Roll your own async compose & pipe functions

JP Follow
Sep 19, 2017 · 6 min read

The functional programming inspired functions `compose` and `pipe` are awesome, but they only work with synchronous functions. I am going to show you how to write your own functions that will work with both synchronous and asynchronous functions combined, if that's your type of thing.

Cool story bro, just show me the code already!

Fine, here you go.

```
nc compose
compose = (…functions) => input => functions.reduceRight((chain, func) => chain.then(func), Pro

ctions fn1, fn2, fn3 can be standard synchronous functions or return a Promise
e(fn3, fn2, fn1)(input).then(result => console.log(`Do with the ${result} as you please`))
```

compose.js hosted with ❤ by GitHub view raw

Async compose function

```
const pipe = (…functions) => input => functions.reduce((chain, func) => chain.then(func), Prom

// Functions fn1, fn2, fn3 can be standard synchronous functions or return a Promise
pipe(fn1, fn2, fn3)(input).then(result => console.log(`Do with the ${result} as you please`))
```

pipe.js hosted with ❤ by GitHub view raw

Async pipe function

Right, now what?

If you came here just to get a code snippet, there is no shame in that, feel free to add it to your project and go on with your life! Otherwise, if you're interested in knowing how it works, continue reading.

It might get a bit hairy but bare with me. If you are familiar with certain parts, just skip ahead.

These functions require `Promises` to be available in your application environment, whether that is the browser or server side with Node. If you are developing for the modern web (not including IE), Promises are already supported in most browsers. Back in the real world, you will probably use a polyfill or some other third-party library, like bluebird, that gives you `Promise` support. Odds are that if you're reading this you are probably already familiar with Promises, if not, here is a quick primer.

Simplified Primer on Promises

A `Promise` is an object that represents an action that will finish at some time in the future. What that means, is that you create your `Promise` for an asynchronous action, like an AJAX request, and when it finishes, similar to using a success callback, you call its `resolve` function; if it fails, you call its `reject` function.

The `Promise` object exposes 2 functions that map to the 2 "callback" functions, namely: `.then()` that will be called when you `resolve` the promise and `.catch()` when you `reject` it.

There's loads more to `Promises` but if you understand that, you'll be just fine.

What's the deal with compose & pipe?

On Medium alone there are numerous articles about the intricacies of `compose` and `pipe` which does a great job at explaining it. If you're too lazy to google it right now and seeing that you are already here, I'll try and explain: composition is like Lego for grown-ups. It's a way to build something more complex by using smaller simpler units. In functional programming, `compose` is the mechanism that composes the smaller units (our functions) into something more complex (you guessed it, another function).

So, `compose` takes a list of functions and returns another function that you can call with your data. An example will clear it up:

```
1  const double = x => x * 2;
2  const square = x => x * x;
3  const plus3 = x => x + 3;
4
5  const composedFunction = compose(double, square, plus3);
6
7  const result = composedFunction(2);
8  // result = 50
9
10 // 2 (plus3) => 5 (square) => 25 (double) => 50
```

compose-example.js hosted with ❤ by **GitHub** **view raw**

To keep things simple, we create 3 basic functions that do simple arithmetics: `double`, `square` and `plus3`. Nothing special here, `double` takes a value and multiplies it by 2; `square` takes a value and multiplies it with itself, and finally, `plus3` does exactly that — it takes a number and adds 3 to it.

On line 5, we use our `compose` method to create a new `composedFunction` that is build up of our 3 simpler functions. Each time this `composedFunction` is invoked it will apply these 3 functions to whatever value we pass to it. On line 7, we invoke our `composedFunction` with the value 2. To visualize what is happening here, picture this:

```
double(square(plus3(2)))
```

That is ugly, but it brings the point across.

The value 2, is passed into the `plus3` function which produces 5. This result becomes the input for our `square` function, which in turn produces 25. Lastly, we invoke the `double` function with 25, which then produces 50.

This is obviously a ridiculously simple example, but the main point is to think of complex problems as the sum of a bunch of smaller problems that you can solve and `compose` together.

Now, imagine you have a response from an API call to retrieve a list of products, you want to map over each product and only pluck certain properties like the `title`, `description` and let's say `price`. Then you also want to filter out all the products that are cheaper than `$5` and lastly sort the results alphabetically.

