Deep Dive Into The RxJs switchMap Operator - How Does it Work ? A Less Well-Known Use Case (selector functions)

Although RxJs has a large number of operators, in practice we end up using a relatively small number of them.

And right after the most familiar operators that are also available in arrays (like map, filter, etc.), probably the first operator that we come across that is not part of the Array API but still very frequently used is the RxJs switchMap operator.

Let's see how this operator works by going over its behavior in a couple of very common situations:

- short-lived streams like for example HTTP Requests, that only emit one value
- long lived streams such as for example the ones returned by AngularFire, which is an Angular library that provides some services for interacting with the Firebase real-time database and authentication

In the end we will be covering a less well-known use case of the operator (emitting values that combine the output of multiple Observables using selector functions).

Simulating HTTP and Firebase RxJs Streams

To focus on the operator behavior, let's introduce here a couple of utility functions that will help us to simulate certain stream types that we usually come across in our day-to-day development:

- Cold, short-lived, emit-once streams like Angular HTTP streams
- Cold, long-lived, emit-many values streams like AngularFire streams

Simulating these 2 very common types of RxJs streams will help us to understand better their characteristics and behavior, besides the `switchMap` operator itself, so let's have a look at them.

How do Angular HTTP Observables work ?

Let's start by simulating the Observables that are returned by the Angular HTTP library, these are probably the first stream type that we will encounter while learning Angular.

These are just one particular type of Observables that have a certain number of properties, not necessarily shared with other types of Observables:

- the HTTP observables are cold (or not live), meaning that they will not start emitting values until we subscribe to them
- these Observables only emit a single value or an error, and then after that they complete, so they are not long-lived Observables

- in most cases we don't have to remember to unsubscribe from these Observables, because they will complete after emission

Simulating HTTP requests

With these features in mind, here is a function for creating an Observable with this behavior:

```
1
2  // Note: add this to enable the of and delay functionality
3  import 'rxjs/add/observable/of';
4  import 'rxjs/add/operator/delay';
5
6  function simulateHttp(val: any, delay:number) {
7      return Observable.of(val).delay(delay);
8  }
9
```

In this function we have used the `of` operator with a single value, to create a "emit once and complete" stream, and the `delay` operator to make it asynchronous, just like an HTTP request.

To make sure that this function is working correctly, let's try it out. Let's use it to simulate a couple of HTTP requests:

```
1
2  console.log('simulating HTTP requests');
3
4  const http1$ = simulateHttp("1", 1000);
5
6  const http2$ = simulateHttp("2", 1000);
7
```

These observables are cold, meaning that they will not emit values if they are not subscribed to. So let's subscribe to both Observables and

have a look at the output:

```
1
2  http1$.subscribe(
3      console.log,
4      console.error,
5      () => console.log('http1$ completed')
6  );
7
8  http2$.subscribe(
9      console.log,
10     console.error,
11     () => console.log('http2$ completed')
12 );
13
```

Quickly outputting observable values for test purposes

Notice that we are using an abbreviated way of printing the output to the console, for example the `http1$` subscription is equivalent to the following code:

```
1
2  http1$.subscribe(
3      val => console.log(val),
4      err => console.error(err),
5      () => console.log('http1$ completed')
6  );
7
```

In this example we are creating inline a function using the arrow operator, that takes the value and logs it. So that is why we can replace that with a direct reference to the `console.log` and `console.error` functions.

This shorthand notation comes in handy when for example debugging RxJs. With this in place, let's have a look at the output of these two subscriptions:

```
1
http1$ completed
2
http2$ completed
```

We can see that this observable has the usual behavior of an Angular HTTP Observable: each observable will emit only one value (it could also have emitted an error), and then the observable completes.

Introducing the Switch Map Operator

Let's then try the `switchMap` operator to combine two HTTP requests, and see the result.

Let's for example simulate a request that saves some user data, and then reloads some other data that is impacted by that server-side modification.

This is how we could simulate this scenario:

```
1
2  const saveUser$ = simulateHttp(" user saved ", 1000);
3
4  const httpResult$ = saveUser$.switchMap(sourceValue => {
5      console.log(sourceValue);
6      return simulateHttp(" data reloaded ", 2000);
7  });
8
9  httpResult$.subscribe(
```

```
10      console.log,
11      console.error,
12      () => console.log('completed httpResult$')
13  );
14
```

Let's have a look at what the output of this program would look like:

```
simulating HTTP requests
user saved
data reloaded
completed httpResult$
```

Given this output, let's break this down step-by-step to see what the switchMap operator is doing in this scenario:

- The `saveUser$` HTTP observable is said to be the *source* observable
- The output of the switchMap is the `resultObservable$` constant, which we will refer to as the result observable
- if we don't subscribe to the result observable, nothing will happen
- if we subscribe to the result observable, that will trigger a subscription to the source Observable `saveUser$`
- once the source observable emits, the source value emitted is then passed on to the function that we have passed to the `switchMap` operator
- that function needs to return an Observable, that might be built using the source value or not
- that returned observable is said to be the *inner* observable

- the inner observable is then subscribed to, and its output is then emitted also by the result observable
- when the source observable completes, the result observable also completes

So as we can seen the `switchMap` operator is a great way of doing one HTTP request using the output of an initial request here, so this is one common way that we can use it.

