

# Comprehensive Guide to Higher-Order RxJs Mapping Operators: `switchMap`, `mergeMap`, `concatMap` (and `exhaustMap`)

Some of the *most commonly used RxJs operators* that we find on a daily basis are the RxJs higher-order mapping operators: `switchMap`, `mergeMap`, `concatMap` and `exhaustMap`.

For example, **most of the network calls in our program** are going to be done using one of these operators, so getting familiar with them is essential in order to write almost any reactive program.

Knowing which operator to use in a given situation (and why) *can be a bit confusing*, and we often wonder how do these operators really work and why they are named like that.

These operators might seem unrelated, but we really want to learn them all in one go, as choosing the wrong operator might accidentally lead to subtle issues in our programs.

## Why are the mapping operators a bit confusing?

There is a reason for that: in order to understand these operators, we need to first understand the Observable combination strategy that each one uses internally.

Instead of trying to understand `switchMap` on its own, we need to first understand what is Observable switching; instead of diving straight into `concatMap`, we need to first learn Observable concatenation, etc.

So that is what we will be doing in this post, we are going to learn in a logical order the `concat`, `merge`, `switch` and `exhaust` strategies and their corresponding mapping operators: `concatMap`, `mergeMap`, `switchMap` and `exhaustMap`.

We will explain the concepts using a combination of marble diagrams and some practical examples (including running code).

In the end, you will know exactly how each of these mapping operators work, when to use each and why, and the reason for their names.

## Table of Contents

In this post, we will cover the following topics:

- The RxJs Map Operator
- What is higher-order Observable Mapping
- Observable Concatenation
- The RxJs `concatMap` Operator
- Observable Merging
- The RxJs `mergeMap` Operator
- Observable Switching

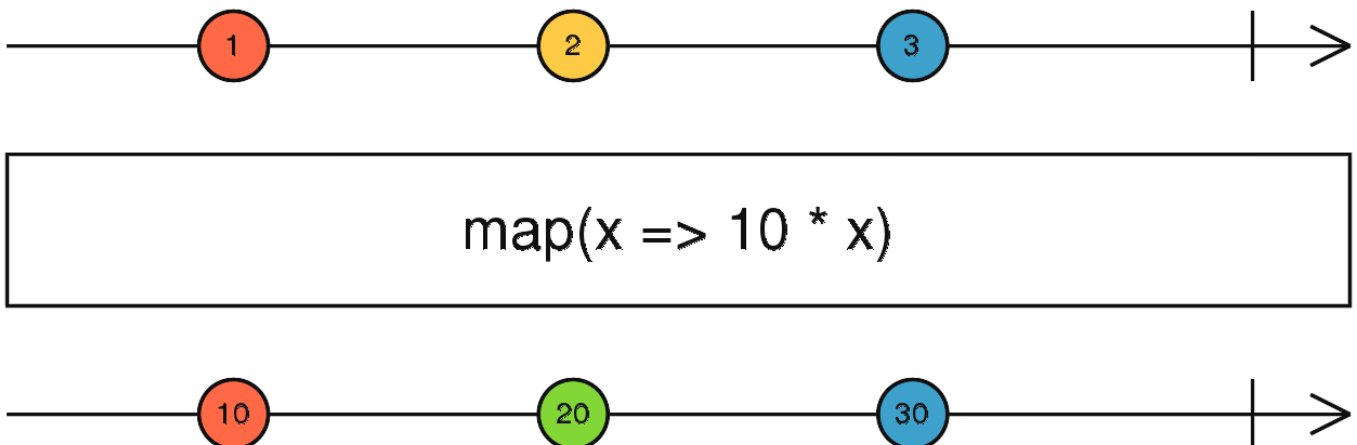
- The RxJs switchMap Operator
- The Exhaust strategy
- The RxJs exhaustMap Operator
- How to choose the right mapping Operator?
- Running GitHub repo (with code samples)
- Conclusions

Note that this post is part of our ongoing [RxJs Series](#). So without further ado, let's get started with our RxJs mapping operators deep dive!

## The RxJs Map Operator

Let's start at the beginning, by covering what these mapping operators are doing in general.

As the names of the operators imply, they are doing some sort of mapping: but *what* is exactly getting mapped? Let's have a look at the marble diagram of the RxJs Map operator first:



## How the base Map Operator works

With the map operator, we can take an input stream (with values 1, 2, 3), and from it, we can create a derived mapped output stream (with values 10, 20, 30).

The values of the output stream in the bottom are obtained by taking the values of the input stream and applying them a function: this function simply multiplies the values by 10.

So the map operator is all about mapping the *values* of the input observable. Here is an example of how we would use it to handle an HTTP request:

```
1
2  const http$ : Observable<Course[]> = this.http.get('/api/courses');
3
4  http$
5    .pipe(
6      tap(() => console.log('HTTP request executed')),
7      map(res => Object.values(res['payload']))
8    )
9    .subscribe(
10      courses => console.log("courses", courses)
11    );
12
```

In this example, we are creating one HTTP observable that makes a backend call and we are subscribing to it. The observable is going to emit the value of the backend HTTP response, which is a JSON

object.

In this case, the HTTP response is wrapping the data in a payload property, so in order to get to the data, we apply the RxJs map operator. The mapping function will then map the JSON response payload and extract the value of the payload property.

Now that we have reviewed how base mapping works, let's now talk about higher-order mapping.

## What is Higher-Order Observable Mapping?

In higher-order mapping, instead of mapping a plain value like 1 to another value like 10, we are going to map a value into an Observable!

The result is a higher-order Observable. It's just an Observable like any other, but its values are themselves Observables as well, that we can subscribe to separately.

This might sound far-fetched, but in reality, this type of mapping happens all the time. Let's give a practical example of this type of mapping. Let's say that for example, we have an Angular Reactive Form that is emitting valid form values over time via an Observable:

```
1
2 @Component({
3     selector: 'course-dialog',
```

```

4      templateUrl: './course-dialog.component.html'
5    })
6    export class CourseDialogComponent implements AfterViewInit {
7
8      form: FormGroup;
9      course: Course;
10
11      @ViewChild('saveButton') saveButton: ElementRef;
12
13      constructor(
14          private fb: FormBuilder,
15          private dialogRef: MatDialogRef<CourseDialogComponent>,
16          @Inject(MAT_DIALOG_DATA) course: Course) {
17
18          this.course = course;
19
20          this.form = fb.group({
21              description: [course.description, Validators.required],
22              category: [course.category, Validators.required],
23              releasedAt: [moment(), Validators.required],
24              longDescription: [course.longDescription, Validators.required],
25          });
26      }
27  }
28
29

```

The Reactive Form provides an Observable

`this.form.valueChanges` that emits the latest form values as the user interacts with the form. This is going to be our source Observable.

What we want to do is to save at least some of these values as they get emitted over time, to implement a form draft pre-save feature. This way the data gets progressively saved as the user fills in the form, which avoids losing the whole form data due to an

accidental reload.

## Why Higher-Order Observables?

In order to implement the form draft save functionality, we need to take the form value, and then create a second HTTP observable that performs a backend save, and then subscribe to it.

We could try to do all of this manually, but then we would fall in the *nested subscribes anti-pattern*:

```
1
2  this.form.valueChanges
3    .subscribe(
4      formValue => {
5
6        const httpPost$ = this.http.put(`/api/course/${courseId}`,
7
8        httpPost$.subscribe(
9          res => ... handle successful save ...
10         err => ... handle save error ...
11       );
12
13     }
14   );
15
```

As we can see, this would cause our code to nest at multiple levels quite quickly, which was one of the problems that we were trying to avoid while using RxJs in the first place.

