asynquence: More Than Just Promises (Part 2)

OSCON, Austin, TX • May 8-11 • Save 20% PC20DWALSH

By Kyle Simpson on June 17, 2014 2

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

- *Part 1: The Promises You Don't Know Yet*
- *Part 2: More Than Just Promises*

asynquence Is Promises

As we saw in part 1, *asynquence* is a wrapper abstraction on top of promises, as **sequences**. A single-step sequence approximates a promise, though they're not identically compatible. However, that's no big deal, since *asynquence* can both consume and vend standard promises/thenables easily.

{Track:js}

30 Day Free Trial

So, what's the big deal? "I don't need promise abstractions, because their limitations don't bother me." Or: "I already have a promise abstraction/extension lib that I like, that's really popular!"

In a sense, I agree with such a sentiment. If you don't see yet the *need* for *asynquence*, or if its flavor isn't appealing to you, I can understand not feeling compelled to switch to it.

But we have only just scratched the surface of *asynquence*. If you just stop here, you've missed the much bigger picture. Please, read on.

asynquence Is Much More... And Growing!

Firstly, we should talk about *asynquence* can be extended to do more than it ships with. I think this is one of the most interesting parts of the utility, especially given how small the package is, and how few of its peers (even much bigger ones) give this level of capability.

The entire list of *asynquence-contrib* plugins are provided as optional extensions to the core *asynquence* capability. That means they're a great place to start to inspect how you might make your own extensions.

A couple of them just add extra static helpers to the `ASQ` namespace, such as `ASQ.iterable(..)` (which we'll get to later). But most of them add chainable methods to the instance API, so that you can do things like call the `first(..)` plugin in mid-sequence chain, like `ASQ().then(..).first(..).then(..)..`. That's pretty powerful.

Let's imagine a simple scenario: You find yourself regularly wanting to log (to the dev console, for instance) the value of some message as it passes through a certain step of your sequence. Here's how you normally do it:

```
ASQ(..)
.then(..)
.val(function(msg){
    console.log(msg);
    return msg;
})
.then(..)
..
```

Would it be nice to have a re-usable way of doing that? You could declare one, like:

```
function ASQlog(msg) {
    console.log(msg);
    return msg;
}

ASQ(..)
.then(..)
```

```
.val( ASQlog )
.then(..)
..
```

But we can make it even better, with our own custom contrib plugin. First, here's how we use it:

```
ASQ(..)
.then(..)
.log()
.then(..)
..
```

Ooo, that's nicer! How do we do it? Make a file called "plugin.log.js" in the contrib package root, then put something like this in it:

```
ASQ.extend( "log", function __log__(api,internals){
    return function __log__() {
        api.val(function(msg){
            console.log(msg);
            return msg;
        });

        return api;
    };
});
```

That's easy, right!? Basically, whatever normal usage you find of the public ASQ API that you repeat frequently, you can wrap up that same sort of call

Now, let's make it a little more robust (to handle more than one success message passing through) and *also* make it log out any errors:

```
ASQ.extend( "log", function __log__(api,internals){
    return function __log__() {
        api.val(function(){
```

```
            console.log.apply(console,arguments);
            return ASQ.messages.apply(null,arguments);
        })
        .or(function(){
            console.error.apply(console,arguments);
        });

        return api;
    };
});
```

Here you see the use of the `ASQ.messages(..)` utility. That's a simple way of creating an array of values that is specifically branded by `ASQ` so that the array can be recognized and unwrapped (into positional parameters) where appropriate.

Let's make another silly example:

```
ASQ("foo and bar are awesome!")
.fOObAR()
.log(); // "fOO and bAR are awesome!"
```

How?

```
ASQ.extend( "fOObAR", function __fOObAR__(api,internals){
    return function __fOObAR__() {
        api.val(function(msg){
            return msg
                .replace(/\bfoo\b/g,"fOO")
                .replace(/\bbar\b/g,"bAR");
        });

        return api;
    };
});
```

Iterable Sequences

If you look at how sequences work, they internally advanced themselves by calling the each step's respective trigger (just like promises do). But there are certainly cases where being able to advance a sequence from the outside would be nice.

For example, let's imagine a one-time-only event like `DOMContentLoaded`, where you need to advanced a main sequence only when that event occurs.

Here's how you have to "hack" it if all you have is *asynquence* core:

```
ASQ(function(done){
    document.addEventListener("DOMContentLoaded",done,false);
})
.then(..)
..
```

Or, you do "capability extraction" (unfortunately more common in Promises than I think it should be), to get better separation of concerns/capabilities:

```
var trigger;

ASQ(function(done){
    trigger = done; // extract the trigger
})
.then(..)
..

// later, elsewhere
document.addEventListener("DOMContentLoaded",trigger,false);
```

All of those options and their variations suck, especially when you consider a multi-step initialization before the main sequence fires, like both the `DOMContentLoaded` firing and an initial setup Ajax request coming back.

