You Don't Know JS: Async & Performance

Appendix B: Advanced Async Patterns

Appendix A introduced the *asynquence* library for sequence-oriented async flow control, primarily based on Promises and generators.

Now we'll explore other advanced asynchronous patterns built on top of that existing understanding and functionality, and see how *asynquence* makes those sophisticated async techniques easy to mix and match in our programs without needing lots of separate libraries.

Iterable Sequences

We introduced *asynquence*'s iterable sequences in the previous appendix, but we want to revisit them in more detail.

To refresh, recall:

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

// ..

domready.val( function(){
        // DOM is ready
} );

// ..

document.addEventListener( "DOMContentLoaded", domready.next );
```

Now, let's define a sequence of multiple steps as an iterable sequence:

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

steps
.then( function STEP1(x){
        return x * 2;
} )
.then( function STEP2(x){
        return x + 3;
} )
.then( function STEP3(x){
        return x * 4;
} );

steps.next( 8 ).value;  // 16
steps.next( 16 ).value; // 19
steps.next( 19 ).value; // 76
steps.next().done;              // true
```

As you can see, an iterable sequence is a standard-compliant *iterator* (see Chapter 4). So, it can be iterated with an ES6 `for..of` loop, just like a generator (or any other *iterable*) can:

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

steps
.then( function STEP1(){ return 2; } )
.then( function STEP2(){ return 4; } )
```

```
.then( function STEP3(){ return 6; } )
.then( function STEP4(){ return 8; } )
.then( function STEP5(){ return 10; } );

for (var v of steps) {
        console.log( v );
}
// 2 4 6 8 10
```

Beyond the event triggering example shown in the previous appendix, iterable sequences are interesting because in essence they can be seen as a stand-in for generators or Promise chains, but with even more flexibility.

Consider a multiple Ajax request example -- we've seen the same scenario in Chapters 3 and 4, both as a Promise chain and as a generator, respectively -- expressed as an iterable sequence:

```
// sequence-aware ajax
var request = ASQ.wrap( ajax );

ASQ( "http://some.url.1" )
.runner(
        ASQ.iterable()

        .then( function STEP1(token){
                var url = token.messages[0];
                return request( url );
        } )

        .then( function STEP2(resp){
                return ASQ().gate(
                        request( "http://some.url.2/?v=" + resp ),
                        request( "http://some.url.3/?v=" + resp )
                );
        } )

        .then( function STEP3(r1,r2){ return r1 + r2; } )
)
.val( function(msg){
        console.log( msg );
} );
```

The iterable sequence expresses a sequential series of (sync or async) steps that looks awfully similar to a Promise chain -- in other words, it's much cleaner looking than just plain nested callbacks, but not quite as nice as the `yield` -based sequential syntax of generators.

But we pass the iterable sequence into `ASQ#runner(..)`, which runs it to completion the same as if it was a generator. The fact that an iterable sequence behaves essentially the same as a generator is notable for a couple of reasons.

First, iterable sequences are kind of a pre-ES6 equivalent to a certain subset of ES6 generators, which means you can either author them directly (to run anywhere), or you can author ES6 generators and transpile/convert them to iterable sequences (or Promise chains for that matter!).

Thinking of an async-run-to-completion generator as just syntactic sugar for a Promise chain is an important recognition of their isomorphic relationship.

Before we move on, we should note that the previous snippet could have been expressed in *asynquence* as:

```
ASQ( "http://some.url.1" )
.seq( /*STEP 1*/ request )
.seq( function STEP2(resp){
        return ASQ().gate(
                request( "http://some.url.2/?v=" + resp ),
                request( "http://some.url.3/?v=" + resp )
        );
} )
.val( function STEP3(r1,r2){ return r1 + r2; } )
```

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

Moreover, step 2 could have even been expressed as:

```
.gate(
        function STEP2a(done,resp) {
                request( "http://some.url.2/?v=" + resp )
                .pipe( done );
        },
        function STEP2b(done,resp) {
                request( "http://some.url.3/?v=" + resp )
                .pipe( done );
        }
)
```

So, why would we go to the trouble of expressing our flow control as an iterable sequence in a `ASQ#runner(..)` step, when it seems like a simpler/flatter *asyquence* chain does the job well?

Because the iterable sequence form has an important trick up its sleeve that gives us more capability. Read on.

Extending Iterable Sequences

Generators, normal *asynquence* sequences, and Promise chains, are all **eagerly evaluated** -- whatever flow control is expressed initially *is* the fixed flow that will be followed.

However, iterable sequences are **lazily evaluated**, which means that during execution of the iterable sequence, you can extend the sequence with more steps if desired.

Note: You can only append to the end of an iterable sequence, not inject into the middle of the sequence.

Let's first look at a simpler (synchronous) example of that capability to get familiar with it:

```
function double(x) {
        x *= 2;

        // should we keep extending?
        if (x < 500) {
                isq.then( double );
        }

        return x;
}

// setup single-step iterable sequence
var isq = ASQ.iterable().then( double );

for (var v = 10, ret;
        (ret = isq.next( v )) && !ret.done;
) {
        v = ret.value;
        console.log( v );
}
```

The iterable sequence starts out with only one defined step (`isq.then(double)`), but the sequence keeps extending itself under certain conditions (`x < 500`). Both *asynquence* sequences and Promise chains technically *can* do something similar, but we'll see in a little bit why their capability is insufficient.

Though this example is rather trivial and could otherwise be expressed with a `while` loop in a generator, we'll consider more sophisticated cases.

For instance, you could examine the response from an Ajax request and if it indicates that more data is needed, you conditionally insert more steps into the iterable sequence to make the additional request(s). Or you could conditionally add a value-formatting step to the end of your Ajax handling.

Consider:

```
var steps = ASQ.iterable()

.then( function STEP1(token){
        var url = token.messages[0].url;

        // was an additional formatting step provided?
        if (token.messages[0].format) {
                steps.then( token.messages[0].format );
        }

        return request( url );
} )

.then( function STEP2(resp){
        // add another Ajax request to the sequence?
        if (/x1/.test( resp )) {
                steps.then( function STEP5(text){
                        return request(
                                "http://some.url.4/?v=" + text
                        );
                } );
        }

        return ASQ().gate(
                request( "http://some.url.2/?v=" + resp ),
                request( "http://some.url.3/?v=" + resp )
        );
} )

.then( function STEP3(r1,r2){ return r1 + r2; } );
```

You can see in two different places where we conditionally extend `steps` with `steps.then(..)`. And to run this `steps` iterable sequence, we just wire it into our main program flow with an *asynquence* sequence (called `main` here) using `ASQ#runner(..)`:

```
var main = ASQ( {
        url: "http://some.url.1",
        format: function STEP4(text){
                return text.toUpperCase();
        }
} )
.runner( steps )
.val( function(msg){
        console.log( msg );
} );
```

Can the flexibility (conditional behavior) of the `steps` iterable sequence be expressed with a generator? Kind of, but we have to rearrange the logic in a slightly awkward way:

```
function *steps(token) {
        // **STEP 1**
        var resp = yield request( token.messages[0].url );

        // **STEP 2**
        var rvals = yield ASQ().gate(
                request( "http://some.url.2/?v=" + resp ),
                request( "http://some.url.3/?v=" + resp )
        );
```

```
        // **STEP 3**
        var text = rvals[0] + rvals[1];

        // **STEP 4**
        // was an additional formatting step provided?
        if (token.messages[0].format) {
                text = yield token.messages[0].format( text );
        }

        // **STEP 5**
        // need another Ajax request added to the sequence?
        if (/foobar/.test( resp )) {
                text = yield request(
                        "http://some.url.4/?v=" + text
                );
        }

        return text;
}

// note: `*steps()` can be run by the same `ASQ` sequence
// as `steps` was previously
```

Setting aside the already identified benefits of the sequential, synchronous-looking syntax of generators (see Chapter 4), the `steps` logic had to be reordered in the `*steps()` generator form, to fake the dynamicism of the extendable iterable sequence `steps`.

What about expressing the functionality with Promises or sequences, though? You *can* do something like this:

```
var steps = something( .. )
.then( .. )
.then( function(..){
        // ..