Let's call this new `httpPost$` Observable the inner Observable, as it was created in an inner nested code block.

## Avoiding nested subscriptions

We would like to do all this process in a much more convenient way: we would like to take the form value, and *map* it into a save Observable. And this would effectively create a higher-order Observable, where each value corresponds to a save request.

We want to then transparently subscribe to each of these network Observables, and directly receive the network response all in one go, to avoid any nesting.

And we could do all this if we would have available some sort of a higher order RxJs mapping operator! Why do we need four different operators then?

To understand that, imagine what happens if multiple form values are emitted by the `valueChanges` observable in quick succession and the save operation takes some time to complete:

- should we wait for one save request to complete before doing another save?
- should we do multiple saves in parallel?
- should we cancel an ongoing save and start a new one?
- should we ignore new save attempts while one is already ongoing?

Before exploring each one of these use cases, let's go back to the nested subscribes code above.



In the nested subscribes example, we are actually triggering the save operations in parallel, which is not what we want because there is no strong guarantee that the backend will handle the saves sequentially and that the last valid form value is indeed the one stored on the backend.

Let's see what it would take to ensure that a save request is done only after the previous save is completed.

## Understanding Observable Concatenation

In order to implement sequential saves, we are going to introduce the new notion of Observable concatenation. In this code example, we are concatenating two example observables using the

`concat()` RxJs function:

```
1
2  const series1$ = of('a', 'b');
3
4  const series2$ = of('x', 'y');
5
6  const result$ = concat(series1$, series2$);
7
8  result$.subscribe(console.log);
9
```

After creating two Observables `series1$` and `series2$` using the `of` creation function, we have then created a third `result$` Observable, which is the result of concatenating `series1$` and `series2$`.

Here is the console output of this program, showing the values emitted by the result Observable:

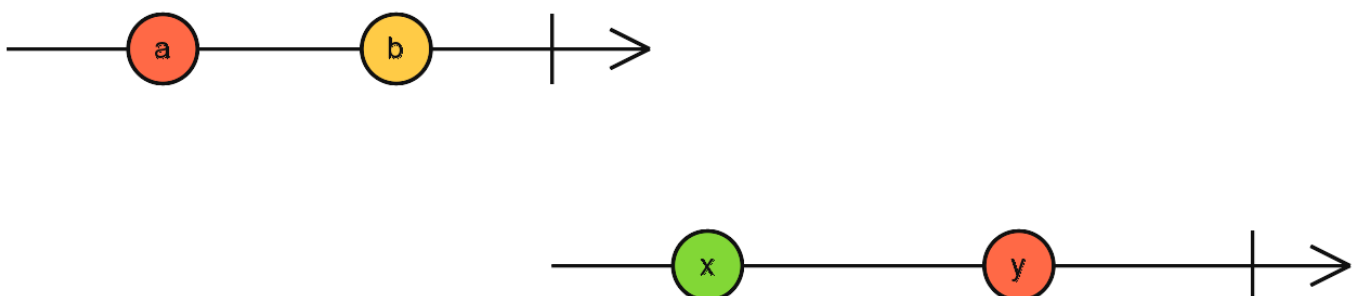
```
a  
b  
x  
y
```

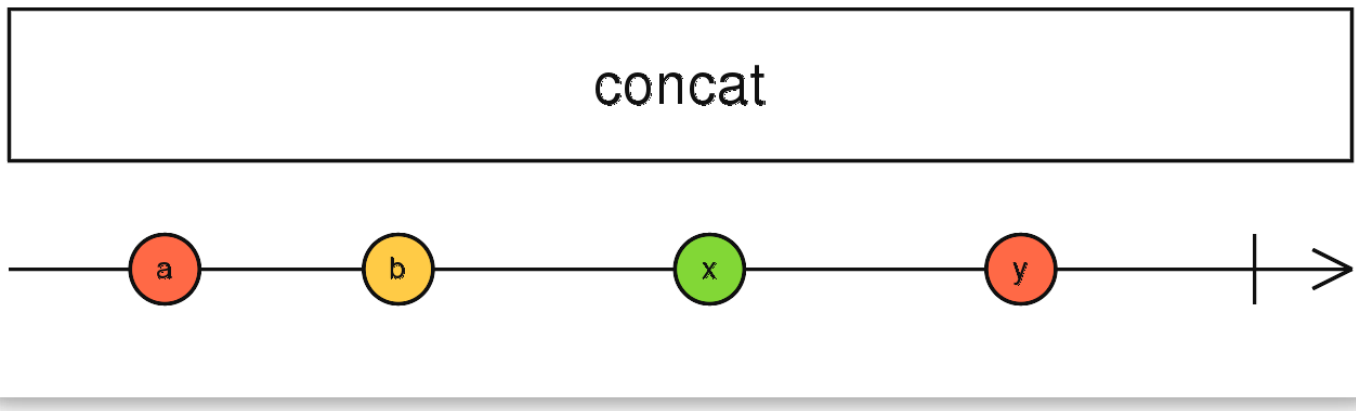
As we can see, the values are the result of concatenating the values of `series1$` with `series2$` together. But here is the catch: this only works because these Observables are completing!!

The `of()` function will create Observables that emit values passed to `of()` and then it will complete the Observables after all values are emitted.

## Observable Concatenation Marble Diagram

To really understand what is going on, we need to look at the Observable concatenation marble diagram:





Do you notice the vertical bar after the value `b` on the first Observable? That marks the point in time when the first Observable with values `a` and `b` (`series1$`) is completed.

Let's break down what is going on here by following step-by-step the timeline:

- the two Observables `series1$` and `series2$` are passed to the `concat()` function
- `concat()` will then subscribe to the first Observable `series1$`, but *not* to the second Observable `series2$` (this is critical to understand concatenation)
- `source1$` emits the value `a`, which gets immediately reflected in the output `result$` Observable
- note that the `source2$` Observable is not yet emitting values, because it has not yet been subscribed to
- `source1$` will then emit the `b` value, which gets reflected in the output
- `source1$` will then complete, and only after that will

`concat()` now subscribe to `source2$`

- the `source2$` values will then start getting reflected in the output, until `source2$` completes
- when `source2$` completes, then the `result$` Observable will also complete
- note that we can pass to `concat()` as many Observables as we want, and not only two like in this example

## The key point about Observable Concatenation

As we can see, Observable concatenation is all about Observable completion! We take the first Observable and use its values, wait for it to complete and then we use the next Observable, etc. until all Observables complete.

Going back to our higher-order Observable mapping example, let's see how the notion of concatenation can help us.

## Using Observable Concatenation to implement sequential saves

As we have seen, in order to make sure that our form values are saved sequentially, we need to take each form value and map it to an `httpPost$` Observable.

We then need to subscribe to it, but we want the save to complete before subscribing to the next `httpPost$` Observable.

*In order to ensure sequentiality, we need to concatenate the multiple*

`httpPost$` *Observables together!*

We will then subscribe to each `httpPost$` and handle the result of each request sequentially. In the end, what we need is an operator that does a mixture of:

- a higher-order *mapping* operation (taking the form value and turning it into an `httpPost$` Observable)
- with a `concat()` operation, concatenating the multiple `httpPost$` Observables together to ensure that an HTTP save is not made before the previous ongoing save completes first

What we need is the aptly named *RxJs concatMap Operator*, which does this mixture of higher order mapping with Observable concatenation.