So, we now introduce a somewhat different concept, provided by the `iterable(..)` plugin: **iterable-sequences**. These are sequences which are not internally advanceable, but are instead

advanced externally, with the familiar *Iterator* interface: `.next(..)`.

Each step of the iterable-sequence doesn't get its own trigger, and there are also no automatically passed success messages from step to step. Instead, you pass a message in with `next(..)`, and you get a value back out at the end of the step (an operation that is itself fundamentally synchronous). The "async" nature of these sequences is external to the sequence, hidden away in whatever logic controls the sequence's iteration.

`DOMContentLoaded` example:

```
var trigger = ASQ.iterable();

document.addEventListener("DOMContentLoaded",trigger.next,false);

// setup main async flow-control
ASQ( trigger ) // wait for trigger to fire before proceeding
.then(..)
.then(..)
..
```

Or for multi-step:

```
var noop = function(){};
var setup = ASQ.iterable().then(noop);

document.addEventListener("DOMContentLoaded",setup.next,false);
ajax("some-url",function(response){
    // do stuff with response
    setup.next();
});

// setup main async flow-control
ASQ( setup ) // wait for setup to complete before proceeding
.then(..)
.then(..)
..
```

Iterating Iterable-Sequences

Iterable-sequences can also be set up to have a pre-defined (or even infinite) set of steps, and then it can be iterated on using normal iteration techniques.

For example, to manually sync iterate an iterable-sequence with a `for` loop:

```
function double(x) { return x * 2; }
function triple(x) { return x * 3; }

var isq = ASQ.iterable()
.then(double)
.then(double)
.then(triple);

for (var seed = 3, ret;
    (ret = isq.next(seed)) && !ret.done;
) {
    seed = ret.value;
    console.log(seed);
}
// 6
// 12
// 36
```

Even better, ES6 gives us `@@Iterator` hooks, plus the `for..of` loop, to automatically iterate over iterable-sequences (assuming each step doesn't need input):

```
var x = 0;
function inc() { return ++x; }

var isq = ASQ.iterable()
.then(inc)
.then(inc)
.then(inc);

for (var v of isq) {
    console.log(v);
}
// 1
```

```
// 2
// 3
```

Of course, these are examples of iterating an iterable-sequence synchronously, but it's trivial to imagine how you call `next(..)` inside of async tasks like timers, event handlers, etc, which has the effect of asynchronously stepping through the iterable-sequence's steps.

In this way, iterable-sequences are kind of like generators (which we'll cover next), where each step is like a `yield`, and `next(..)` restarts the sequence/generator.

Generators

In addition to `Promise`, ES6 adds generators capability, which is another huge addition to JS's ability to handle async programming more sanely.

I won't teach all of generators here (there's plenty of stuff already written about them). But let me just quickly code the previous example with a generator instead, for illustration purposes:

```
function* gen() {
    var x = 0;
    yield ++x;
    yield ++x;
    yield ++x;
}
for ( var v of gen() ) {
    console.log(v);
}
// 1
// 2
// 3
```

As you can see, generators essentially look like synchronous code, but the `yield` keyword pauses it mid-execution, optionally returning a value. The `for..of` loop hides the `next()` calls, and thus sends nothing in, but you could manually iterate a generator if you needed to pass values in at each iteration, just like I did above with iterable-sequences.

But this isn't the cool part of generators. The cool part is when generators are combined with promises. For example:

```
function asyncIncrement(x) {
    return new Promise(function(resolve){
        setTimeout(function(){
            resolve(++x);
        },500);
    });
}

runAsyncGenerator(function*(){
    var x = 0;
    while (x < 3) {
        x = yield asyncIncrement(x);
    }
    console.log(x);
});
// 3
```

Some very important things to notice:

1. I have used some mythical `runAsyncGenerator(..)` utility. We'll come back to that in a minute.
2. What we `yield` out of our generator is actually a promise for a value, rather than an immediate value. We obviously get something back after our promise completes, and that something is the incremented number.

Inside the `runAsyncGenerator(..)` utility, I would have an iterator controlling my generator, which would be calling `next(..)` on it successively.

What it gets back from a `next(..)` call is a **promise**, so we just listen for that promise to finish, and when it does, we take its success value and pass it back into the next `next(..)` call.

In other words, `runAsyncGenerator(..)` automatically and asynchronously runs our generator to its completion, with each async promise "step" just pausing the iteration until resolution.