        // extending the chain, right?
        steps = steps.then( .. );

        // ..
})
.then( .. );
```

The problem is subtle but important to grasp. So, consider trying to wire up our `steps` Promise chain into our main program flow -- this time expressed with Promises instead of *asynquence*:

```
var main = Promise.resolve( {
        url: "http://some.url.1",
        format: function STEP4(text){
                return text.toUpperCase();
        }
} )
.then( function(..){
        return steps;                   // hint!
} )
.val( function(msg){
        console.log( msg );
} );
```

Can you spot the problem now? Look closely!

There's a race condition for sequence steps ordering. When you `return steps`, at that moment `steps` *might* be the originally defined promise chain, or it might now point to the extended promise chain via the `steps = steps.then(..)` call, depending on what order things happen.

Here are the two possible outcomes:

- If `steps` is still the original promise chain, once it's later "extended" by `steps = steps.then(..)`, that extended promise on the end of the chain is **not** considered by the `main` flow, as it's already tapped the `steps` chain. This is the unfortunately limiting **eager evaluation**.
- If `steps` is already the extended promise chain, it works as we expect in that the extended promise is what `main` taps.

Other than the obvious fact that a race condition is intolerable, the first case is the concern; it illustrates **eager evaluation** of the promise chain. By contrast, we easily extended the iterable sequence without such issues, because iterable sequences are **lazily evaluated**.

The more dynamic you need your flow control, the more iterable sequences will shine.

Tip: Check out more information and examples of iterable sequences on the *asynquence* site (https://github.com/getify/asynquence/blob/master/README.md#iterable-sequences).

Event Reactive

It should be obvious from (at least!) Chapter 3 that Promises are a very powerful tool in your async toolbox. But one thing that's clearly lacking is in their capability to handle streams of events, as a Promise can only be resolved once. And frankly, this exact same weakness is true of plain *asynquence* sequences, as well.

Consider a scenario where you want to fire off a series of steps every time a certain event is fired. A single Promise or sequence cannot represent all occurrences of that event. So, you have to create a whole new Promise chain (or sequence) for *each* event occurrence, such as:

```
listener.on( "foobar", function(data){

        // create a new event handling promise chain
        new Promise( function(resolve,reject){
                // ..
        } )
        .then( .. )
        .then( .. );

} );
```

The base functionality we need is present in this approach, but it's far from a desirable way to express our intended logic. There are two separate capabilities conflated in this paradigm: the event listening, and responding to the event; separation of concerns would implore us to separate out these capabilities.

The carefully observant reader will see this problem as somewhat symmetrical to the problems we detailed with callbacks in Chapter 2; it's kind of an inversion of control problem.

Imagine uninverting this paradigm, like so:

```
var observable = listener.on( "foobar" );

// later
observable
.then( .. )
.then( .. );

// elsewhere
observable
.then( .. )
.then( .. );
```

The `observable` value is not exactly a Promise, but you can *observe* it much like you can observe a Promise, so it's closely related. In fact, it can be observed many times, and it will send out notifications every time its event (`"foobar"`) occurs.

Tip: This pattern I've just illustrated is a **massive simplification** of the concepts and motivations behind reactive programming (aka RP), which has been implemented/expounded upon by several great projects and languages. A variation on RP is

functional reactive programming (FRP), which refers to applying functional programming techniques (immutability, referential integrity, etc.) to streams of data. "Reactive" refers to spreading this functionality out over time in response to events. The interested reader should consider studying "Reactive Observables" in the fantastic "Reactive Extensions" library ("RxJS" for JavaScript) by Microsoft (http://rxjs.codeplex.com/); it's much more sophisticated and powerful than I've just shown. Also, Andre Staltz has an excellent write-up (https://gist.github.com/staltz/868e7e9bc2a7b8c1f754) that pragmatically lays out RP in concrete examples.

ES7 Observables

At the time of this writing, there's an early ES7 proposal for a new data type called "Observable" (https://github.com/jhusain/asyncgenerator#introducing-observable), which in spirit is similar to what we've laid out here, but is definitely more sophisticated.

The notion of this kind of Observable is that the way you "subscribe" to the events from a stream is to pass in a generator -- actually the *iterator* is the interested party -- whose `next(..)` method will be called for each event.

You could imagine it sort of like this:

```
// `someEventStream` is a stream of events, like from
// mouse clicks, and the like.

var observer = new Observer( someEventStream, function*(){
        while (var evt = yield) {
                console.log( evt );
        }
} );
```

The generator you pass in will `yield` pause the `while` loop waiting for the next event. The *iterator* attached to the generator instance will have its `next(..)` called each time `someEventStream` has a new event published, and so that event data will resume your generator/*iterator* with the `evt` data.

In the subscription to events functionality here, it's the *iterator* part that matters, not the generator. So conceptually you could pass in practically any iterable, including `ASQ.iterable()` iterable sequences.

Interestingly, there are also proposed adapters to make it easy to construct Observables from certain types of streams, such as `fromEvent(..)` for DOM events. If you look at a suggested implementation of `fromEvent(..)` in the earlier linked ES7 proposal, it looks an awful lot like the `ASQ.react(..)` we'll see in the next section.

Of course, these are all early proposals, so what shakes out may very well look/behave differently than shown here. But it's exciting to see the early alignments of concepts across different libraries and language proposals!

Reactive Sequences

With that crazy brief summary of Observables (and F/RP) as our inspiration and motivation, I will now illustrate an adaptation of a small subset of "Reactive Observables," which I call "Reactive Sequences."

First, let's start with how to create an Observable, using an *asynquence* plug-in utility called `react(..)`:

```
var observable = ASQ.react( function setup(next){
        listener.on( "foobar", next );
} );
```

Now, let's see how to define a sequence that "reacts" -- in F/RP, this is typically called "subscribing" -- to that `observable`:

```
observable
.seq( .. )
.then( .. )
.val( .. );
```

So, you just define the sequence by chaining off the Observable. That's easy, huh?

In F/RP, the stream of events typically channels through a set of functional transforms, like `scan(..)`, `map(..)`, `reduce(..)`, and so on. With reactive sequences, each event channels through a new instance of the sequence. Let's look at a more concrete example:

```
ASQ.react( function setup(next){
        document.getElementById( "mybtn" )
        .addEventListener( "click", next, false );
} )
.seq( function(evt){
        var btnID = evt.target.id;
        return request(
                "http://some.url.1/?id=" + btnID
        );
} )
.val( function(text){
        console.log( text );
} );
```

The "reactive" portion of the reactive sequence comes from assigning one or more event handlers to invoke the event trigger (calling `next(..)`).

The "sequence" portion of the reactive sequence is exactly like the sequences we've already explored: each step can be whatever asynchronous technique makes sense, from continuation callback to Promise to generator.

Once you set up a reactive sequence, it will continue to initiate instances of the sequence as long as the events keep firing. If you want to stop a reactive sequence, you can call `stop()`.

If a reactive sequence is `stop()`'d, you likely want the event handler(s) to be unregistered as well; you can register a teardown handler for this purpose:

```
var sq = ASQ.react( function setup(next,registerTeardown){
        var btn = document.getElementById( "mybtn" );

        btn.addEventListener( "click", next, false );

        // will be called once `sq.stop()` is called
        registerTeardown( function(){
                btn.removeEventListener( "click", next, false );
        } );
} )
.seq( .. )
.then( .. )
.val( .. );

// later
sq.stop();
```

Note: The `this` binding reference inside the `setup(..)` handler is the same `sq` reactive sequence, so you can use the `this` reference to add to the reactive sequence definition, call methods like `stop()`, and so on.

Here's an example from the Node.js world, using reactive sequences to handle incoming HTTP requests:

```
var server = http.createServer();
server.listen(8000);

// reactive observer
var request = ASQ.react( function setup(next,registerTeardown){
        server.addListener( "request", next );
        server.addListener( "close", this.stop );

        registerTeardown( function(){
                server.removeListener( "request", next );
```

```
                server.removeListener( "close", request.stop );
        } );
});

// respond to requests
request
.seq( pullFromDatabase )
.val( function(data,res){
        res.end( data );
} );

// node teardown
process.on( "SIGINT", request.stop );
```

The `next(..)` trigger can also adapt to node streams easily, using `onStream(..)` and `unStream(..)`:

```
ASQ.react( function setup(next){
        var fstream = fs.createReadStream( "/some/file" );

        // pipe the stream's "data" event to `next(..)`
        next.onStream( fstream );

        // listen for the end of the stream
        fstream.on( "end", function(){
                next.unStream( fstream );
        } );
} )
.seq( .. )
.then( .. )
.val( .. );
```

You can also use sequence combinations to compose multiple reactive sequence streams:

```
var sq1 = ASQ.react( .. ).seq( .. ).then( .. );
var sq2 = ASQ.react( .. ).seq( .. ).then( .. );

var sq3 = ASQ.react(..)
.gate(
        sq1,
        sq2
)
.then( .. );
```

The main takeaway is that `ASQ.react(..)` is a lightweight adaptation of F/RP concepts, enabling the wiring of an event stream to a sequence, hence the term "reactive sequence." Reactive sequences are generally capable enough for basic reactive uses.

Note: Here's an example of using `ASQ.react(..)` in managing UI state (http://jsbin.com/rozipaki/6/edit?js,output), and another example of handling HTTP request/response streams with `ASQ.react(..)` (https://gist.github.com/getify/bba5ec0de9d6047b720e).

Generator Coroutine

Hopefully Chapter 4 helped you get pretty familiar with ES6 generators. In particular, we want to revisit the "Generator Concurrency" discussion, and push it even further.

We imagined a `runAll(..)` utility that could take two or more generators and run them concurrently, letting them cooperatively `yield` control from one to the next, with optional message passing.

In addition to being able to run a single generator to completion, the `ASQ#runner(..)` we discussed in Appendix A is a similar implementation of the concepts of `runAll(..)`, which can run multiple generators concurrently to completion.

So let's see how we can implement the concurrent Ajax scenario from Chapter 4:

```
ASQ(
        "http://some.url.2"
)
.runner(
        function*(token){
                // transfer control
                yield token;

                var url1 = token.messages[0]; // "http://some.url.1"

                // clear out messages to start fresh
                token.messages = [];

                var p1 = request( url1 );

                // transfer control
                yield token;

                token.messages.push( yield p1 );
        },
        function*(token){
                var url2 = token.messages[0]; // "http://some.url.2"

                // message pass and transfer control
                token.messages[0] = "http://some.url.1";
                yield token;

                var p2 = request( url2 );

                // transfer control
                yield token;

                token.messages.push( yield p2 );

                // pass along results to next sequence step
                return token.messages;
        }
)
.val( function(res){
        // `res[0]` comes from "http://some.url.1"
        // `res[1]` comes from "http://some.url.2"
} );
```

The main differences between `ASQ#runner(..)` and `runAll(..)` are as follows:

- Each generator (coroutine) is provided an argument we call `token`, which is the special value to `yield` when you want to explicitly transfer control to the next coroutine.
- `token.messages` is an array that holds any messages passed in from the previous sequence step. It's also a data structure that you can use to share messages between coroutines.
- `yield`ing a Promise (or sequence) value does not transfer control, but instead pauses the coroutine processing until that value is ready.
- The last `return`ed or `yield`ed value from the coroutine processing run will be forward passed to the next step in the sequence.

It's also easy to layer helpers on top of the base `ASQ#runner(..)` functionality to suit different uses.

State Machines

One example that may be familiar to many programmers is state machines. You can, with the help of a simple cosmetic utility, create an easy-to-express state machine processor.

Let's imagine such a utility. We'll call it `state(..)`, and will pass it two arguments: a state value and a generator that handles that state. `state(..)` will do the dirty work of creating and returning an adapter generator to pass to `ASQ#runner(..)`.

Consider:

```
function state(val,handler) {
	// make a coroutine handler for this state
	return function*(token) {
		// state transition handler
		function transition(to) {
			token.messages[0] = to;
		}

		// set initial state (if none set yet)
		if (token.messages.length < 1) {
			token.messages[0] = val;
		}

		// keep going until final state (false) is reached
		while (token.messages[0] !== false) {
			// current state matches this handler?
			if (token.messages[0] === val) {
				// delegate to state handler
				yield *handler( transition );
			}

			// transfer control to another state handler?
			if (token.messages[0] !== false) {
				yield token;
			}
		}
	};
}
```

If you look closely, you'll see that `state(..)` returns back a generator that accepts a `token`, and then it sets up a `while` loop that will run until the state machine reaches its final state (which we arbitrarily pick as the `false` value); that's exactly the kind of generator we want to pass to `ASQ#runner(..)`!

We also arbitrarily reserve the `token.messages[0]` slot as the place where the current state of our state machine will be tracked, which means we can even seed the initial state as the value passed in from the previous step in the sequence.

How do we use the `state(..)` helper along with `ASQ#runner(..)`?

```
var prevState;

ASQ(
	/* optional: initial state value */
	2
)
// run our state machine
// transitions: 2 -> 3 -> 1 -> 3 -> false
.runner(
	// state `1` handler
	state( 1, function *stateOne(transition){
		console.log( "in state 1" );

		prevState = 1;
		yield transition( 3 );  // goto state `3`
	} ),

	// state `2` handler
	state( 2, function *stateTwo(transition){
		console.log( "in state 2" );

		prevState = 2;
		yield transition( 3 );  // goto state `3`
	} ),

	// state `3` handler
	state( 3, function *stateThree(transition){
		console.log( "in state 3" );
```

```
                if (prevState === 2) {
                        prevState = 3;
                        yield transition( 1 ); // goto state `1`
                }
                // all done!
                else {
                        yield "That's all folks!";

