RxJS: Combining Operators

Nicholas Jamieson Follow
May 14, 2018 · 2 min read

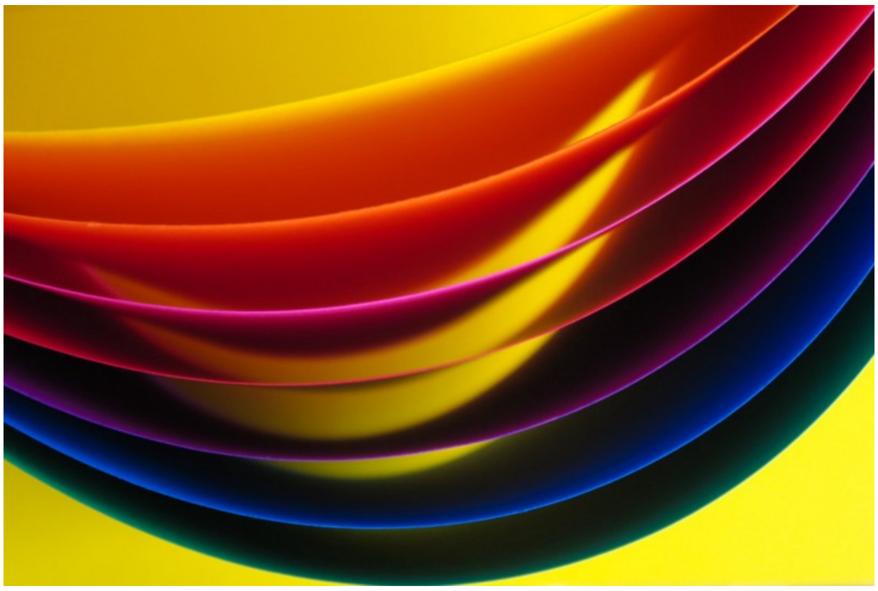
Photo by Daniele Levis Pelusi on Unsplash

In version 5.5, pipeable operators were added to RxJS. And in version 6, their non-pipeable namesakes were removed.

Pipeable operators have numerous advantages. The most obvious is that they are easier to write. A less obvious advantage is that they can be composed into reusable combinations.

Let's have a look at how code can be simplified by combining operators.

Combining multiple operators

Debouncing user input—to avoid the execution of an operation every time the user presses a key—is a common use case for RxJS. And doing so usually involves the `debounceTime` and `distinctUntilChanged` operators.

If an app performs a lot of debouncing, combining the two into a single operator can be a worthwhile simplification.

Here's one way the two operators can be combined:

```
import { Observable } from "rxjs";
import { debounceTime, distinctUntilChanged } from "rxj

const debounceInput = (changes: Observable<string>) =>
  debounceTime(400),
  distinctUntilChanged()
```

`debounceInput` is a function that takes and returns an observable, so it can be passed to the `Observable.prototype.pipe` function, like this: `valueChanges.pipe(debounceInput)`.

This combination of the `debounceTime` and `distinctUntilChanged` operators can itself be simplified using RxJS's general-purpose, static `pipe` function—which can be used like this:

```
import { pipe } from "rxjs";
import { debounceTime, distinctUntilChanged } from "rxj

const debounceInput = pipe(
  debounceTime<string>(400),
  distinctUntilChanged()
```

The returned `debounceInput` function is identical to its namesake function in the first code snippet and can be passed to `Observable.prototype.pipe`.

So, whenever you find yourself using the same combination of operators in many places, you could consider using the static `pipe` function to create a reusable operator combination.

The static `pipe` function also makes something else much simpler: dealing with `pipe`-like overload signatures. Let's look at that next.

Implementing a pipe-like API

The `Observable.prototype.pipe` and static `pipe` functions have a lot of TypeScript overload signatures. And authoring an API that behaves in a `pipe`-like manner requires a similar number of overload signatures.

The signatures for such an API end up looking something like this:

```typescript
import { Observable, OperatorFunction } from "rxjs";

/* ... */

interface Collection {
  traverse(): Observable<Document>;
  traverse<A>(
    op1: OperatorFunction<Document, A>): Observable<A>
  traverse<A, B>(
    op1: OperatorFunction<Document, A>,
    op2: OperatorFunction<A, B>): Observable<B>;
  traverse<A, B, C>(
    op1: OperatorFunction<Document, A>,
    op2: OperatorFunction<A, B>,
    op3: OperatorFunction<B, C>): Observable<C>;
  /* ... 5 signatures elided ... */
  traverse<A, B, C, D, E, F, G, H, I>(
    op1: OperatorFunction<Document, A>,
```

Here, the `traverse` function can be passed numerous operators, which will be connected—as they would be by `Observable.prototype.pipe` —and injected into the observable composed within the function's implementation. (The reason for injecting the operators—rather than appending them to the returned observable—is so that the operators can control backpressure during the traversal. We'll look at controlling backpressure with RxJS in a future article.)

There is an alternative API that's just as flexible and doesn't involve declaring all of those overload signatures. `traverse` could instead be declared with just two overload signatures, like this:

```typescript
import { Observable, OperatorFunction } from "rxjs";

/* ... */

interface Collection {
  traverse(): Observable<Document>;
```

With the alternative API, even though the function's overload signatures allow only a single operator to be passed, callers can use the static `pipe` function to combine any number of operators and can pass the result, like this:

```
import { pipe } from "rxjs";
import { ajax } from "rxjs/ajax"
import { concatMap, filter, ignoreElements } from "rxj

/* ... */

collection.traverse(pipe(
  filter(shouldPut),
  concatMap(doc => ajax.put(
    `${uri}/docs/${doc.id}`,
    doc.toJSON(),
    { "Content-Type": "application/json" }
```

Which is great, because having to otherwise declare all of those `pipe` -like overload signatures is beyond tedious.

• • •

My next article takes operator composition a little further and looks at: Improving the Static pipe Function.