This is a hugely powerful technique, as it allows us to write sync-looking code, like our `while` loop, but hide away **as an implementation detail** the fact that the promises we `yield` out introduce asynchronicity into the iteration loop.

asynquence?

Several other async/promises libraries have a utility like `runAsyncGenerator(..)` already built-in (called `spawn(..)` or `co(..)`, etc). And so does *asynquence*, called `runner(..)`. But the one *asynquence* provides is much more powerful!

The most important thing is that *asynquence* lets you wire a generator up to run right in the middle of a normal sequence, like a specialized `then(..)` sort of step, which also lets you pass previous sequence step messages *into* the generator, and it lets you `yield` value(s) out from the end of the generator to continue in the main sequence.

To my knowledge, no other library has that capability! Let's see what it looks like:

```
function inc(x,y) {
    return ASQ(function(done){
        setTimeout(function(){
            done(x + y);
        },500);
    });
}

ASQ( 3, 4 )
.runner(function*(control){
    var x = control.messages[0];
    var y = control.messages[1];

    while (x < 20) {
        x = yield inc(x,y);
    }

    // Note: `23` was the last value yielded out,
    // so it's automatically the success value from
    // the generator. If you wanted to send some
    // other value out, just call another `yield __`
    // here.
})
.val(function(msg){
```

```
    console.log(msg); // 23
});
```

The `inc(..)` shown returns an *asynquence* instance, but it would have worked identically if it had returned a normal promise, as `runner(..)` listens for either promises or sequences and treats them appropriately. Of course, you could have yielded out a much more complex, multi-step sequence (or promise-chain) if you wanted, and `runner(..)` would just sit around patiently waiting.

That's pretty powerful, don't you think!? **Generators + Promises unquestionablly represents the future direction of async programming in JS.** In fact, early proposals for ES7 suggest we'll get `async` functions which will have native syntactic support for what `spawn(..)` and `runner(..)` do. Super exciting!

But that's just barely scratching the surface of how *asynquence* leverages the power of generators.

CSP-style Concurrency (like go)

We just saw the power of a single generator being run-to-completion in the middle of a sequence.

But what if you paired two or more generators together, so that they yield back-and-forth to each other? In essence, you'd be accomplishing CSP-style (**C**ommunicating **S**equential **P**rocesses) concurrency, where each generator was like a sequential "process", and they cooperatively interleaved their own individual steps. They also have a shared message channel to send messages between them.

I cannot overstate the power of this pattern.

It's basically what the *go* language supports naturally, and what ClojureScript's `core.async` functionality automatically creates in JS. I highly recommend you read David Nolen's fantastic writings on the topic, like this post and this post, as well as others. Also, check out his Om framework which makes use of these ideas and more.

In fact, there's also a stand-alone library for exactly this CSP-style concurrency task, called js-csp.

asynquence CSP-style

But this post is about *asynquence*, right? Rather than needing a separate library or different language, the power of *asynquence* is that you can do CSP-style programming with the same utility that you do all your other promises work.

Rather than fully teaching the whole concept, I'll choose to just illustrate it with code and let you examine and learn to whatever extent this piques your interest. I personally feel this is a big part of the future of advanced async programming in the language.

I'm going to rip/fork/port this example directly from *go* and *js-csp*... the classic "Ping Pong" demo example. To see it work, run the demo in a browser (**Note:** currently, only Chrome's generators are spec-compliant enough to run the example -- FF is close but not quite there).

A snippet of the demo's code:

```
ASQ(
    ["ping","pong"], // player names
    { hits: 0 } // the ball
)
.runner(
    referee,
    player,
    player
)
.val(function(msg){
    console.log("referee",msg); // "Time's up!"
});
```

Briefly, if you examine the full JS code at that demo link, you can see 3 generators (`referee` and two instances of `player`) that are run by `runner(..)` , trading control with each other (by `yield table` statements), and messaging each other through the shared message channels in `table.messages` .

You can still yield promises/sequences from a generator, as `yield sleep(500)` does, which doesn't transfer control but just pauses that generator's progression until the promise/sequence completes.

Again... wow. Generators paired together as CSP-style coroutines is a huge and largely untapped horizon we're just starting to advanced towards. *asynquence* is on the leading edge of that evolution, letting you explore the power of these techniques right alongside the more familiar promises capabilities. No framework switching -- it's all in one utility.

Event-Reactive

OK, the last advanced pattern I am going to explore here with *asynquence* is the "reactive observables" pattern from the RxJS -- Reactive Extensions library from the smart folks (like Matt Podwysocki) at Microsoft. I was inspired by their "reactive observables" and added a similar concept, which I call "reactive sequences", via the `react(..)` plugin.