                        prevState = 3;
                        yield transition( false ); // terminal state
                }
        } )
)
// state machine complete, so move on
.val( function(msg){
        console.log( msg );     // That's all folks!
} );
```

It's important to note that the `*stateOne(..)`, `*stateTwo(..)`, and `*stateThree(..)` generators themselves are reinvoked each time that state is entered, and they finish when you `transition(..)` to another value. While not shown here, of course these state generator handlers can be asynchronously paused by `yield`ing Promises/sequences/thunks.

The underneath hidden generators produced by the `state(..)` helper and actually passed to `ASQ#runner(..)` are the ones that continue to run concurrently for the length of the state machine, and each of them handles cooperatively `yield`ing control to the next, and so on.

Note: See this "ping pong" example (http://jsbin.com/qutabu/1/edit?js,output) for more illustration of using cooperative concurrency with generators driven by `ASQ#runner(..)`.

Communicating Sequential Processes (CSP)

"Communicating Sequential Processes" (CSP) was first described by C. A. R. Hoare in a 1978 academic paper (http://dl.acm.org/citation.cfm?doid=359576.359585), and later in a 1985 book (http://www.usingcsp.com/) of the same name. CSP describes a formal method for concurrent "processes" to interact (aka "communicate") during processing.

You may recall that we examined concurrent "processes" back in Chapter 1, so our exploration of CSP here will build upon that understanding.

Like most great concepts in computer science, CSP is heavily steeped in academic formalism, expressed as a process algebra. However, I suspect symbolic algebra theorems won't make much practical difference to the reader, so we will want to find some other way of wrapping our brains around CSP.

I will leave much of the formal description and proof of CSP to Hoare's writing, and to many other fantastic writings since. Instead, we will try to just briefly explain the idea of CSP in as un-academic and hopefully intuitively understandable a way as possible.

Message Passing

The core principle in CSP is that all communication/interaction between otherwise independent processes must be through formal message passing. Perhaps counter to your expectations, CSP message passing is described as a synchronous action, where the sender process and the receiver process have to mutually be ready for the message to be passed.

How could such synchronous messaging possibly be related to asynchronous programming in JavaScript?

The concreteness of relationship comes from the nature of how ES6 generators are used to produce synchronous-looking actions that under the covers can indeed either be synchronous or (more likely) asynchronous.

In other words, two or more concurrently running generators can appear to synchronously message each other while preserving the fundamental asynchrony of the system because each generator's code is paused (aka "blocked") waiting on resumption of an asynchronous action.

How does this work?

Imagine a generator (aka "process") called "A" that wants to send a message to generator "B." First, "A" `yield`s the message (thus pausing "A") to be sent to "B." When "B" is ready and takes the message, "A" is then resumed (unblocked).

Symmetrically, imagine a generator "A" that wants a message **from** "B." "A" `yield`s its request (thus pausing "A") for the message from "B," and once "B" sends a message, "A" takes the message and is resumed.

One of the more popular expressions of this CSP message passing theory comes from ClojureScript's core.async library, and also from the *go* language. These takes on CSP embody the described communication semantics in a conduit that is opened between processes called a "channel."

Note: The term *channel* is used in part because there are modes in which more than one value can be sent at once into the "buffer" of the channel; this is similar to what you may think of as a stream. We won't go into depth about it here, but it can be a very powerful technique for managing streams of data.

In the simplest notion of CSP, a channel that we create between "A" and "B" would have a method called `take(..)` for blocking to receive a value, and a method called `put(..)` for blocking to send a value.

This might look like:

```
var ch = channel();

function *foo() {
        var msg = yield take( ch );

        console.log( msg );
}

function *bar() {
        yield put( ch, "Hello World" );

        console.log( "message sent" );
}

run( foo );
run( bar );
// Hello World
// "message sent"
```

Compare this structured, synchronous(-looking) message passing interaction to the informal and unstructured message sharing that `ASQ#runner(..)` provides through the `token.messages` array and cooperative `yield`ing. In essence, `yield put(..)` is a single operation that both sends the value and pauses execution to transfer control, whereas in earlier examples we did those as separate steps.

Moreover, CSP stresses that you don't really explicitly "transfer control," but rather you design your concurrent routines to block expecting either a value received from the channel, or to block expecting to try to send a message on the channel. The blocking around receiving or sending messages is how you coordinate sequencing of behavior between the coroutines.

Note: Fair warning: this pattern is very powerful but it's also a little mind twisting to get used to at first. You will want to practice this a bit to get used to this new way of thinking about coordinating your concurrency.

There are several great libraries that have implemented this flavor of CSP in JavaScript, most notably "js-csp" (https://github.com/ubolonton/js-csp), which James Long (http://twitter.com/jlongster) forked (https://github.com/jlongster/js-csp) and has written extensively about (http://jlongster.com/Taming-the-Asynchronous-Beast-with-CSP-in-JavaScript). Also, it cannot be stressed enough how amazing the many writings of David Nolen (http://twitter.com/swannodette) are on the topic of adapting ClojureScript's go-style core.async CSP into JS generators (http://swannodette.github.io/2013/08/24/es6-generators-and-csp).

asynquence CSP emulation

Because we've been discussing async patterns here in the context of my *asynquence* library, you might be interested to see that we can fairly easily add an emulation layer on top of `ASQ#runner(..)` generator handling as a nearly perfect porting of

the CSP API and behavior. This emulation layer ships as an optional part of the "asynquence-contrib" package alongside *asynquence*.

Very similar to the `state(..)` helper from earlier, `ASQ.csp.go(..)` takes a generator -- in go/core.async terms, it's known as a goroutine -- and adapts it to use with `ASQ#runner(..)` by returning a new generator.

Instead of being passed a `token`, your goroutine receives an initially created channel (`ch` below) that all goroutines in this run will share. You can create more channels (which is often quite helpful!) with `ASQ.csp.chan(..)`.

In CSP, we model all asynchrony in terms of blocking on channel messages, rather than blocking waiting for a Promise/sequence/thunk to complete.

So, instead of `yield`ing the Promise returned from `request(..)`, `request(..)` should return a channel that you `take(..)` a value from. In other words, a single-value channel is roughly equivalent in this context/usage to a Promise/sequence.

Let's first make a channel-aware version of `request(..)`:

```
function request(url) {
	var ch = ASQ.csp.channel();
	ajax( url ).then( function(content){
		// `putAsync(..)` is a version of `put(..)` that
		// can be used outside of a generator. It returns
		// a promise for the operation's completion. We
		// don't use that promise here, but we could if
		// we needed to be notified when the value had
		// been `take(..)`n.
		ASQ.csp.putAsync( ch, content );
	} );
	return ch;
}
```

From Chapter 3, "promisory" is a Promise-producing utility, "thunkory" from Chapter 4 is a thunk-producing utility, and finally, in Appendix A we invented "sequory" for a sequence-producing utility.

Naturally, we need to coin a symmetric term here for a channel-producing utility. So let's unsurprisingly call it a "chanory" ("channel" + "factory"). As an exercise for the reader, try your hand at defining a `channelify(..)` utility similar to `Promise.wrap(..)` / `promisify(..)` (Chapter 3), `thunkify(..)` (Chapter 4), and `ASQ.wrap(..)` (Appendix A).

Now consider the concurrent Ajax example using *asyquence*-flavored CSP:

```
ASQ()
.runner(
	ASQ.csp.go( function*(ch){
		yield ASQ.csp.put( ch, "http://some.url.2" );

		var url1 = yield ASQ.csp.take( ch );
		// "http://some.url.1"

		var res1 = yield ASQ.csp.take( request( url1 ) );

		yield ASQ.csp.put( ch, res1 );
	} ),
	ASQ.csp.go( function*(ch){
		var url2 = yield ASQ.csp.take( ch );
		// "http://some.url.2"

		yield ASQ.csp.put( ch, "http://some.url.1" );

		var res2 = yield ASQ.csp.take( request( url2 ) );
		var res1 = yield ASQ.csp.take( ch );

		// pass along results to next sequence step
		ch.buffer_size = 2;
		ASQ.csp.put( ch, res1 );
		ASQ.csp.put( ch, res2 );
```

```
        } )
)
.val( function(res1,res2){
        // `res1` comes from "http://some.url.1"
        // `res2` comes from "http://some.url.2"
} );
```

The message passing that trades the URL strings between the two goroutines is pretty straightforward. The first goroutine makes an Ajax request to the first URL, and that response is put onto the `ch` channel. The second goroutine makes an Ajax request to the second URL, then gets the first response `res1` off the `ch` channel. At that point, both responses `res1` and `res2` are completed and ready.

If there are any remaining values in the `ch` channel at the end of the goroutine run, they will be passed along to the next step in the sequence. So, to pass out message(s) from the final goroutine, `put(..)` them into `ch`. As shown, to avoid the blocking of those final `put(..)`s, we switch `ch` into buffering mode by setting its `buffer_size` to `2` (default: `0`).

Note: See many more examples of using *asynquence*-flavored CSP here (https://gist.github.com/getify/e0d04f1f5aa24b1947ae).

Review

Promises and generators provide the foundational building blocks upon which we can build much more sophisticated and capable asynchrony.

asynquence has utilities for implementing *iterable sequences*, *reactive sequences* (aka "Observables"), *concurrent coroutines*, and even *CSP goroutines*.

Those patterns, combined with the continuation-callback and Promise capabilities, gives *asynquence* a powerful mix of different asynchronous functionalities, all integrated in one clean async flow control abstraction: the sequence.