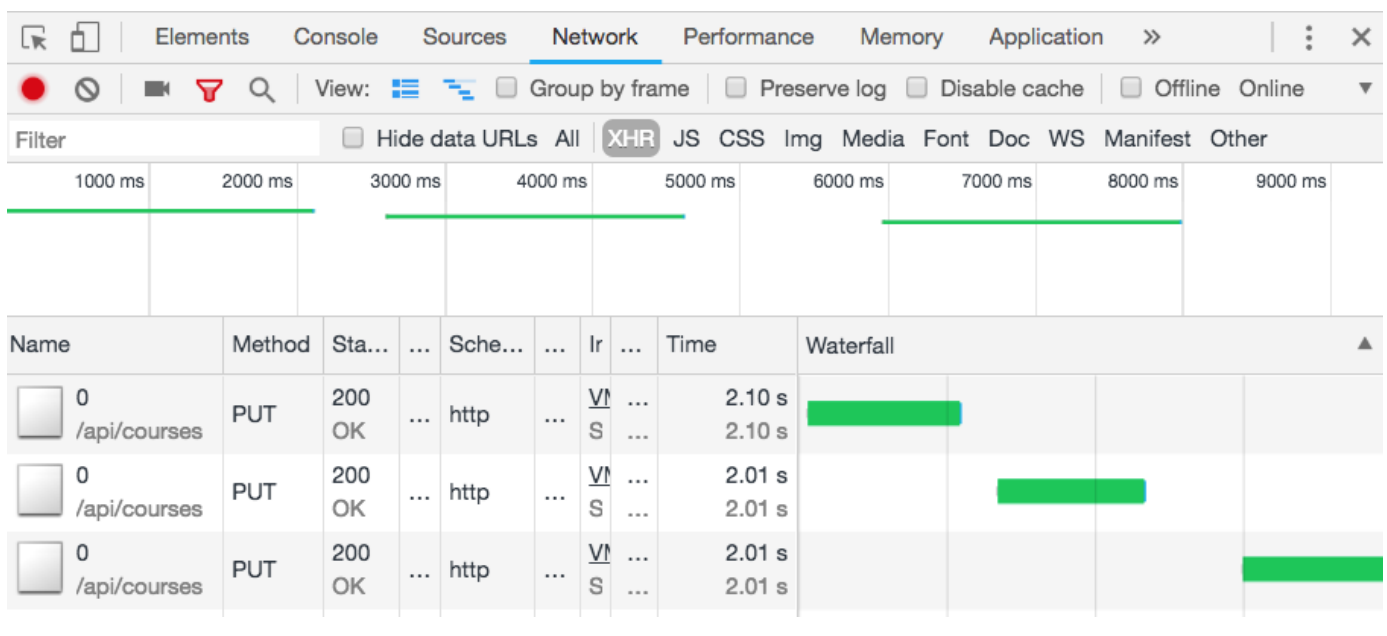
## The RxJs concatMap Operator

Here is what our code looks like if we now use the concatMap Operator:

```
1
2  this.form.valueChanges
3    .pipe(
4      concatMap(formValue => this.http.put(`/api/course/${courseId}`
5    )
6    .subscribe(
7      saveResult => ... handle successful save ...,
8      err => ... handle save error ...
9    );
```

As we can see, the first benefit of using a higher-order mapping operator like `concatMap` is that now we no longer have nested subscribes.

By using `concatMap`, now all form values are going to be sent to the backend sequentially, as shown here in the Chrome DevTools Network tab:



## Breaking down the `concatMap` network log diagram

As we can see, one save HTTP request starts only after the previous save has completed. Here is how the `concatMap` operator is ensuring that the requests always happen in sequence:

- `concatMap` is taking each form value and transforming it into a save HTTP Observable, called an *inner Observable*

- concatMap then subscribes to the inner Observable and sends its output to the result Observable
- a second form value might come faster than what it takes to save the previous form value in the backend
- If that happens, that new form value will **not** be immediately mapped to an HTTP request
- instead, concatMap will wait for previous HTTP Observable to complete *before* mapping the new value to an HTTP Observable, subscribing to it and therefore triggering the next save

Notice that the code here is just the basis of an implementation to save draft form values. You can combine this with other operators to for example save only valid form values, and throttle the saves to make sure that they don't occur too frequently.

## Observable Merging

Applying Observable concatenation to a series of HTTP save operations seems like a good way to ensure that the saves happen in the intended order.

But there are other situations where we would like to instead run things in parallel, without waiting for the previous inner Observable to complete.

And for that, we have the merge Observable combination strategy! Merge, unlike concat, will not wait for an Observable to complete before subscribing to the next Observable.

Instead, merge subscribes to every merged Observable at the same time, and then it outputs the values of each source Observable to the combined result Observable as the multiple values arrive over time.

## Practical Merge Example

To make it clear that merging does not rely on completion, let's merge two Observables that *never* complete, as these are interval Observables:

```
1
2  const series1$ = interval(1000).pipe(map(val => val*10));
3
4  const series2$ = interval(1000).pipe(map(val => val*100));
5
6  const result$ = merge(series1$, series2$);
7
8  result$.subscribe(console.log);
9
```

The Observables created with `interval()` will emit the values 0, 1, 2, etc. at a one second interval and will never complete.

Notice that we are applying a couple of map operator to these interval Observables, just to make it easier to distinguish them in the console output.

Here are the first few values visible in the console:

```
0
```



```
0
10
100
20
200
30
300
```

