RxJS: Understanding Lettable Operators

Nicholas Jamieson Follow
Sep 26, 2017 · 4 min read

Photo by Steven Wang on Unsplash

An update: lettable operators are now officially pipeable operators.

· · ·

The version 5.5.0 beta of RxJS introduces lettable operators.

Lettable operators offer a new way of composing observable chains and they have advantages for both application developers and library authors. Let's look briefly at the existing composition mechanisms in RxJS and then look at lettable operators in more detail.

Using the bundle

When the entire—and rather large—RxJS bundle is imported, all operators will have been added to `Observable.prototype`. So observables can be composed by chaining the operator methods, like this:

```
import * as Rx from "rxjs";

const name = Rx.Observable.ajax
  .getJSON<{ name: string }>("/api/employees/alice")
  .map(employee => employee.name)
```

The obvious disadvantage with this approach is that the applications will contain everything that's in RxJS, even if it's not required.

Using prototype patching

RxJS includes—under its `add` directory—modules that patch `Observable` and `Observable.prototype`. These can be imported on a per-operator basis, allowing developers to import only what's needed. As with the bundle approach, observables can be composed by chaining the operator methods, like this:

```
import { Observable } from "rxjs/Observable";
import "rxjs/add/observable/dom/ajax";
import "rxjs/add/operator/catch";
import "rxjs/add/operator/map";

const name = Observable.ajax
  .getJSON<{ name: string }>("/api/employees/alice")
```

The disadvantage with this approach is that developers are burdened with managing the prototype-patching imports.

Using call

Library authors are discouraged from patching `Observable.prototype`, as doing so creates a dependency. That is, if a library patches `Observable.prototype` with an operator, any consumers of the library that depend upon the operator being present will break if the library implementation changes and the operator is no longer patched.

Instead, library authors are encouraged to import the operator methods and invoke them using `Function.prototype.call`. So composing observables looks like this:

```
import { ajax } from "rxjs/observable/dom/ajax";
import { of } from "rxjs/observable/of";
import { _catch } from "rxjs/operator/catch";
import { map } from "rxjs/operator/map";

const source = ajax.getJSON<{ name: string }>("/api/emp
```

The disadvantage with this approach—apart from the verbosity—is that `Function.prototype.call` returns `any`, so the operator's type information is lost. In this example, the inferred type of `mapped` and `name` will be `any`.

Using lettable operators

As with the `call`-based approach, lettable operators are imported explicitly. They can be imported from the `operators` directory (note the plural) and a tree-shaking bundler can be used to ensure that only operators that are used are bundled. Or, they can be imported from operator-specific directories—like `operators/map`.

To facilitate the composition of observables, `Observable.prototype` includes a new method—`pipe`—which accepts an arbitrary number of lettable operators.

The `pipe`-based approach to observable composition is less verbose than the `call`-based approach and it allows the correct types to be inferred. It looks like this:

```
import { ajax } from "rxjs/observable/dom/ajax";
import { of } from "rxjs/observable/of";
import { catchError, map } from "rxjs/operators";

const name = ajax
  .getJSON<{ name: string }>("/api/employees/alice")
  .pipe(
    map(employee => employee.name)
```

In this example, the inferred type of `name` will be `Observable<string | null>`.

What are lettable operators and what does lettable mean?

If lettable operators are used with a method named `pipe`, you might wonder why they are referred to as lettable. The term is derived from RxJS's `let` operator.

The `let` operator is conceptually similar to the `map` operator, but instead of taking a projection function that receives and returns a value, `let` takes a function that receives and returns an observable. It's unfortunate that `let` is one of the less-well-known operators, as it's very useful for composing reusable functionality.

Let's look at an example. Sometimes, when requesting resources from a HTTP API, it's desirable to retry if an error occurs. Let's write a `retry` function that will return a function that can be passed to the `let` operator. Our `retry` function will look like this (using the bundle, to keep the example's imports to a minimum):

```
import * as Rx from "rxjs";

export function retry<T>(
  count: number,
  wait: number
): (source: Rx.Observable<T>) => Rx.Observable<T> {

  return (source: Rx.Observable<T>) => source
    .retryWhen(errors => errors
      // Each time an error occurs, increment the accu
      // When the maximum number of retries have been
      .scan((acc, error) => {
        if (acc >= count) { throw error; }
```

When `retry` is called, it's passed the number of retry attempts that should be made and the number of milliseconds to wait between attempts, and it returns a function that receives an observable and returns another observable into which the retry logic is composed. The returned function can be passed to the `let` operator, like this:

```
import * as Rx from "rxjs";
import { retry } from "./retry";

const name = Rx.Observable.ajax
  .getJSON<{ name: string }>("/api/employees/alice")
  .let(retry(3, 1000))
```

Using the `let` operator, we've been able to create a reusable function much more simply than we would have been able to create a prototype-patching operator. What we've created is a lettable operator.

Lettable operators are a higher-order functions. Lettable operators return functions that receive and return observables; and those functions can be passed to the `let` operator.

We can also use our lettable `retry` operator with `pipe` , like this:

```
import { ajax } from "rxjs/observable/dom/ajax";
import { of } from "rxjs/observable/of";
import { catchError, map } from "rxjs/operators";
import { retry } from "./retry";

const name = ajax
  .getJSON<{ name: string }>("/api/employees/alice")
  .pipe(
    retry(3, 1000),
```

Let's return to our `retry` function and replace the chained methods with lettable operators and a `pipe` call, so that it looks like this:

```
import { Observable } from "rxjs/Observable";
import { delay, retryWhen, scan } from "rxjs/operators

export function retry<T>(
  count: number,
  wait: number
): (source: Observable<T>) => Observable<T> {

  return retryWhen(errors => errors.pipe(
    // Each time an error occurs, increment the accumu
    // When the maximum number of retries have been at
    scan((acc, error) => {
      if (acc >= count) { throw error; }
```

With the chained methods replaced, we now have a proper, reusable lettable operator that imports only what it requires.

Why should lettable operators be preferred?

For application developers, lettable operators are much easier to manage:

- Rather then relying upon operators being patched into `Observable.prototype` , lettable operators are explicitly imported into the modules in which they are used.
- It's easy for TypeScript and bundlers to determine whether the lettable operators imported into a module are actually used. And if they are not, they can be left unbundled. If prototype patching is used, this task is manual and tedious.

For library authors, lettable operators are much less verbose than `call` -based alternative, but it's the correct inference of types that is —at least for me—the biggest advantage.

Switching to lettable operators

Some months ago, I wrote an article—Managing RxJS Imports with TSLint—that introduced set of TSLint rules I compiled to help make managing RxJS prototype-patching imports a little easier.

The article outlines rule combinations that can be used to enforce various import policies. The rule combination to use to enforce a lettable-operator-only policy would be:

```
"rxjs-no-add": { "severity": "error" },
"rxjs-no-operator": { "severity": "error" },
"rxjs-no-patched": { "severity": "error" },
"rxjs-no-wholesale": { "severity": "error" }
```

The rules are in an npm package: `rxjs-tslint-rules` .

If you've decided to switch to lettable operators, you might find the rules useful. You could initially configure the rules to warn, gradually change your code base to use lettable operators, and then re-configure the rules to error.

· · ·

The `pipe` method in RxJS is related to the ECMAScript pipeline operator proposal. For a look at their relationship, see *RxJS: Pipelining Lettable Operators*.