Debugging RxJS, Part 2: Logging

Nicholas Jamieson Follow
Aug 4, 2017 · 4 min read

Logging is not exciting.

However, it is a straightforward method for obtaining enough information to beginning reasoning about a problem, without resorting to outright guessing. And it's often the go-to approach for debugging RxJS-based code.

This is the second in a series of articles—following *Debugging RxJS, Part 1: Tooling*, which introduced `rxjs-spy` —and is focused on using logging to solve actual problems. In this article, I'll show how `rxjs-spy` can be used to obtain detailed and targeted information in an unobtrusive way.

Let's look at a simple example that uses the `rxjs` and `rxjs-spy` UMD bundles:

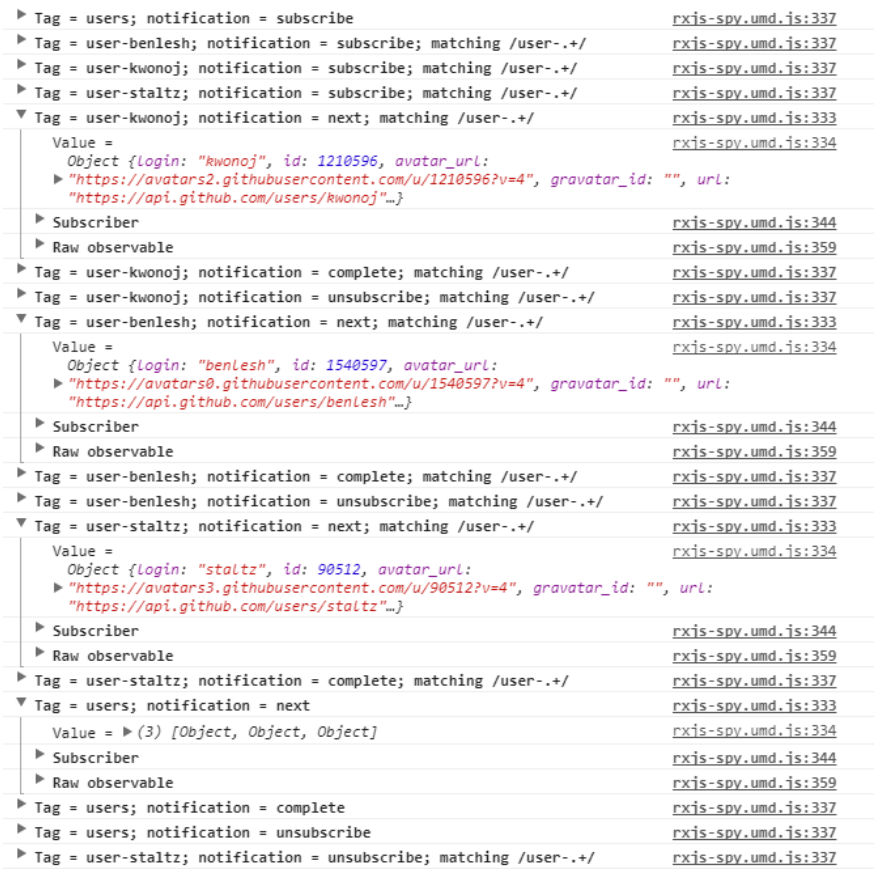
```
RxSpy.spy();
RxSpy.log(/user-.+/);
RxSpy.log('users');

const names = ['benlesh', 'kwonoj', 'staltz'];
const users = Rx.Observable.forkJoin(...names.map(name
  Rx.Observable
    .ajax
    .getJSON(`https://api.github.com/users/${name}`)
    .tag(`user-${name}`)
```

The example uses `forkJoin` to compose an observable that emits an array of GitHub users.

`rxjs-spy` works with observables that have been tagged using its `tag` operator—which annotates an observable with a string tag, and nothing more. Before the observable is composed, the example enables spying and configures loggers for tagged observables with tags that match the `/user-.+/` regular expression or observables that have a `users` tag.

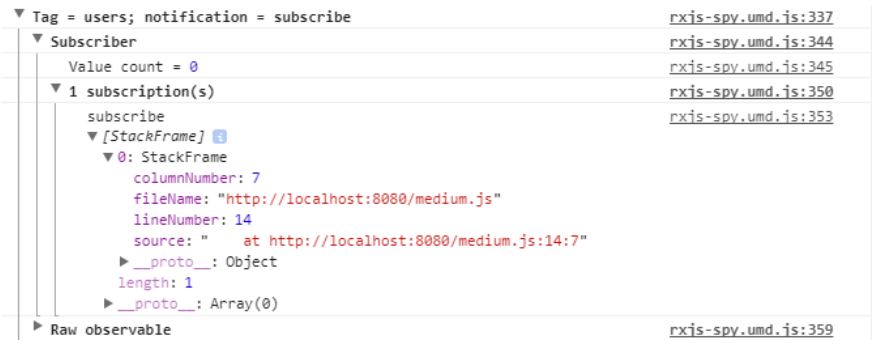
The example's console output looks like this:



In addition to the observable `next` and `complete` notifications, the logged output includes notifications for subscriptions and unsubscriptions. And it shows everything that occurs:

- the subscription to the composed observable effects parallel subscriptions to the observable for the API request for each user;
- the requests complete in any order;
- the observables all complete;
- and the subscription to the composed observable is automatically unsubscribed upon completion.