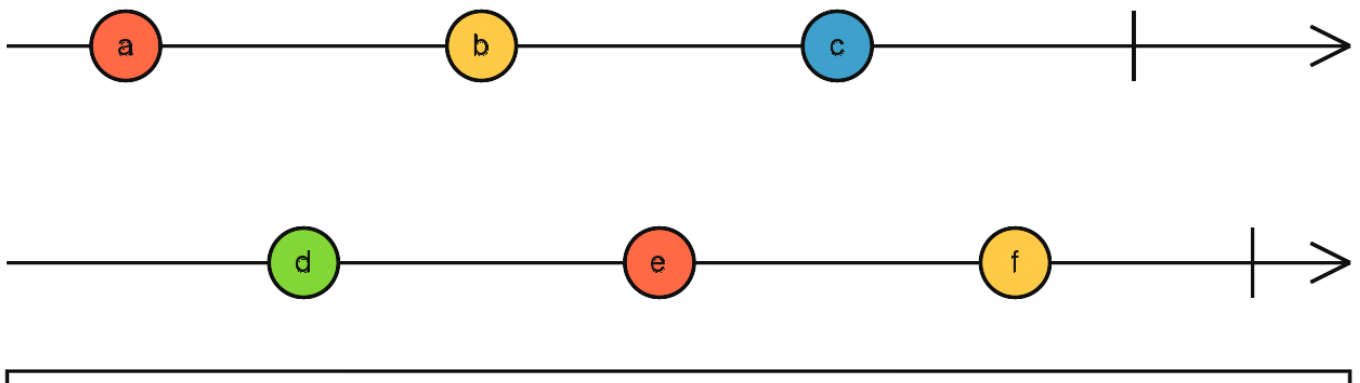
## Merging and Observable Completion

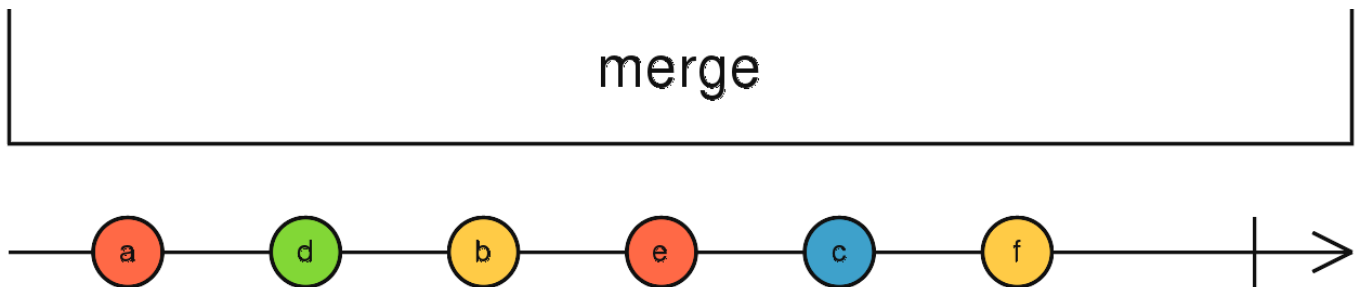
As we can see, the values of the merged source Observables show up in the result Observable immediately as they are emitted. If one of the merged Observables completes, merge will continue to emit the values of the other Observables as they arrive over time.

Notice that if the source Observables do complete, merge will still work in the same way.

## The Merge Marble Diagram

Let's take another merge example, depicted in the following marble diagram:



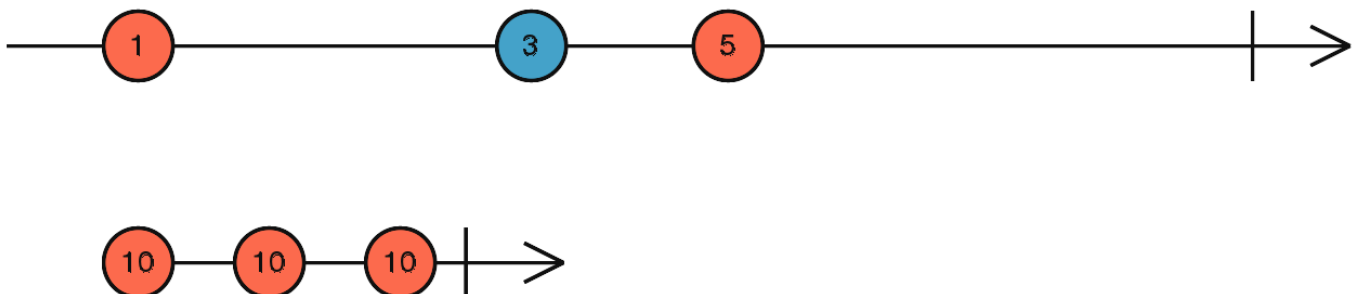


As we can see, the values of the merged source Observables show up immediately in the output. The result Observable will not be completed until *all* the merged Observables are completed.

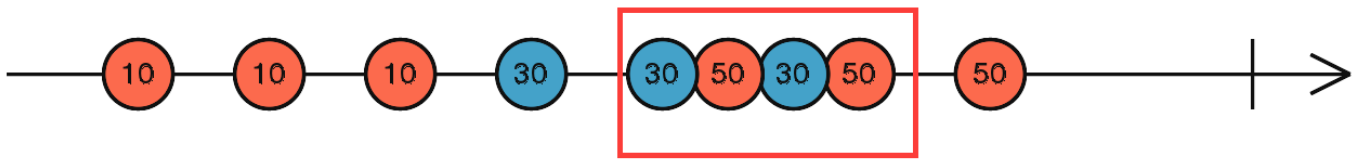
Now that we understand the merge strategy, let's see how it can be used in the context of higher-order Observable mapping.

## The RxJs mergeMap Operator

If we combine the merge strategy with the notion of higher-order Observable mapping, we get the RxJs mergeMap Operator. Let's have a look at the marble diagram for this operator:



```
mergeMap(i => 10*i——10*i——10*i—| )
```



Here is how the mergeMap operator works:

- each value of the source Observable is still being mapped into an inner Observable, just like the case of concatMap
- Like concatMap, that inner Observable is also subscribed to by mergeMap
- as the inner Observables emit new values, they are immediately reflected in the output Observable
- but unlike concatMap, in the case of mergeMap we don't have to wait for the previous inner Observable to complete before triggering the next inner Observable
- this means that with mergeMap (unlike concatMap) we can have multiple inner Observables overlapping over time, emitting values in parallel like we see highlighted in red in the picture

## Checking the mergeMap Network Log

Going back to our previous form draft save example, its clear that what we need concatMap in that case and **not** mergeMap, because we *don't* want the saves to happen in parallel.

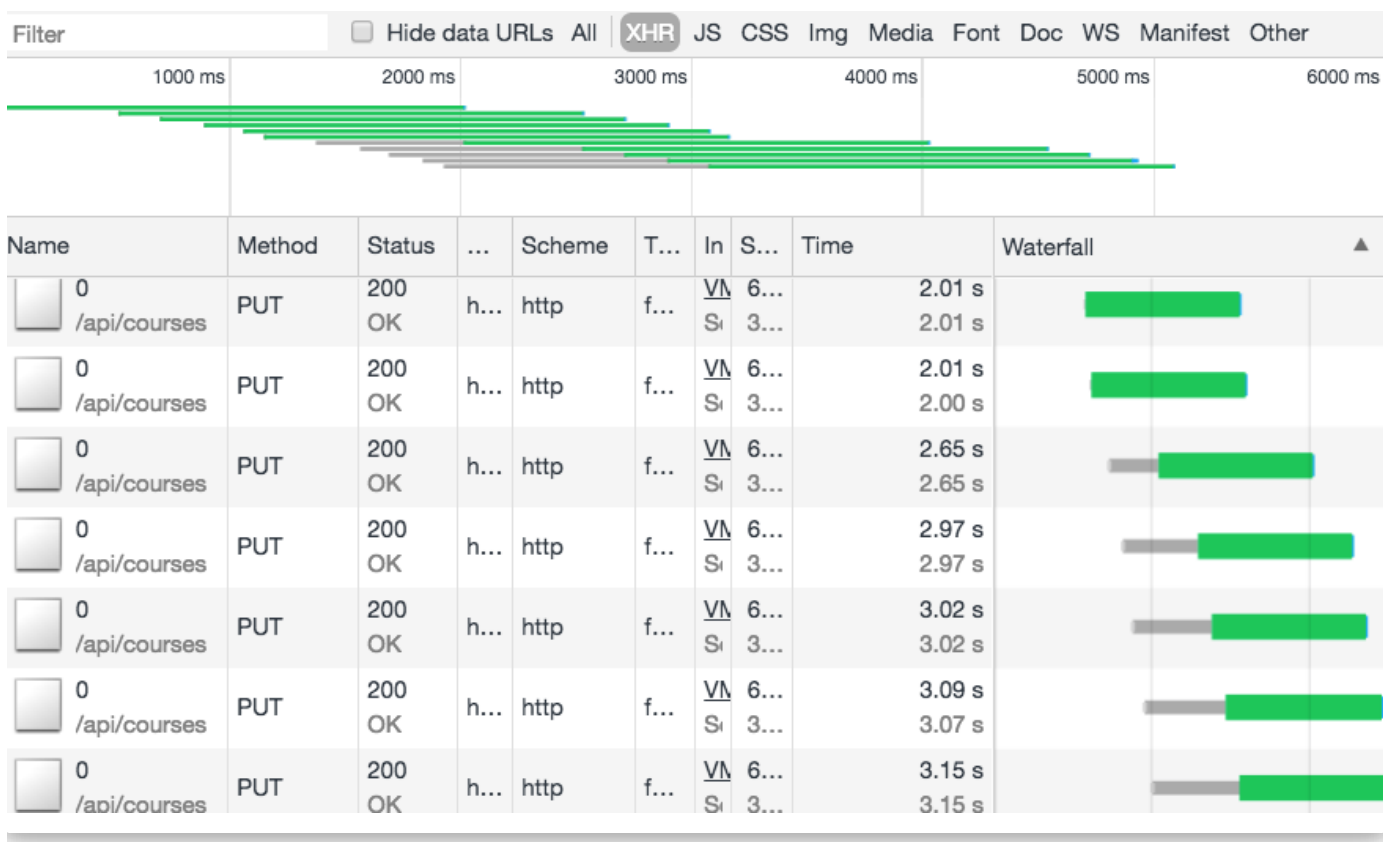
Let's see what happens if we would accidentally choose mergeMap instead:

```

1
2  this.form.valueChanges
3    .pipe(
4      mergeMap(formValue => this.http.put(`/api/course/${courseId}`
5    )
6    .subscribe(
7      saveResult => ... handle successful save ...,
8      err => ... handle save error ...
9    );
10

