RxJS Marble Testing: RTFM

Nicholas Jamieson Follow
Jul 29, 2017 · 5 min read

Last week, I finally got around to something that I'd been putting off for far too long: incorporating marble tests into projects in which I am using RxJS.

I'm not going to go into detail about the syntax with which marble tests are declared—as that's covered in the *Writing Marble Tests* document in the RxJS repo—but I will go over a few aspects of marble testing that confused me more than they would have if I had read the manual with greater care. Also, I'll introduce a marble testing package I've published that can be used with any test framework.

Marble Tests

The marble tests below call the Mocha-based, basic methods that are used throughout the RxJS code base. Being opinionated regarding the test framework, they are not included in the RxJS distribution. Essentially, the basic methods are thin wrappers around the `TestScheduler` and they reduce the boilerplate that would otherwise be required by the tests.

A simple test looks something like this:

```
const a =   cold("--1--2--|");
const asub =     "^-------!";
const expected = "--2--3--|";

const result = a.map(s => `${Number(s) + 1}`);
expectObservable(result).toBe(expected);
expectSubscriptions(a.subscriptions).toBe(asub);
```

There’s a single, cold source named `a` that will start emitting values upon subscription, so the marble diagrams for the subscription to `a` and the expected result can be aligned with the beginning of the marble diagram for `a`. The `result` observable just maps the emitted values, so the expected values and completion are also aligned with those of `a`.

The value characters declared within the marble diagrams are emitted as strings regardless of whether the characters are numeric or alphabetical—unless the optional `values` argument is specified in the `cold` call.

A test with more than one source better illustrates the diagram alignments:

```
const a =   cold("--1--2--|");
const b =   cold(        "--3--|");
const asub =     "^-------!";
const bsub =     "--------^----!";
const expected = "--1--2----3--|";

const result = a.concat(b);
expectObservable(result).toBe(expected);
expectSubscriptions(a.subscriptions).toBe(asub);
expectSubscriptions(b.subscriptions).toBe(bsub);
```

The `result` observable is the source named `b` concatenated onto the source named `a`, so the diagram is easier to understand if `b` is aligned with the completion of `a` —as that’s the frame at which the subscription to `b` will occur.

The subscription and expected marble diagrams are always aligned with the first frame—referred to as the zero frame.

Hot Observables

The situation starts to get more complicated (and confusing, if you don’t read the documentation carefully) when hot observables are introduced:

```
const a =   cold("--1--2--|");
const b =    hot("^----3----4--|");
const asub =     "^-------!";
const bsub =     "--------^----!";
const expected = "--1--2----4--|";

const result = a.concat(b);
expectObservable(result).toBe(expected);
expectSubscriptions(a.subscriptions).toBe(asub);
expectSubscriptions(b.subscriptions).toBe(bsub);
```

Hot observables include a `^` character that indicates (according to the documentation) the subscription point—*"the point at which the tested observables will be subscribed to the hot observable."*

I found that confusing, as the tested observable doesn't subscribe to the hot observable named `b` until the observable named `a` completes.

It seems that the mentioned subscription is the testing infrastructure's implicit subscription to the hot observable. To me, it makes more sense to think of the `^` character in a hot observable as the indicator of the zero frame's position and nothing more.

The initial subscription in the test is to `a`, so it's aligned with the hot observable's zero frame—as are the diagrams for the subscriptions and the expected result.

The hot observable's `3` is emitted prior to subscription, so it does not appear in the expected result.

Hot observables can emit values prior to the zero frame, but their doing so doesn't change the test—only the position of the zero frame and the alignment of the other diagrams relative to it:

```
const a = cold(    "--1--2--|");
const b =  hot("0--^----3----4--|");
const asub =       "^-------!";
const bsub =       "--------^----!";
const expected =   "--1--2----4--|";

const result = a.concat(b);
expectObservable(result).toBe(expected);
expectSubscriptions(a.subscriptions).toBe(asub);
expectSubscriptions(b.subscriptions).toBe(bsub);
```

Indeed, because no subscriptions occur before the zero frame, prior values are ignored by the testing infrastructure:

```
const a = hot("0--^----1----2--|");
const expected = "-----1----2--|";

expectObservable(a).toBe(expected);
```

Synchronous Notifications

To indicate the synchronous emission of notifications within a frame, value characters (and errors or completions) are grouped within parentheses:

```
const a =   cold("-1-----2----|");
const b =   cold("-3-----4----|");
const asub =     "^-----------!";
const bsub =     "^-----------!";
const expected = "-(13)--(24)-|";

const result = a.merge(b);
expectObservable(result).toBe(expected);
expectSubscriptions(a.subscriptions).toBe(asub);
expectSubscriptions(b.subscriptions).toBe(bsub);
```

In the above test, `1` and `3` are emitted in the frame within which the group's opening parenthesis is placed. However, each of the parenthesis-enclosed groups of values spans four actual frames. That means there must be a sufficient number of frames separating the values emitted by the source observables named `a` and `b`, as the groups in the `expected` observable diagram cannot overlap. (It took me far too long to figure out that groups span actual frames, but it's all in the documentation.)

Schedulers

Some of the `Observable` methods have an optional parameter that accepts a scheduler. Methods that have a non-`null` default scheduler will need to be passed a `TestScheduler` instance if they are used in a marble test. The defaults for the scheduler parameters are indicated in the documentation.

For example, `concat` has an optional scheduler parameter that defaults to `null`, so it's not necessary to specify a scheduler in tests.

However, `delay` has an optional scheduler parameter that defaults to `async`, so it is necessary to specify a scheduler in tests:

```
const a =   cold("-1-----2---|");
const asub =     "^------------!";
const expected = "---1-----2---|";

const result = a.delay(time("--|"), rxTestScheduler);
expectObservable(result).toBe(expected);
expectSubscriptions(a.subscriptions).toBe(asub);
```

Because the `TestScheduler` has its own frame-based units of virtual time, arguments such as the number of milliseconds passed to the `delay` operator must use those units. Otherwise, the resulting observable would not be comparable with an expected marble diagram. There is a basic method named `time` for that purpose.

When testing code that calls RxJS methods that have non-`null` default schedulers, it's necessary to provide a mechanism for injecting a `TestScheduler`. Also, any code that relies upon the current time will need to use a scheduler's `now()` method instead of `Date.now()`.

Alternative Test Frameworks

The projects into which I wanted to incorporate marble testing used different test frameworks. There are a number of marble testing packages that contain equivalents of the basic methods, but I wanted an implementation that was consistent across test frameworks.

Instead of extending each framework to include the basic methods, I wrapped the `TestScheduler` in a method that provides a context to the code under test. And that context includes the basic methods. This approach has effected a small, simple, framework-independent package.

The package is named `rxjs-marbles` and the code is on GitHub; it's installable via NPM. It can be used with Mocha, Jasmine, Jest, AVA or Tape; and in Mocha and Jasmine, the tests look like this:

```
it("should map the values", marbles(m => {

  const a = m.cold("--1--2--|");
  const asub =     "^-------!";
  const expected = "--2--3--|";

  const result = a.map(s => `${Number(s) + 1}`);
  m.expect(result).toBeObservable(expected);
  m.expect(a).toHaveSubscriptions(asub);
}));
```

Marble tests are awesome and I regret delaying their inclusion in my projects.

I also regret not reading the manual more carefully.