. This is what we disaffectionately call a **"pit of failure"**.

asynquence reverses this paradigm, creating a **"pit of success"**. The default behavior of a sequence is to report any error (intentional or accidental) in a global exception (in your dev console), rather than swallow it. Of course, reporting it in a global exception doesn't erase the sequences's state, so it can still be programmatically observed later as usual.

You can "opt-out" of this global error reporting in one of two ways: (1) register at least one `or` error handler on the sequence; (2) call `defer()` on the sequence, which signals that you intend to register an error handler later.

Furthermore, if sequence **A** is consumed by (combined into) another sequence **B**, `A.defer()` is automatically called, shifting the error handling burden to **B**, just like you'd want and expect.

With promises, you have to work hard to make sure you catch errors, and if you fall short, you'll be confused as they'll be hidden in subtle, hard-to-find ways. With *asynquence* sequences, you have to work hard to **NOT** catch errors. *asynquence* makes your error handling easier and saner.

Messages

With promises, the resolution (success or failure) can only happen with one distinct value. It's up to you to wrap multiple values into a container (object, array, etc) should you need to pass more than one value along.

asynquence assumes you need to pass any number of parameters (either success or failure), and automatically handles the wrapping/un-wrapping for you, in the way you'd most naturally expect:

```
ASQ(function step1(done){
    done( "Hello", "World" );
})
.then(function step2(done,msg1,msg2){
    console.log(msg1,msg2); // "Hello"  "World"
});
```

In fact, messages can easily be injected into a sequence:

```
ASQ( "Hello", "World" )
.then(function step1(done,msg1,msg2){
    console.log(msg1,msg2); // "Hello"  "World"
})
.val( 42 )
.then(function(done,msg){
    console.log(msg); // 42
});
```

In addition to injecting success messages into a sequence, you can also create an automatically failed sequence (that is, messages that are error reasons):

```
// make a failed sequence!
ASQ.failed( "Oops", "My bad" )
.then(..) // will never run!
.or(function(err1,err2){
    console.log(err1,err2); // "Oops"  "My bad"
});
```

Halting Problem

With promises, if you have say 4 promises chained, and at step 2 you decide you don't want 3 and 4 to occur, you're only option is to throw an error. Sometimes this makes sense, but more often it's rather limiting.

You'd probably like to just be able to cancel any promise. But, if a promise itself can be aborted/canceled from the outside, that actually violates the important principle of trustably externally immutable state.

```
var sq = ASQ(function step1(done){
    done(..);
})
.then(function step2(done){
    done.abort();
})
.then(function step3(done){
    // never called
});

// or, later:
sq.abort();
```

Aborting/canceling shouldn't exist at the promise level, but in the abstraction on layer on top of them. So, **asynquence** lets you call `abort()` on a sequence, or at any step of a sequence on the trigger. To the extent possible, the rest of the sequence will be completely abandoned (side effects from async tasks cannot be prevented, obviously!).

Sync Steps

Despite much of our code being async in nature, there are always tasks which are fundamentally synchronous. The most common example is performing a data extraction or transformation task in the middle of a sequence:

```
ASQ(function step1(done){
    done( "Hello", "World" );
})
// Note: `val(..)` doesn't receive a trigger!
.val(function step2(msg1,msg2){
    // sync data transformation step
    // `return` passes sync data messages along
    // `throw` passes sync error messages along
    return msg1 + " " + msg2;
})
.then(function step3(done,msg){
    console.log(msg); // "Hello World"
});
```

The `val(..)` step method automatically advances the promise for that step after you `return` (or `throw` for errors!), so it doesn't pass you a trigger. You use `val(..)` for any synchronous step in the middle of the sequence.

Callbacks

Especially in node.js, (error-first style) callbacks are the norm, and promises are the new kid on the block. This means that you'll almost certainly be integrating them into your async sequences code. When you call some utility that expects an error-first style callback, *asynquence* provides `errfcb()` to create one for you, automatically wired into your sequence:

```
ASQ(function step1(done){
    // `done.errfcb` is already an error-first
    // style callback you can pass around, just like
    // `done` and `done.fail`.
    doSomething( done.errfcb );
})
.seq(function step2(){
    var sq = ASQ();

    // calling `sq.errfcb()` creates an error-first
    // style callback you can pass around.
    doSomethingElse( sq.errfcb() );

    return sq;
})
.then(..)
..
```

Note: `done.errfcb` and `sq.errfcb()` differ in that the former is already created so you don't need to `()` invoke it, whereas the latter needs to be called to make a callback wired to the sequence at that point.