```

Let's now say that the user interacts with the form and starts inputting data rather quickly. In that case, we would now see multiple save requests running in parallel in the network log:



As we can see, the requests are happening in parallel, which in this case is an error! Under heavy load, it's possible that these requests

would be processed out of order.

## Observable Switching

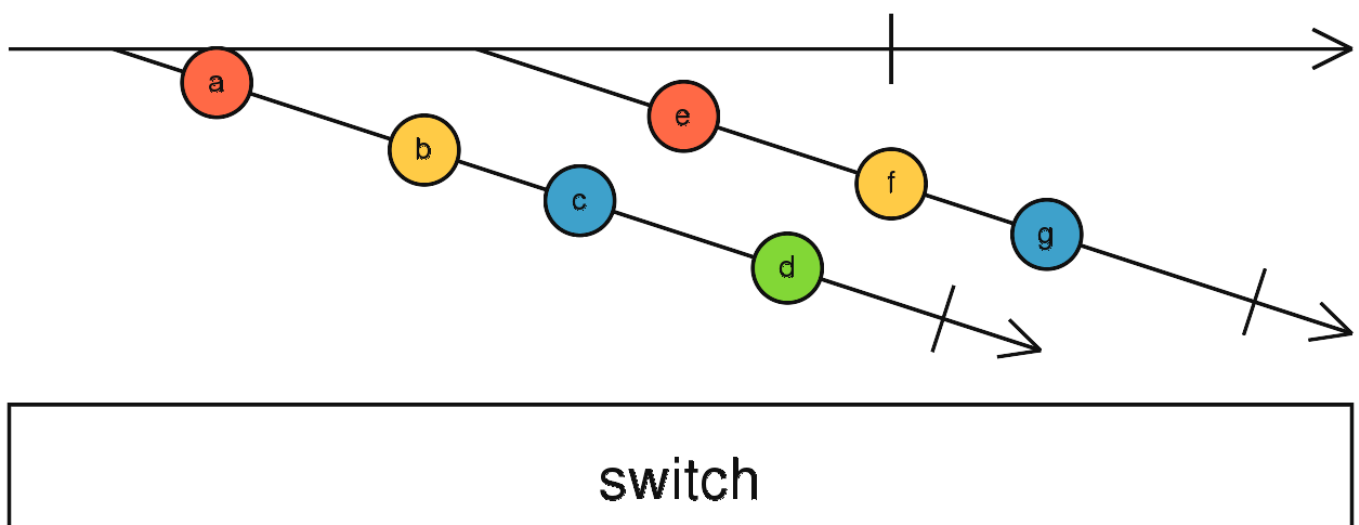
Let's now talk about another Observable combination strategy: switching. The notion of switching is closer to merging than to concatenation, in the sense that we don't wait for any Observable to terminate.

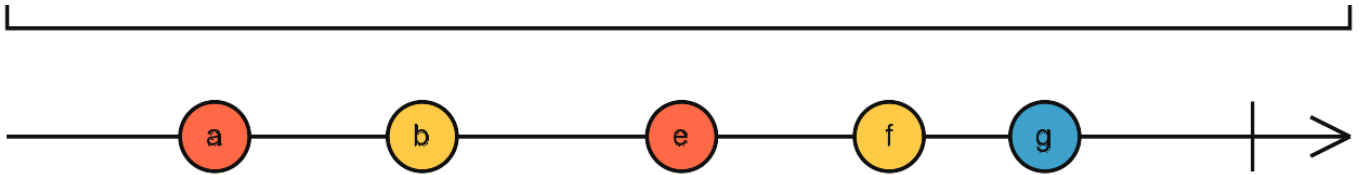
But in switching, unlike merging, if a new Observable starts emitting values we are then going to unsubscribe from the previous Observable, before subscribing to the new Observable.

*Observable switching is all about ensuring that the unsubscription logic of unused Observables gets triggered, so that resources can be released!*

### Switch Marble Diagram

Let's have a look at the marble diagram for switching:





Notice the diagonal lines, these are not accidental! In the case of the switch strategy, it was important to represent the higher-order Observable in the diagram, which is the top line of the image.

This higher-order Observable emits values which are themselves Observables.

*The moment that a diagonal line forks from the higher-order Observable top line, is the moment when a value Observable was emitted and subscribed to by switch.*

## Breaking down the switch Marble Diagram

Here is what is going on in this diagram:

- the higher-order Observable emits its first inner Observable (a-b-c-d), that gets subscribed to (by the switch strategy implementation)
- the first inner Observable (a-b-c-d) emits values a and b, that get immediately reflected in the output
- but then the second inner Observable (e-f-g) gets emitted, which *triggers the unsubscription* from the first inner Observable (a-b-c-d), and this is the key part of switching

- the second inner Observable (e-f-g) then starts emitting new values, that get reflected in the output
- but notice that the first inner Observable (a-b-c-d) is meanwhile *still* emitting the new values c and d
- these later values, however, are **not** reflected in the output, and that is because we had meanwhile *unsubscribed* from the first inner Observable (a-b-c-d)

