RxJS: Avoiding switchMap-Related Bugs

Nicholas Jamieson Follow
Mar 12, 2018 · 5 min read

A while ago, Victor Savkin tweeted about a subtle bug that occurs through the misuse of `switchMap` in NgRx effects in Angular applications:

> Every single Angular app I've looked at has a lot of bugs due to an incorrectly used switchMap. It's the biggest source of RxJS-related issues. #angular
>
> —@victorsavkin

So what's the bug?

Let's use a shopping cart as an example and have a look at an effect—and an epic—that misuses `switchMap` and then consider some alternative operators.

Here's an NgRx effect that misuses `switchMap`:

```
@Effect()
public removeFromCart = this.actions.pipe(
  ofType(CartActionTypes.RemoveFromCart),
  switchMap(action => this.backend
    .removeFromCart(action.payload)
    .pipe(
      map(response => new RemoveFromCartFulfilled(resp
      catchError(error => of(new RemoveFromCartRejecte
```

And here's its equivalent `redux-observable` epic:

```
const removeFromCart = actions$ => actions$.pipe(
  ofType(actions.REMOVE_FROM_CART),
  switchMap(action => backend
    .removeFromCart(action.payload)
    .pipe(
      map(response => actions.removeFromCartFulfilled(
      catchError(error => of(actions.removeFromCartRej
```

Our shopping cart lists the items the user intends to purchase and against each item is a button that removes the item from the cart. Clicking the button dispatches a `RemoveFromCart` action to the effect/epic which communicates with the application's backend and sees the item removed from the cart.

Most of the time, this will function as intended. However, the use of `switchMap` has introduced a race condition.

If the user clicks the remove buttons for several items in the cart, what happens depends upon how rapidly the buttons are clicked.

If a remove button is clicked when the effect/epic is communicating with the backend—that is, when a removal is pending—the use of `switchMap` will see the pending removal aborted.

So, depending upon how rapidly the buttons are clicked, the application might:

- remove all of the clicked items from the cart;

- remove only some of the clicked items from the cart; or
- remove some of the clicked items from the backend's cart but not reflect their removal from the frontend's cart.

Clearly, this is a bug.

It's unfortunate that `switchMap` is often suggested as the first choice when a flattening operator is required, as it's not safe for all scenarios.

RxJS has four flattening operators to choose from:

- `mergeMap` (also known as `flatMap`);
- `concatMap`;
- `switchMap;` and
- `exhaustMap`.

Let's have a look at these operators to see how they differ and to determine the shopping-cart scenarios for which each operator is best suited.

mergeMap/flatMap

If `switchMap` is replaced with `mergeMap`, the effect/epic will concurrently handle each dispatched action. That is, pending removals will not be aborted; the backend requests will occur concurrently and, when fulfilled, the response actions will be dispatched.

It's important to note that due to the concurrent handling of the actions, the order of the responses might not match the order of the requests. For example, if the user clicks the remove buttons of the first and second items, it's possible that the removal of the second item might occur before the removal of the first.

With our cart, the ordering of the removals doesn't matter, so using `mergeMap` instead of `switchMap` fixes the bug.

concatMap

The order in which items are removed from the cart might not matter, but there are often actions for which the ordering is important.

For example, if our shopping cart has a button for increasing an item's quantity, it's important that the dispatched actions are handled in the correct order. Otherwise, the quantities in the frontend's cart could end up out-of-sync with the quantities in the backend's cart.

With actions for which the ordering is important, `concatMap` should be used. `concatMap` is the equivalent of using `mergeMap` with a concurrency of one. That is, an effect/epic using `concatMap` will be processing only one backend request at a time—and actions are queued in the order in which they are dispatched.

`concatMap` is a safe, conservative choice. If you are unsure of which flattening operator to use in an effect/epic, use `concatMap`.

switchMap

The use of `switchMap` will see pending backend requests aborted whenever an action of the same type is dispatched. That makes `switchMap` unsafe for create, update and delete actions. However, it can also introduce bugs for read actions.

Whether or not `switchMap` is appropriate for a particular read action depends upon whether or not the backend response is still required after another action of the same type is dispatched. Let's look at an action with which the use of `switchMap` would introduce a bug.

If each item in our shopping cart has a details button—for showing some inline details—and the effect/epic that handles the details action uses `switchMap`, a race condition is introduced. If the user clicks the details buttons of several items, whether or not the details for those items will be displayed depends upon how rapidly the user clicks the buttons.

As with the `RemoveFromCart` action, using `mergeMap` would fix the bug.

`switchMap` should only be used in effects/epics for read actions and only when the backend response is not required after another action of the same type is dispatched.

Let's look at a scenario in which `switchMap` would be useful.

If our application's cart shows the total cost of the items plus the shipping, each change to the cart's content would see a `GetCartTotal`

action dispatched. Using `switchMap` for the effect/epic that handles the `GetCartTotal` action would be entirely appropriate.

If the cart is changed whilst the effect/epic is handling a `GetCartTotal` action, the response to the pending request will be stale—it'll be the total for the items in the cart prior to the change—so aborting the pending request is of no consequence. In fact, aborting is preferable to allowing the pending request to complete and then ignoring—or, worse, rendering—the stale response.

exhaustMap

`exhaustMap` is perhaps the least-well-known of the flattening operators, but it's easily explained: it can be thought of as the opposite of `switchMap`.

If `switchMap` is used, pending backend requests are aborted in favour of more recently dispatched actions. However, if `exhaustMap` is used, dispatched actions are ignored whilst there is a pending backend request.

Let's look at a scenario in which `exhaustMap` could be used.

There's a particular type of user with whom developers should be familiar: the incessant button clicker. When the incessant button clicker clicks a button and nothing happens, they click it again. And again. And again.

If our shopping cart has a refresh button and the effect/epic that handles the refresh uses `switchMap`, each incessant button click will abort the pending refresh. That doesn't make a whole lot of sense and the incessant button clicker could be clicking for a long, long time before a refresh occurs.

If the effect/epic that handles the refreshing of the cart instead used `exhaustMap`, a pending refresh request would see the incessant clicks ignored.

TL;DR

To summarise, when you need to use a flattening operator in an effect/epic you should:

- use `concatMap` with actions that should be neither aborted nor ignored and for which the ordering must be preserved—it's also

a conservative choice that will always behave in a predictable manner;

- use `mergeMap` with actions that should be neither aborted nor ignored and for which the ordering is unimportant;
- use `switchMap` with read actions that should be aborted when another action of the same type is dispatched; and
- use `exhaustMap` with actions that should be ignored whilst an action of the same type is pending.

Using TSLint to avoid switchMap misuse

I've added an `rxjs-no-unsafe-switchmap` rule to the `rxjs-tslint-rules` package.

The rule recognises NgRx effects and `redux-observable` epics, determines the their action types and then searches for particular verbs within the action types (e.g. `add`, `update`, `remove`, etc.). It has some sensible defaults and can be configured if the defaults are too general.

After enabling the rule and running TSLint over some applications I wrote last year, I found more than a few effects that used `switchMap` in an unsafe manner. So thanks for your tweet, Victor.