Some other libraries provide methods to wrap other function calls, but this seems too intrusive for *asynquence*'s design philosophy. So, to make a sequence-producing method wrapper, make your own, like this:

```
// in node.js, using `fs` module,
// make a suitable sequence-producing
// wrapper for `fs.write(..)`
function fsWrite(filename,data) {
    var sq = ASQ();
    fs.write( filename, data, sq.errfcb() );
    return sq;
}

fsWrite( "meaningoflife.txt", "42" )
.val(function step2(){
    console.log("Phew!");
})
.or(function oops(err){
    // file writing failed!
});
```

Promises, Promises

asynquence should be good enough at async flow-control that for nearly all your needs, it's all the utility you need. But the reality is, promises themselves will still show up in your program. *asynquence* makes it easy to go from promise to sequence to promise as you see fit.

```
var sq = ASQ()
.then(..)
.promise( doTaskA() )
.then(..)
..

// doTaskB(..) requires you to pass
// a normal promise to it!
doTaskB( sq.toPromise() );
```

`promise(..)` consumes one or more standard thenables/promises vended from elsewhere (like inside `doTaskA()`) and wires it into the sequence. `toPromise()` vends a new promise forked from that point in the sequence. All success and error message streams flow in and out of promises exactly as you'd expect.

Sequences + Sequences

The next thing you'll almost certainly find yourself doing regularly is creating multiple sequences and wiring them together.

For example:

```
var sq1 = doTaskA();
var sq2 = doTaskB();
var sq3 = doTaskC();

ASQ()
.gate(
    sq1,
    sq2
)
.then( sq3 )
.seq( doTaskD )
.then(function step4(done,msg){
    // Tasks A, B, C, and D are done
});
```

`sq1` and `sq2` are separate sequences, so they can be wired directly in as `gate(..)` segments, or as `then(..)` steps. There's also `seq(..)` which can either accept a sequence, or more commonly, a function that it will call to produce a sequence. In the above snippet, `function doTaskD(msg1,..) { .. return sq; }` would be the general signature. It receives the messages from the previous step (`sq3`), and is expected to return a new sequence as step 3.

Note: This is another API sugar where *asynquence* can shine, because with a promise-chain, to wire in another promise, you have to do the uglier:

```
pr1
.then(..)
.then(function(){
    return pr2;
})
..
```

As seen above, *asynquence* just accepts sequences directly into `then(..)`, like:

```
sq1
.then(..)
.then(sq2)
..
```

Of course, if you find yourself needing to manually wire in a sequence, you can do so with `pipe(..)`:

```
ASQ()
.then(function step1(done){
    // pipe the sequence returned from `doTaskA(..)`
    // into our main sequence
    doTaskA(..).pipe( done );
})
.then(function step2(done,msg){
    // Task A succeeded
})
.or(function oops(err){
    // errors from anywhere, even inside of the
    // Task A sequence
});
```

As you'd reasonably expect, in all these variations, both success and error message streams are piped, so errors propagate up to the outermost sequence naturally and automatically. That doesn't stop you from manually listening to and handling errors at any level of sub-sequence, however.

```
ASQ()
.then(function step1(done){
    // instead of `pipe(..)`, manually send
    // success message stream along, but handle
    // errors here
    doTaskA()
    .val(done)
    .or(function taskAOops(err){
        // handle Task A's errors here only!
    });
})
.then(function step2(done,msg){
    // Task A succeeded
})
.or(function oops(err){
    // will not receive errors from Task A sequence
});
```

Forks > Spoons

You may need to split a single sequence into two separate paths, so `fork()` is provided:

```
var sq1 = ASQ(..).then(..)..;

var sq2 = sq1.fork();

sq1.then(..)..; // original sequence

sq2.then(..)..; // separate forked sequence
```

In this snippet, `sq2` won't proceed as its separate forked sequence until the pre-forked sequence steps complete (successfully).

Sugary Abstractions

OK, that's what you need to know about the foundational core of *asynquence*. While there's quite a bit of power there, it's still pretty limited compared to the feature lists of utilities like "Q" and "async". Fortunately, *asynquence* has a lot more up its sleeve.

In addition to the *asynquence* core, you can also use one or many of the provided *asynquence-contrib* plugins, which add lots of tasty abstraction helpers to the mix. The contrib builder lets you pick which ones you want, but builds all of them into the `contrib.js` package by default. In fact, you can even make your own plugins quite easily, but we'll discuss that in the next post in this series.

Gate Variations

There are 6 simple variations to the core `gate(..)` / `all(..)` functionality provided as contrib plugins: `any(..)`, `first(..)`, `race(..)`, `last(..)`, `none(..)`, and `map(..)`.

`any(..)` waits for all segments to complete just like `gate(..)`, but only one of them has to be a success for the main sequence to proceed. If none succeed, the main sequence is set to error state.