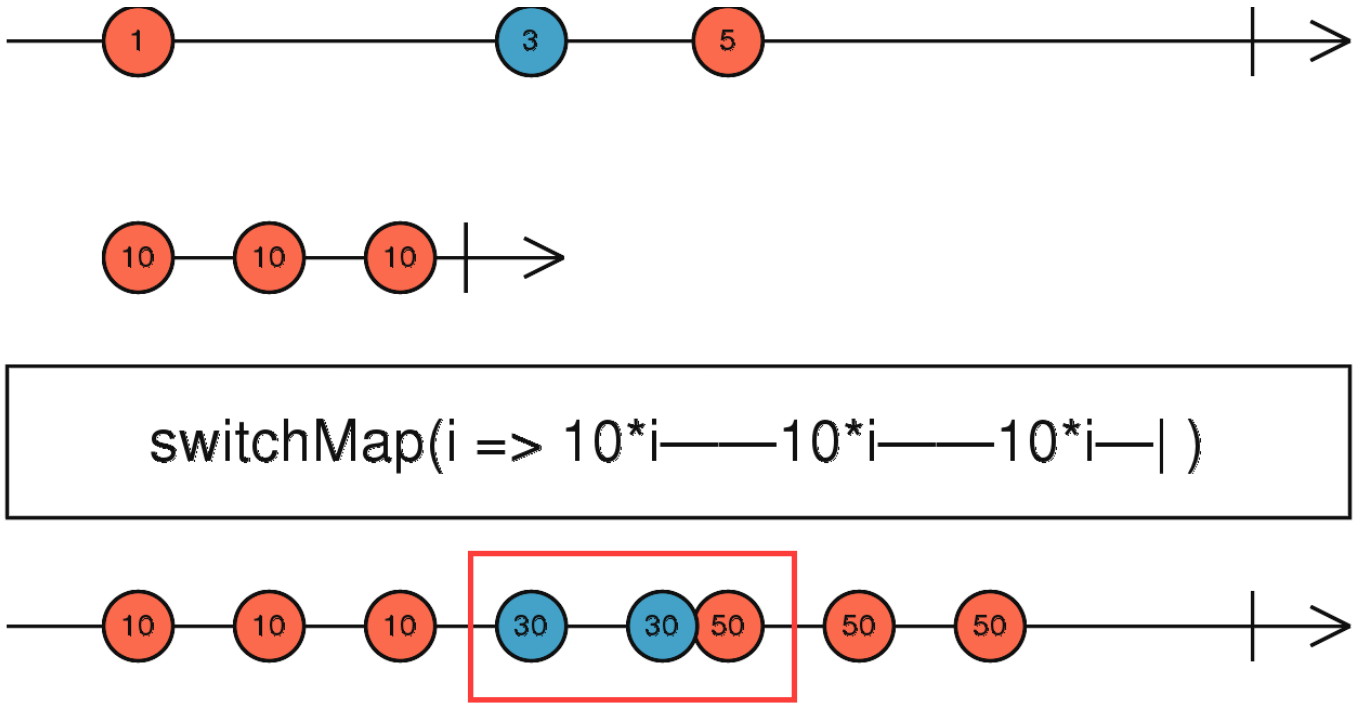
We can now understand why the diagram had to be drawn in this unusual way, with diagonal lines: its because we need to represent visually when each inner Observable gets subscribed (or unsubscribed) from, which happens at the points the diagonal lines fork from the source higher-order Observable.

## The RxJs switchMap Operator

Let's then take the switch strategy and apply it to higher order mapping. Let's say that we have a plain input stream that is emitting the values 1, 3 and 5.

We are then going to map each value to an Observable, just like we did in the cases of concatMap and mergeMap and obtain a higher-order Observable.

If we now switch between the emitted inner Observables, instead of concatenating them or merging them, we end up with the switchMap Operator.



## Breaking down the switchMap Marble Diagram

Here is how this operator works:

- the source observable emits values 1, 3 and 5
- these values are then turned into Observables by applying a mapping function
- the mapped inner Observables get subscribed to by `switchMap`
- when the inner Observables emit a value, the value gets immediately reflected in the output
- but if a new value like 5 gets emitted *before* the previous Observable got a chance to complete, the previous inner Observable (30-30-30) will be unsubscribed from, and its values will no longer be reflected in the output
- notice the 30-30-30 inner Observable in red in the



diagram above: the last 30 value was not emitted because the 30-30-30 inner Observable got unsubscribed from

So as we can see, Observable switching is all about making sure that we trigger that unsubscription logic from unused Observables. Let's now see switchMap in action!

## Search TypeAhead - switchMap Operator Example

A very common use case for switchMap is a search Typeahead. First let's define our source Observable, whose values are themselves going to trigger search requests.

This source Observable is going to emit values which are the search text that the user types in an input:

```
1
2  const searchText$: Observable<string> = fromEvent<any>(this.input.nati
3      .pipe(
4          map(event => event.target.value),
5          startWith('')
6      )
7      .subscribe(console.log);
8
9
```

This source Observable is linked to an input text field where the user types its search. As the user types the words "Hello World" as a search, these are the values emitted by `searchText$`:

```
H
H
He
Hel
Hell
Hello
Hello
Hello W
Hello W
Hello Wo
Hello Wor
Hello Worl
Hello World
```

## Debouncing and removing duplicates from a Typeahead

Notice the duplicate values, either caused by the use of the space between the two words, or the use of the Shift key for capitalizing the letters H and W.

In order to avoid sending all these values as separate search requests to the backend, let's wait for the user input to stabilize by using the `debounceTime` operator:

```
1
2  const searchText$: Observable<string> = fromEvent<any>(this.input.nat
3      .pipe(
4          map(event => event.target.value),
5          startWith(''),
```

```
6         debounceTime(400)
7     )
8     .subscribe(console.log);
9
10
```

With the use of this operator, if the user types at a normal speed, we now have only one value in the output of `searchText$`:

```
Hello World
```

This is already much better than what we had before, now a value will only be emitted if its stable for at least 400ms!

But if the user types slowly as he is thinking about the search, to the point that it takes more than 400 ms between two values, then the search stream could look like this:

```
He
Hell
Hello World
```

Also, the user could type a value, hit backspace and type it again, which might lead to duplicate search values. We can prevent the occurrence of duplicate searches by adding the `distinctUntilChanged` operator.

## Cancelling obsolete searches in a Typeahead

But more than that, we would need a way to cancel previous

searches, as a new search gets started.

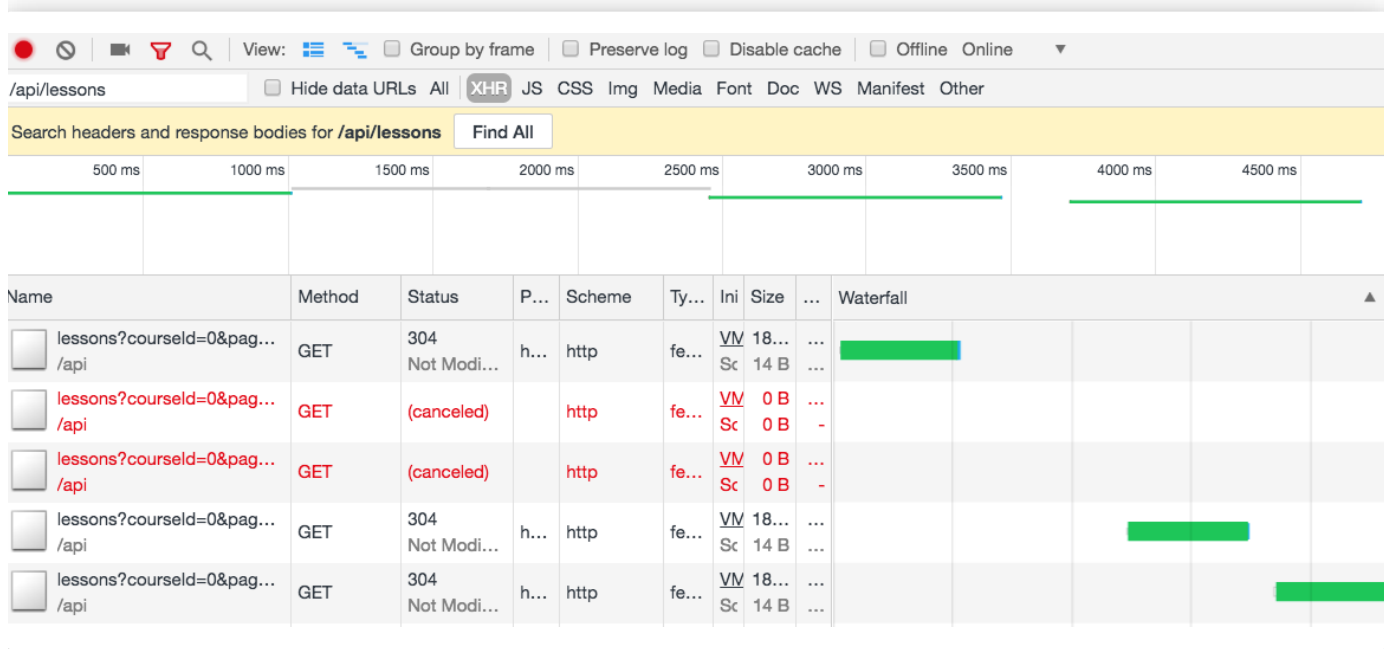
What we want to do here is to transform each search string into a backend search request and subscribe to it, and apply the switch strategy between two consecutive search requests, causing the previous search to be canceled if a new search gets triggered.

