Asynchronous I/O with Generators & Promises

The article covers designing a JavaScript API to deal with ***asynchronous data inputs and outputs*** *using generators to succinctly describe a set of operations. Promises can also be used to chain other operations after the output has been registered.*

Nicolás Bevacqua

Published a year ago | *8 minute read* | 3

Coming up with practical code examples to keep a book interesting is – for me – one of the hardest aspects of writing engaging material. I find that the best examples are the ones that get you thinking about API design and coding practices, beyond just explaining what a specific language feature does. That way, if you already understand the language feature at hand, you might still find the practical thought exercise interesting.

The example in question involved finding a use case for `return` in a generator function. As we know, generators treat `return` statements differently from `yield` expressions. Take for example the following generator.

```
function* numbers () {
  yield 1;
  yield 2;
  return 3;
  yield 4;
}
```

```
}
```

If we use `Array.from(numbers())` , `[...numbers()]` , or even a `for..of` loop, we'll only ever see `1` and `2` . However, if we went ahead and used the generator object, we'd see the `3` as well – although the iterator result would indicate `done: true` .

```
var g = numbers();
console.log(g.next());
// <- { done: false, value: 1 }
console.log(g.next());
// <- { done: false, value: 2 }
console.log(g.next());
// <- { done: true, value: 3 }
```

The example I came up with involved a function call passing in a generator, where you `yield` resources that should be persisted, and then you `return` the endpoint where you'd like to persist those resources. The iterator would then pull each resource at a time, and finally push the data for each resource to another endpoint, which would presumably save all that data in an object.

The API

The API in question can be found below. The `saveProducts` method would `GET` the JSON description for both products in series, and then `POST` data about the products to a user's shopping cart.

```
saveProducts(function* () {
  yield '/products/javascript-application-design';
  yield '/products/barbie-doll';
  return '/cart';
});
```

In addition, I thought it'd be nice if `saveProducts` also returned a `Promise` , meaning you could chain some other operations to be executed after the products had been saved to the cart.

```
saveProducts(productList)
  .then(data => console.log('Saved', data));
```

Naturally, some conditional logic would allow this hypothetical method to save the products to a wish list instead of onto the shopping cart.

```
saveProducts(function* () {
  yield '/products/javascript-application-design';
  yield '/products/barbie-doll';
  if (addToCart) {
    return '/cart';
  }
  return '/wishlists/nerd-items';
});
```

This example could also apply to the server side, where each yielded value could result in a database query and the returned value could also indicate what kind of object we want to save back to the database. Similarly, the iterator can decide the pace at which yielded inputs are processed: it could be as simple as a synchronous queue, process all queries in parallel, or maybe use a concurrent queue with limited concurrency. Regardless, the API can stay more or less the same *(depending on whether consumers expect to be able to use the product data in the generator itself or not)*.

Implementing `saveProducts`

First off, the method in question takes in a generator and initializes a generator object to iterate over the values produced by the generator function.

```
function saveProducts (productList) {
  var g = productList();
}
```

In a naïve implementation, we could pull each product one by one in an asynchronous series pattern. In the piece of code below, I’m using `fetch` to pull the resources yielded by the user-provided generator – *as JSON*.

```
function saveProducts (productList) {
  var g = productList();
  var item = g.next();
  more();
  function more () {
    if (item.done) {
      return;
    }
    fetch(item.value)
      .then(res => res.json())
      .then(product => {
        item = g.next(product);
        more();
      });
  }
}
```

By calling `g.next(product)` we’re allowing the consumer to read product data by doing `data = yield '/resource'`.

So far we're pulling all data and passing it back, an item at a time to the generator, which has a synchronous feel to it. In order to leverage the `return` statement, we'll save the products in a temporary array and then `POST` them back when we're done iterating.

```
function saveProducts (productList) {
  var products = [];
  var g = productList();
  var item = g.next();
  more();
  function more () {
    if (item.done) {
      save(item.value);
    } else {
      details(item.value);
    }
  }
  function details (endpoint) {
    fetch(endpoint)
      .then(res => res.json())
      .then(product => {
        products.push(product);
        item = g.next(product);
        more();
      });
  }
  function save (endpoint) {
    fetch(endpoint, {
      method: 'POST',
      body: JSON.stringify({ products })
    });
  }
}
```

At this point product descriptions are being pulled down, cached in the `products` array, forwarded to the generator body, and eventually saved in one fell swoop using the endpoint provided by the `return` statement. Where are the promises? Those are very simple to add: `fetch` returns a `Promise`, and it's `return` all the way down.

```
function saveProducts (productList) {
  var products = [];
  var g = productList();
  var item = g.next();
  return more();
  function more () {
    if (item.done) {
      return save(item.value);
    }
    return details(item.value);
  }
  function details (endpoint) {
    return fetch(endpoint)
      .then(res => res.json())
      .then(product => {
        products.push(product);
        item = g.next(product);
        return more();
      });
  }
  function save (endpoint) {
    return fetch(endpoint, {
        method: 'POST',
        body: JSON.stringify({ products })
      })
      .then(res => res.json());
  }
}
```

We're also casting the `save` operation's response as JSON, so that promises chained onto `saveProducts` can leverage response `data`.

As you may notice the implementation doesn't hardcode any important aspects of the operation, which means you could use something like this pretty generically, as long as you have zero or more inputs you want to pipe into one output. The consumer ends up with an elegant-looking method that's easy to understand – they `yield` input stores and `return` an output store. Furthermore, our use of promises makes it easy to concatenate this operation with others. This way, we're keeping a potential tangle of conditional statements and flow control mechanisms in check, by abstracting away flow control into the iteration mechanism under the `saveProducts` method.