Each logged notification also includes information about the subscriber that received the notification—including the number of subscriptions the subscriber has and the stack trace for the `subscribe` call:



The stack trace refers to the root `subscribe` call—that is, the explicit call that effected the subscriber's subscription to the observable. So the stack traces for the user request observables also refer to the `subscribe` call made in `medium.js`:

When I'm debugging, I find that knowing the location of the actual, root `subscribe` call is more useful than knowing the location of a `subscribe` made somewhere in the middle of a composed observable.

Let's now look at a real world problem.

When writing `redux-observable` epics—or `ngrx` effects—I've seen several developers write code similar to this:

```
1  import { Observable } from 'rxjs/Observable';
2  import { ajax } from 'rxjs/observable/dom/ajax';
3
4  const getRepos = action$ =>
5    action$.ofType('REPOS_REQUEST')
6      .map(action => action.payload.user)
7      .switchMap(user => ajax.getJSON(`https://api.notgi
```

At first glance, it looks okay. And it works okay, too, most of the time. It's also the sort of bug that sneaks past unit tests.

The problem is that sometimes the epic just stops working. In particular, it stops working after an error action is dispatched.

Logging shows what's happening:

```
▶ Tag = getRepos; notification = subscribe                    bundle.js:58942
✖ ▶OPTIONS https://api.notgithub.com/users/cartant/repos      bundle.js:9141
  net::ERR_NAME_NOT_RESOLVED
▼ Tag = getRepos; notification = next                         bundle.js:58938
    Value = ▶Object {type: "REPOS_ERROR"}                     bundle.js:58939
  ▶ Subscriber                                                bundle.js:58949
  ▶ Raw observable                                            bundle.js:58964
▶ Tag = getRepos; notification = complete                     bundle.js:58942
▶ Tag = getRepos; notification = unsubscribe                  bundle.js:58942
>
```

After the error action is emitted, the observable completes—which sees the `redux-observable` infrastructure unsubscribe from the epic. The documentation for `catch` explains why this occurs:

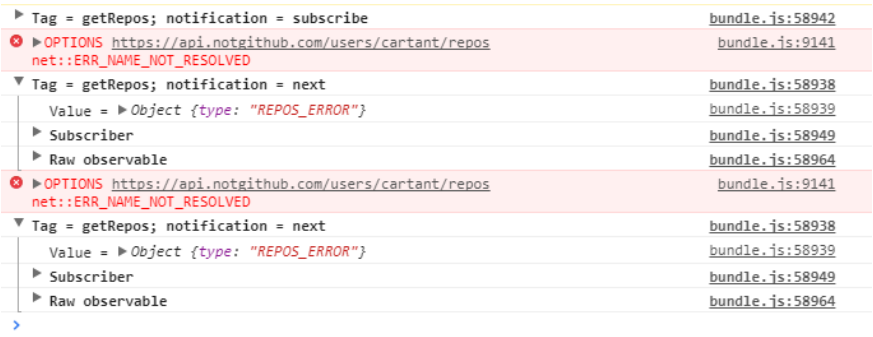
> *Whatever observable is returned by the* `selector` *will be used to continue the observable chain.*

In the epic, the observable returned by `catch` completes—which sees the epic complete, too.

The solution is to move the `map` and `catch` calls into the `switchMap`, like this:

```
1  import { Observable } from 'rxjs/Observable';
2  import { ajax } from 'rxjs/observable/dom/ajax';
3
4  const getRepos = action$ =>
5    action$.ofType('REPOS_REQUEST')
6      .map(action => action.payload.user)
7      .switchMap(user => ajax
8        .getJSON(`https://api.notgithub.com/users/${user
9        .map(repos => { type: 'REPOS_RESPONSE', payload:
```

The epic will then no longer complete and will continue to dispatch error actions:



In both of these examples, the only modification that needed to be made to the code being debugged was the addition of some tag annotations.

The annotations are light-weight and, once added, I tend to leave them in the code. The tag operator can be consumed independently of the diagnostics in `rxjs-spy` —using either `rxjs-spy/add/operator/tag` or a direct import from `rxjs-spy/operator/tag` —so there is little overhead in keeping the tags.

The loggers can be configured using regular expressions, which leads to a number of possible tagging approaches. For example, using compound tags like `github/users` and `github/repos` would allow you to switch on logging for all `github` observables for just for those that deal with repositories.

Logging is not exciting, but the information that can be gleaned from logged output can often save an immense amount of time. And

adopting a flexible approach to tagging can further reduce the amount of time you spend dealing with logging-related code.