But at this point, we might have a couple of questions in mind, like for example:

Why is this operator called switchMap ?

Understanding why a functionality has a given name usually helps a lot in its understanding. So based on the name of the operator, we might ask the following questions:

- why the term `switch`, concretely what is being switched, both from and to?
- why the term `map`, what is being mapped in this example ?

At this point we might think the reason for the term `switch` is that we are switching from the source observable to the inner observable that is being created by the function passed on the `switchMap`, but as we will see that won't be the case.

The reason for the term `map` is more or less clear, what is being mapped is the emitted source value, that is getting mapped to an observable using the mapping function passed to `switchMap`.

So with this in mind let's now find out what is being switched.

Why we need a different type of stream to better understand switchMap

The problem with the type of short-lived HTTP-like streams that we have used so far, is that although we can already use `switchMap` for a lot of use cases, its not 100% clear at this point how the operator works.

This is because the stream only emits once and then they complete, so its a very particular case. To better understand the `switchMap` operator, we need to introduce a different type of streams: long-lived AngularFire-like streams.

Simulating AngularFire streams

Let's then introduce a utility function for simulating this new type of long-lived streams:

```
1
2  // Note: add this to enable the interval and map functionality
3  import 'rxjs/add/observable/interval';
4  import 'rxjs/add/operator/map';
5
6  function simulateFirebase(val: any, delay: number) {
7      return Observable.interval(delay).map(index => val + " " + index);
8  }
9
```

Using this utility function, let's have a look at this small program that simulates two Firebase-like streams:

```
1
2  const firebase1$ = simulateFirebase("FB-1 ", 5000);
3  const firebase2$ = simulateFirebase("FB-2 ", 1000);
```

```
4
5  firebase1$.subscribe(
6      console.log,
7      console.error,
8      () => console.log('firebase1$ completed')
9  );
10
11 firebase2$.subscribe(
12     console.log,
13     console.error,
14     () => console.log('firebase2$ completed')
15 );
16
```

How do the Firebase long-lived streams work ?

Let's understand how this type of streams work, by looking at the output of the small program that we just wrote:

```
1:  FB-2  0
2:  FB-2  1
3:  FB-2  2
4:  FB-2  3
5:  FB-1  0
6:  FB-2  4
7:  FB-2  5
8:  FB-2  6
9:  FB-2  7
10: FB-2  8
11: FB-1  1
12: FB-2  9
...
```

Long-lived streams

This type of streams are quite different from HTTP streams. To understand how they work, let's break this down line by line, starting at the last one:

- maybe the most noticeable thing here is that both streams never complete, they keep emitting values !
- this type of streams will continue to emit new values (if a new value is available) until we unsubscribe from them
- The `firebase2$` observable emitted a few values, because it has an interval of only 1 second
- The `firebase1$` stream emitted less values, because it has an interval of 5 seconds

Understanding the switchMap Operator

So with this in place, let's take these two long-lived streams and switch map one into the other, and see what happens - this new scenario will make more apparent what the operator is doing under the hood.

For example, let's try to guess the output of this small program:

```
1
2  const firebase1$ = simulateFirebase("FB-1 ", 5000);
3  const firebase2$ = simulateFirebase("FB-2 ", 1000);
4
5  const firebaseResult$ = firebase1$.switchMap(sourceValue => {
6    console.log("source value " + sourceValue);
7    return simulateFirebase("inner observable ", 1000)
8  });
9
10 firebaseResult$.subscribe(
11   console.log,
12   console.error,
13   () => console.log('completed firebaseResult$')
```

```
14  );
15
```

Here is the program output, with some indentation added to it:

```
source value FB-1  0
    inner observable  0
    inner observable  1
    inner observable  2
    inner observable  3
source value FB-1  1
    inner observable  0
    inner observable  1
    inner observable  2
    inner observable  3
source value FB-1  2
    inner observable  0
    inner observable  1
    inner observable  2
...
```

The reason for the term Switch

Let's now learn why the term `switch` is used to name the `switchMap` operator, by breaking down this output step-by-step:

- just like in the case of the HTTP example, nothing will happen until the result observable is subscribed to
- the source observable will then emit the first value `FB-1  0`
- the source value will be mapped into an inner observable, which is the output of the mapping function passed to

`switchMap`

- that inner observable will be subscribed to
- the result observable will emit the values also emitted by the inner observable
- we can see that the inner observable is emitting the values 0, 1, 2 that are the output of newly created interval

Why the term Switch ?