The old you would write 3 functions: `pluckProperties`, `getCheap` and `sortByTitle` and do something like this:

```
const products = // array of product objects

const result = pluckProperties(sortByTitle(getCheap(products)))
```

Bonus Tip: *always filter first before doing anything else with a dataset, so that you loop through the smallest number of entries.*

The new you will use `compose` to create your reusable `getProducts` function:

```
const getProducts = compose(pluckProperties, sortByTitle, getCheap);

const products = // array of product objects
const result = getProducts(products)
```

Hopefully, you are seeing the beauty in using `compose` by now. If not, all hope is lost and I can't help you. Jokes. Keep thinking about it and see where it will help you write cleaner more reusable (read: **better!**) code in your day job.

We'll quickly touch `pipe` as well. It is exactly the same as `compose` but it applies the functions from left-to-right instead of right-to-left.

Using `pipe`, our example will look like this:

```
const getProducts = pipe(getCheap, sortByTitle, pluckProperties);

const products = // array of product objects
const result = getProducts(products)
```

`pipe` is very useful for writing step by step procedures, like:

```
pipe(logUserIn, displayNotification, redirectToHomepage)(user)
```

What's cool about async `compose` and `pipe`?

Traditionally, `compose` and `pipe` only works on synchronous functions that you can pass an input value, it does something with the input and then returns an output. Looking at the above `pipe` example, you will agree that the `logUserIn` function will most likely be an asynchronous function because you need some communication with a server/database. This won't work then. Time to cry in the shower.

Let's fix that by using our own improved `pipe` function:

```
pipe(logUserIn, displayNotification, redirectToHomepage)(user)
  .then(() => {
    // The user is logged in
    // the login notification has been displayed
    // and s/he has been redirected to the homepage
  })
```

That's it, you're done!

Let's break it down

For reference, here is the `compose` snippet again.

```
=> input => functions.reduceRight((chain, func) => chain.then(func), Promise.resolve(input));

n be standard synchronous functions or return a Promise
.then(result => console.log(`Do with the ${result} as you please`))
```

compose.js hosted with ❤ by **GitHub** **view raw**

First, we see `compose` is a function that returns a function (example uses standard JavaScript functions):

```
const compose = function(...functions) {
  return function(input) {
    // ...
  };
}
```

It uses the ES2015 rest parameters to combine all the passed in functions as an array of `functions`.

To understand the body of the inner function, it is important that you are comfortable with the `reduce` and `reduceRight` array methods that are natively part of JavaScript.

`reduce` loops over each item in a given array and apply a function to it, with each function's result being the input of the next item's function. Whenever you have a list of something and you want to "reduce" it to a single value, use the `reduce` or `reduceRight` methods. These methods take 2 arguments, the first is the function that needs to run for each item in the array and the second is the optional starting value.

The easiest is to think of it as a `SUM()` or a `TOTAL()` method, for example:

```
const numbers = [ 1, 5, 9 ];
const total = items.reduce(function(sum, number) {
  return sum + number;
}, 0);

// total = 15 (1 + 5 + 9)
```

The output of each function is the input of the next. For the first number in the array (1), `sum` will be `0` as that is the starting value and `number` will be `1`. The second number (5) will take `1` as the value of `sum` and `number` will be `5`, which produces `6`. For the third number (9), `sum` will, therefore, be `6` and `number` will be `9`, which produces the final output of `15`.

The only difference between the `reduce` and `reduceRight` is that `reduceRight` loops over the items in your array from right-to-left (last-to-first) instead of left-to-right (first-to-last).

Okay, now that we understand `reduce` and `Promises`, let's put them together to understand the final piece of our `compose` function.

```
functions.reduceRight(
  (chain, func) => chain.then(func),
  Promise.resolve(input)
)
```

Instead of looping through an array of `numbers` as we did in our example, here we loop over an array of `functions`. Instead of starting with the value `0`, here we start with a `Promise` that immediately resolves to the value we called our composed function with.

Inside our reduce/accumulator function, instead of building up a `sum`, we are chaining together Promises that will resolve in sequence. To visualize this using our user login example, this is what would be produced:

```
logUserIn(user)
  .then(displayNotification)
  .then(redirectToHomepage)
  .then(result => `Do whatever we want with the ${result}`)
```

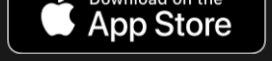
Wrapping up

Hope you found this useful and if you want to include these functions in your project, either copy-paste it in or grab it from NPM:

- https://www.npmjs.com/package/compose-then
- https://www.npmjs.com/package/fp-pipe-then

If you have better ideas or something cool to share, please let me know; would love to learn from you.

JavaScript | Functional Programming | Composition | Promises | DIY


Download on the
App Store

Google Play