`first(..)` waits only for the first successful segment before the main sequence succeeds (subsequent segments are just ignored). If none succeed, the main sequence is set to error state.

`race(..)` is identical in concept to native `Promise.race(..)`, which is kind of like `first(..)`, except it's racing for the first completion regardless of success or failure.

`last(..)` waits for all segments to complete, but only the latest successful segment's success messages (if any) are sent along to the main sequence to proceed. If none succeed, the main sequence is set to error state.

`none(..)` waits for all segments to complete. It then transposes success and error states, which has the effect that the main sequence proceeds only if all segments failed, but is in error if any or all segments succeeded.

`map(..)` is an asynchronous "map" utility, much like you'll find in other libraries/utilities. It takes an array of values, and a function to call against each value, but it assumes the mapping may be asynchronous. The reason it's listed as a `gate(..)` variant is that it calls all mappings in parallel, and waits for all to complete before it proceeds. `map(..)` can have either the array or the iterator callback or both provided to it directly, or as messages from the previous main sequence step.

```
ASQ(function step1(done){
    setTimeout(function(){
        done( [1,2,3] );
    });
})
.map(function step2(item,done){
    setTimeout(function(){
        done( item * 2 );
    },100);
})
.val(function(arr){
    console.log(arr); // [2,4,6]
});
```

Step Variations

Other plugins provide variations on normal step semantics, such as `until(..)`, `try(..)`, and `waterfall(..)`.

`until(..)` keeps re-trying a step until it succeeds, or you call `done.break()` from inside it (which triggers error state on the main sequence).

`try(..)` attempts a step, and proceeds with success on the sequence regardless. If an error/failure is caught, it passes forward as a special success message in the form `{ catch: .. }`.

`waterfall(..)` takes multiple steps (like that would be provided to `then(..)` calls), and processes them in succession. However, it cascades the success message(s) from each step into the next, such that after the waterfall is complete, all success messages are passed along to the subsequent step. It saves you having to manually collect and pass them along, which can be quite tedious if you have many steps to waterfall.

Higher Order Abstractions

Any abstraction that you can dream up can be expressed as a combination of the above utilities and abstractions. If you have a common abstraction you find yourself doing regularly, you can make it repeatably usable by putting it into its own plugin (again, covered in the next post).

One example would be providing timeouts for a sequence, using `race(..)` (explained above) and the `failAfter(..)` plugin (which, as it sounds, makes a sequence that fails after a specified delay):

```
ASQ()
.race(
    // returns a sequence for some task
    doSomeTask(),
    // makes a sequence that will fail eventually
    ASQ.failAfter( 2000, "Timed Out!" )
)
.then(..)
.or(..);
```

This example sets up a race between a normal sequence and an eventually-failing sequence, to provide the semantics of a timeout limit.

If you found yourself doing that regularly, you could easily make a `timeoutLimit(..)` plugin for the above abstraction (see the next post).

Functional (Array) Operations

All the above examples have made one fundamental assumption, which is that you know ahead of time exactly what your flow-control steps are.

Sometimes, though, you need to respond to a varying amount of steps, such as each step representing a resource request, where you may need to request 3 or 30.

Using some very simple functional programming operations, like Array `map(..)` and `reduce(..)` , we can easily achieve this flexibility with promies, but you'll find that the API sugar of *asynquence* makes such tasks **even nicer**.

Note: If you don't know about map/reduce yet, you're going to want to spend some time (should only take a few hours tops) learning them, as you will find their usefulness all over promises-based coding!

Functional Example

Let's say you want to request 3 (or more) files in parallel, render their contents ASAP, but make sure they still render in natural order. If file1 comes back before file2, render file1 right away. If file2 comes back first, though, wait until file1 and then render both.

Here’s how you can do that with normal promises (we’ll ignore error handling for simplification purposes):

```
function getFile(file) {
    return new Promise(function(resolve){
        ajax(file,resolve);
    });
}

// Request all files at once in "parallel" via `getFile(..)`
[ "file1", "file2", "file3" ]
.map(getFile)
.reduce(
    function(chain,filePromise){
        return chain
            .then(function(){
                return filePromise;
            })
            .then(output);
    },
    Promise.resolve() // fulfilled promise to start chain
)
.then(function() {
    output("Complete!");
});
```

Not too bad, if you parse what’s happening with `map(..)` and then `reduce(..)`. The `map(..)` call turns an array of strings into an array of promises. The `reduce(..)` call ”reduces” the array of promises into a single chain of promises that will perform the steps in order as required.