But then something different than the HTTP example happens: because the source observable is long-lived, it will eventually emit a new value, in this case `FB-1  1`.

And from that new value onwards, the following happens:

- it looks like the long running inner observable that was already at index 3 is no longer being used
- the mapping function is called again, and a new inner observable is created
- the new inner observable is now subscribed to
- the newly created inner observable will start emitting values again from index 0
- the result observable now is emitting the values emitted by the newly created inner observable, instead of the previously running inner observable

So what happened to the previously running inner observable?

That previous inner observable has been unsubscribed from, and so those values are no longer being used.

> *The result observable has `switched` from emitting the values of the first inner observable, to emitting the values of the newly created inner observable*

And this explains the use of the term `switch` in the name `switchMap` !

The switchMap Operator in a Nutshell

Let's then summarize what we have learned so far: The `switchMap` operator will create a derived observable (called inner observable) from a source observable and emit those values.

When the source emits a new value, it will create a new inner observable and `switch` to those values instead. What gets unsubscribed from are the inner observables that get created on the fly, and not the source observable.

I hope this helps explain the operator, its name and some frequently used use cases for which you might want to use it (let me know in the comments).

> *But there is also less well known use case of this operator, that might come in very handy*

So let's then cover the RxJs `switchMap` selector functions.

What are selector functions, in the context of the switchMap operator ?

It turns out that the mapping function passed to `switchMap` is not the only argument that we can pass to the operator. We can also pass a second argument which is the selector function.

Let's see a common use case for this function: so far we have been emitting the value if the inner observable directly as the output of the result observable.

But what if we would like to emit a different output that instead combines:

- the output of the first Observable
- and the output of the inner Observable ?

An example of of an use case for selector functions

Let's have a look at a common use case: let's say that we have an initial HTTP request that loads a course object. We would like to take the course id and then load the course lessons via a second HTTP request.

To implement this use case using `switchMap`, we could do something like this:

```
1
2  const course$ = simulateHttp({id:1, description: 'Angular For Beginners'},
3
4  const httpResult$ = course$.switchMap(
5    sourceValue => simulateHttp([... returns a lessons array ...], 2000));
6
7  httpResult$.subscribe(
8      console.log,
9      console.error,
10     () => console.log('completed httpResult$')
11 );
12
```

And with this code, we would emit as value of the result observable the list of lessons. But what if we wanted to emit both the course *and* the

list of lessons ?

That is when the second argument of `switchMap` can be used. We can pass in as the second argument another function to `switchMap` that will allow us to combine the multiple values of the inner and the source observable.

Using switchMap selector functions

This is how we could use a selector function to emit both the course and its lessons:

```
1
2  const course$ = simulateHttp({id:1, description: 'Angular For Beginners'},
3
4  const httpResult$ = course$.switchMap(
5      courses => simulateHttp([], 2000),
6      (courses, lessons, outerIndex, innerIndex) => [courses, lessons] );
7
8  httpResult$.subscribe(
9      console.log,
10     console.error,
11     () => console.log('completed httpResult$')
12 );
13
```

> *Reminder: an example similar to this one can be viewed in this video*

Breakdown of the selector functions arguments

As we can see, the selector function takes 4 values:

- the first argument is the value of the source observable, which in our case is a course object returned by the initial HTTP request

- the second argument is the value of the inner observable, which in our case is an array with a list of lessons
- the third argument is the source observable index. this will be zero for the first source value emitted, 1, 2, 3, etc.
- the fourth and last argument is the inner index. This will start at zero for the first value of the first inner observable created, then 1, 2, etc. until the inner observable is unsubscribed from. From there it will start again at zero for the second inner observable

Notice that the output of the selector function is a Typescript Tuple type, meaning its essentially a strongly typed Array where we can specify what type we have on the first argument, the second argument, etc..

Inspecting the output of the selector function

Let's then see the end result on the console. As we can see, the value emitted by the result observable using the new selector function would be:

```
[Object, Array(30)]
```

At runtime the result observable is emitting values that are plain Javascript arrays. And looking inside these arrays we have:

- the first element of the array is the course object, which is the output of the initial source observable HTTP call
- the second element of the array is an array of lessons, which is the value emitted by the inner observable

Conclusion

The RxJs `switchMap` its a very useful and commonly used operator that is important to have a good grasp on, because we will be using it to implement many use cases.

This is one of those few operators that we will probably use in just about any application that we build.