And that is exactly what the `switchMap` operator will do! Here is the final implementation of our Typeahead logic that uses it:

```
1
2  const searchText$: Observable<string> = fromEvent<any>(this.input.nativeElement, 'input')
3    .pipe(
4      map(event => event.target.value),
5      startWith(''),
6      debounceTime(400),
7      distinctUntilChanged()
8    );
9
10 const lessons$: Observable<Lesson[]> = searchText$
11   .pipe(
12     switchMap(search => this.loadLessons(search))
13   )
14   .subscribe();
15
16 function loadLessons(search:string): Observable<Lesson[]> {
17
18   const params = new HttpParams().set('search', search);
19
20   return this.http.get(`/api/lessons/${coursesId}`, {params});
21 }
22
```

## switchMap Demo with a Typeahead

Let's now see the switchMap operator in action! If the user types on the search bar, and then hesitates and types something else, here is what we can typically see in the network log:



As we can see, several of the previous searches have been canceled as they were ongoing, which is awesome because that will release server resources that can then be used for other things.

## The Exhaust Strategy

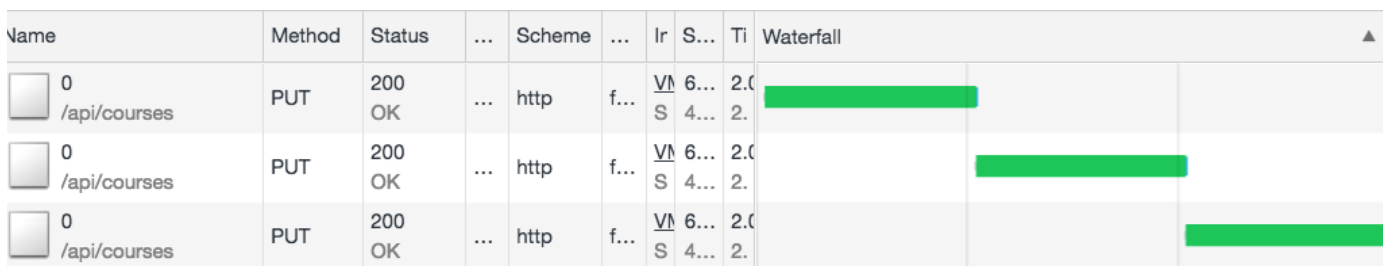
The switchMap operator is ideal for the typeahead scenario, but there are other situations where what we want to do is to *ignore* new values in the source Observable until the previous value is completely processed.

For example, let's say that we are triggering a backend save request in response to a click in a save button. We might try first to implement this using the concatMap operator, in order to ensure

that the save operations happen in sequence:

```
1
2 fromEvent(this.saveButton.nativeElement, 'click')
3   .pipe(
4     concatMap(() => this.saveCourse(this.form.value))
5   )
6   .subscribe();
7
```

This ensures the saves are done in sequence, but what happens now if the user clicks the save button multiple times? Here is what we will see in the network log:



Name	Method	Status	...	Scheme	...	Ir	S...	Ti	Waterfall
0 /api/courses	PUT	200 OK	...	http	f...	VN S	6... 4...	2.0 2.	
0 /api/courses	PUT	200 OK	...	http	f...	VN S	6... 4...	2.0 2.	
0 /api/courses	PUT	200 OK	...	http	f...	VN S	6... 4...	2.0 2.	

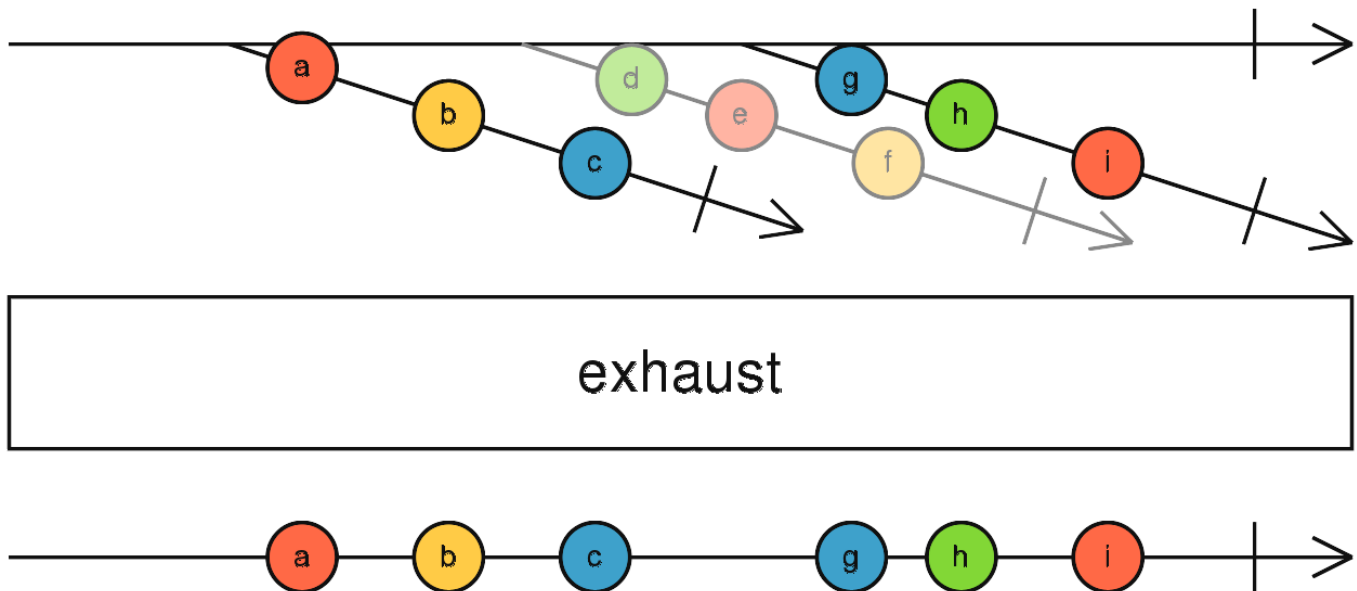
As we can see, each click triggers its own save: if we click 20 times, we get 20 saves! In this case, we would like something more than just ensuring that the saves happen in sequence.

We want also to be able to ignore a click, but only *if* a save is already ongoing. The exhaust Observable combination strategy will allow us to do just that.