Now, let’s look at how *asynquence* can do the same task:

```
function getFile(file) {
    return ASQ(function(done){
        ajax(file,done);
    });
}

ASQ()
.seq.apply(null,
    [ "file1", "file2", "file3" ]
    .map(getFile)
    .map(function(sq){
        return function(){
            return sq.val(output);
        };
    })
)
.val(function(){
    output("Complete!");
});
```

Note: These are sync map calls, so there's no real benefit to using *asynquence*'s async `map(..)` plugin discussed earlier.

Owing to some of the API sugar of *asynquence*, you can see we don't need `reduce(..)`, we just use two `map(..)` calls. The first turns the array of strings into an array of sequences. The second turns the array of sequences into an array of functions which each return a sub-sequence. This second array is sent as parameters to the `seq(..)` call in *asynquence*, which processes each sub-sequence in order.

Easy as cake, right?

`.summary(..)`

I think by now, if you've read this far, *asynquence* is speaking for itself. It's powerful, but it's also very terse and distinctly lacking in boilerplate cruft, compared to other libraries and especially compared to native promises.

It's also extensible (with plugins, as the next post will cover), so you have virtually no limits to what you can make it do for you.

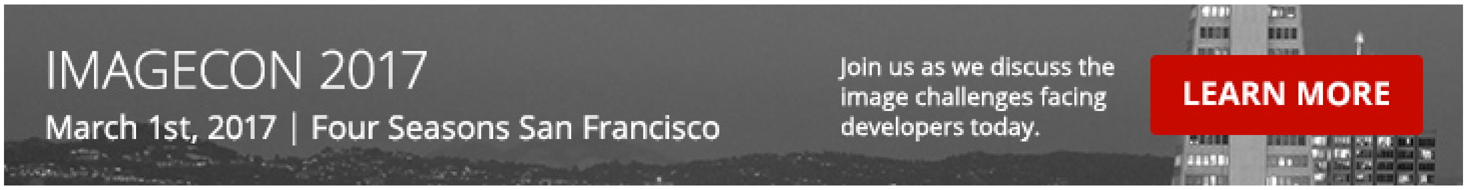
I hope you are convinced to at least give *asynquence* a try, now.

But if promise abstraction and API sugar was all *asynquence* had to offer, it might not obviously outshine its much more well-known peers. The next post will go way beyond promises into some much more advanced async capabilities. Let's find out just how deep the rabbit hole goes.

About Kyle Simpson

Kyle Simpson is an Open Web Evangelist from Austin, TX, who's passionate about all things JavaScript. He's an author, workshop trainer, tech speaker, and OSS contributor/leader.

getify.me getify posts


IMAGECON 2017
March 1st, 2017 | Four Seasons San Francisco
Join us as we discuss the image challenges facing developers today.
LEARN MORE

Recent Features

CSS Filters

CSS filter support recently landed within WebKit nightlies. CSS filters provide a method for modifying the rendering of a basic DOM element, image, or video. CSS filters allow for blurring, warping, and modifying the color intensity of elements. Let's have...

I'm an Impostor

This is the hardest thing I've ever had to write, much less admit to myself. I've written resignation letters from jobs I've loved, I've ended relationships, I've failed at a host of tasks, and let myself down in my life. All of those feelings were very...

Incredible Demos

CSS Vertical Center with Flexbox

I'm 31 years old and feel like I've been in the web development game for centuries. We knew forever that layouts in CSS were a nightmare and we all considered flexbox our savior. Whether it turns out that way remains to be seen but flexbox does easily...

jQuery UI DatePicker: Disable Specified Days

One project I'm currently working on requires jQuery. The project also features a datepicker for requesting a visit to their location. jQuery UI's DatePicker plugin was the natural choice and it does a really nice job. One challenge I encountered was the...

Discussion

Moshe Kolodny

This looks similar to something I wrote:
https://github.com/kolodny/wttt

Rick

Perhaps this post tries to go too quick over such advanced topics. I have read Kyle's book series titled 'You Don't Know JS'

Rick

… and it goes into much more detail about the language and the reasoning behind the need for generators,
iterators (etc) and how to avoid callback hell. I have yet to find anyone else who has tackled the technical
issues of the modern-day (2015) JS programming hurdles that arise inside complex (web) applications.

Please keep up the great work, Kyle. At this point, I see no one else on the JS scene who articulates JS core
constructs as well as methods on how to more robustly build advanced architecture than you.

Ivan Garavito

What a great work! I'm understanding everything.

I have to read the book YDKJS (https://github.com/getify/You-Dont-Know-JS).

Name	Email	Website

Wrap your code in `<pre class="{language}"></pre>` *tags, link to a GitHub gist, JSFiddle fiddle, or CodePen pen to embed!*