Briefly, the problem we want to address is that promises only work well for single-fire types of events. What if you had a repeating event (like a button click) that you wanted to fire off a sequence of events for each trigger?

We could do it like this:

```
$("#button").click(function(evt){
    ASQ(..)
    .then(..)
    .then(..)
    ..
});
```

But that kinda sucks for separation of concerns/capabilities. We'd like to be able to separate the specification of the flow-control sequence from the listening for the event that will fire it off. In other words, we'd like to invert that example's "nesting".

The *asynquence* `react(..)` plugin gives you that capability:

```
var sq = ASQ.react(function(trigger){
    $("#button").click(trigger);
});

// elsewhere:
```

```
sq
.then(..)
.then(..)
..
```

Each time the `trigger` function is called, **a new copy** of the defined sequence (aka template) is spun off and runs independently.

Though not shown here, you can also register steps to take when tearing down the reactive-sequence (to unbind handlers, etc). There's also a special helper for listening for events on node.js streams.

Here's some more concrete examples:

1. DEMO: Reactive Sequences + `gate(..)`
2. CODE: Reactive Sequences + node.js HTTP streams

So, bottom line, you could easily switch to using the whole RxJS library (it's quite large/complex but extremely capable!) for such event-reactive async programming, **or you can use *asynquence** and get some of that important functionality just built-in to the utility that already handles your other async flow-control tasks.

Wrapping Up

I think you can probably agree by now: that's a whole bunch of advanced functionality and patterns you get out-of-the-box with *asynquence*.

I encourage you to give *asynquence* a shot and see if it doesn't simplify and revolutionize your async coding in JS.

And if you find something that's substantially missing in terms of functionality, I bet we can write a plugin that'll do it pretty easily!

Here's the most important take-away I can leave you with: I didn't write *asynquence* or this blog post series **just** so that you'd use the lib (although I hope you give it a shot). I built it in the open,

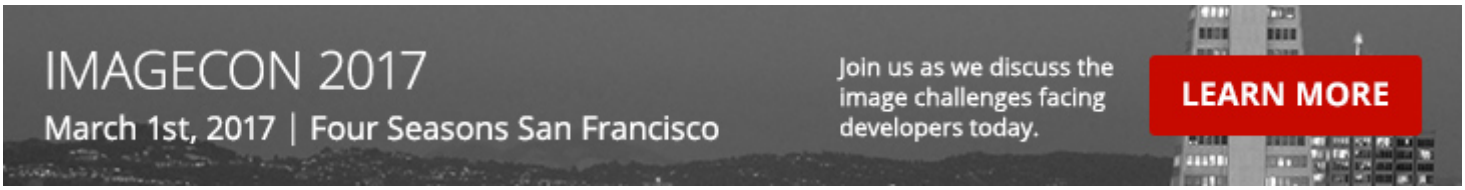
and wrote these public posts, to inspire you to help me make it better and better.

I want *asynquence* to be the most powerful collection of async flow-control utilities anywhere. **You can help me** make that happen.

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



Recent Features

How to Create a Twitter Card

One of my favorite social APIs was the Open Graph API adopted by Facebook. Adding just a few META tags to each page allowed links to my article to be styled and presented the way I wanted them to, giving me a bit of control...

CSS Animations Between Media Queries

CSS animations are right up there with sliced bread. CSS animations are efficient because they can be hardware accelerated, they require no JavaScript overhead, and they are composed of very little CSS code. Quite often we add CSS transforms to elements via CSS during...

Incredible Demos

Using jQuery and MooTools Together

There's yet another reason to master more than one JavaScript library: you can use some of them together! Since MooTools is prototype-based and jQuery is not, jQuery and MooTools may be used together on the same page. The XHTML and JavaScript jQuery is namespaced so the...

MooTools PulseFade Plugin

I was recently driven to create a MooTools plugin that would take an element and fade it to a min from a max for a given number of times. Here's the result of my Moo-foolery. The MooTools JavaScript Options of the class include: min: (defaults to .5) the...

Discussion

Rick

Since this writing (June of 2014), Kyle went on to write 'You Don't Know JS', a series of small books (published on o'reilly) that address the above concepts in much greater detail (Async & Performance). He also clarifies some grave misunderstandings about the mis-use of the JavaScript language by most JS programmers who come to the language with a class based understanding/training of programming (Scope & Closures, This & Object Prototypes).

If anyone can link to more in-depth teachings of JS as well as a more competent library to handle the most advanced techniques of JS programming then I would be appreciative.

Ivan Garavito

What a great post! I'll be playing around asynquence

Name | Email | Website

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

Continue this conversation via email

Post Comment! Use Code Editor