## Exhaust Marble Diagram

To understand how exhaust works, let's have a look at this marble

diagram:



Just like before, we have here a higher-order Observable on the first line, whose values are themselves Observables, forking from that top line. Here is what is going on in this diagram:

- Just like in the case of switch, exhaust is subscribing to the first inner Observable (a-b-c)
- The value a, b and c get immediately reflected in the output, as usual
- then a second inner Observable (d-e-f) is emitted, while the first Observable (a-b-c) is still ongoing
- This second Observable gets *discarded* by the exhaust strategy, and it will not be subscribed to (this is the key part of exhaust)

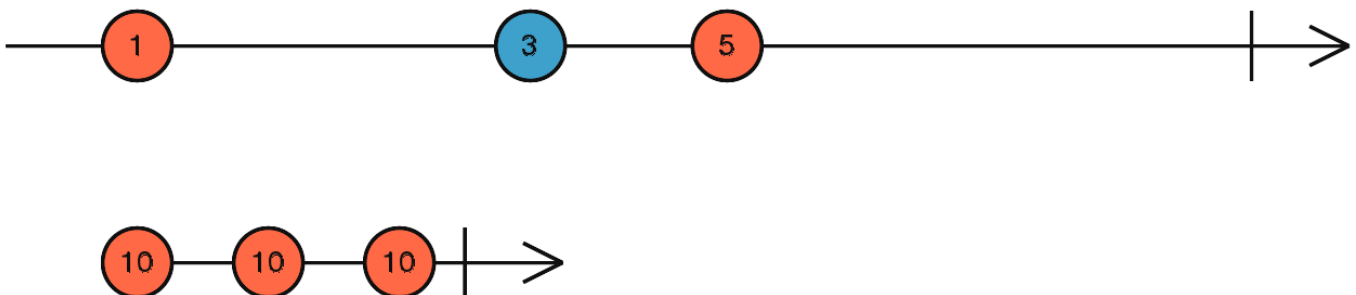
- only after the first Observable (a-b-c) completes, will the exhaust strategy subscribe to new Observables
- when the third Observable (g-h-i) is emitted, the first Observable (a-b-c) has already completed, and so this third Observable will not be discarded and will be subscribed to
- the values g-h-i of the third Observable will then show up in the output of the result Observable, unlike to values d-e-f that are *not* present in the output

Just like the case of concat, merge and switch, we can now apply the exhaust strategy in the context of higher-order mapping.

## The RxJs exhaustMap Operator

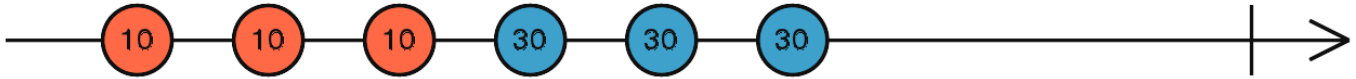
Let's now have a look at the marble diagram of the exhaustMap operator. Let's remember, unlike the top line of the previous diagram, the source Observable 1-3-5 is emitting values that are *not* Observables.

Instead, these values could for example be mouse clicks:





```
exhaustMap(i => 10*i——10*i——10*i—| )
```



So here is what is going on in the case of the exhaustMap diagram:

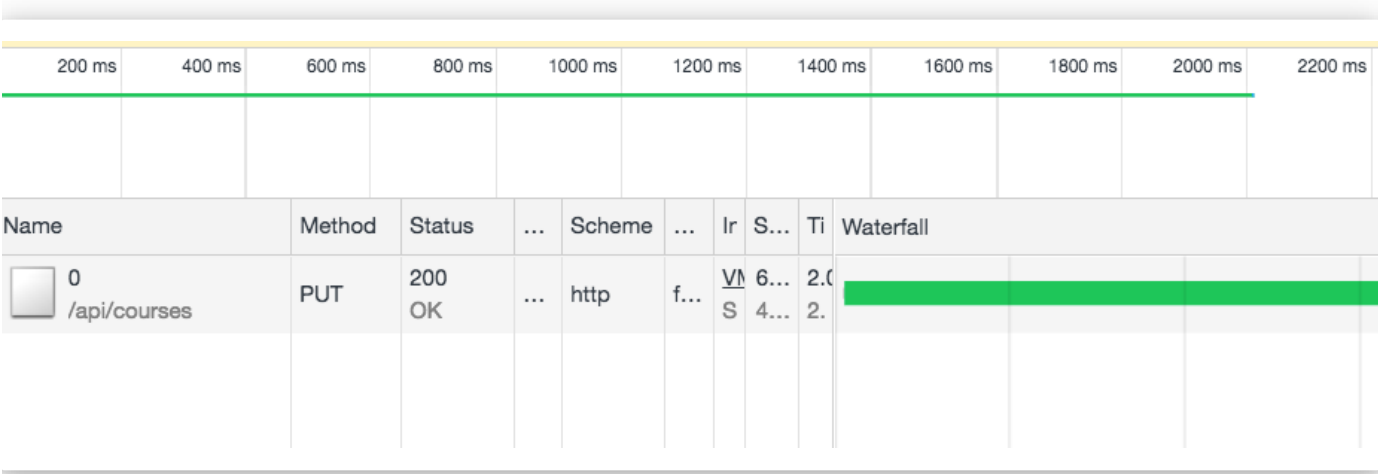
- the value 1 gets emitted, and a inner Observable 10-10-10 is created
- the Observable 10-10-10 emits all values and completes before the value 3 gets emitted in the source Observable, so all 10-10-10 values where emitted in the output
- a new value 3 gets emitted in the input, that triggers a new 30-30-30 inner Observable
- but now, while 30-30-30 is still running, we get a new value 5 emitted in the source Observable
- this value 5 is discarded by the exhaust strategy, meaning that a 50-50-50 Observable was never created, and so the 50-50-50 values never showed up in the output

## A Practical Example for exhaustMap

Let's now apply this new exhaustMap Operator to our save button scenario:

```
2 fromEvent(this.saveButton.nativeElement, 'click')
3   .pipe(
4     exhaustMap(() => this.saveCourse(this.form.value))
5   )
6   .subscribe();
7
```

If we now click save let's say 5 times in a row, we are going to get the following network log:



As we can see, the clicks that we made while a save request was still ongoing were ignored, as expected!

Notice that if we would keep clicking for example 20 times in a row, eventually the ongoing save request would finish and a second save request would then start.

## How to choose the right mapping Operator?

The behavior of `concatMap`, `mergeMap`, `switchMap` and `exhaustMap` is similar in the sense they are all higher order mapping operators.

But its also so different in so many subtle ways, that there isn't really one operator that can be safely pointed to as a default.

Instead, we can simply choose the appropriate operator based on the use case:

- if we need to do things in sequence while waiting for completion, then `concatMap` is the right choice
- for doing things in parallel, `mergeMap` is the best option
- in case we need cancellation logic, `switchMap` is the way to go
- for ignoring new Observables while the current one is still ongoing, `exhaustMap` does just that

## Running GitHub repo (with code samples)

If you want to try out the examples in this post, here is a playground [repository](#) containing the running code for this post.

This repository includes a small HTTP backend that will help to try out the RxJs mapping operators in a more realistic scenario, and includes running examples like the draft form pre-save, a typeahead, subjects and examples of components written in Reactive style:



## RxJs In Practice Course



Type your search

#	Description	Duration
1	Welcome to the RxJs In Practice Course	4:17
2	Introduction to the Observable Pattern	2:07
3	The map and filter RxJs Operators	2:33
4	The switchMap, concatMap and mergeMap RxJs Operators	4:44
5	Error Handling in RxJs	2:55

## Conclusions

As we have seen, the RxJs higher-order mapping operators are essential for doing some very common operations in reactive programming, like network calls.

In order to really understand these mapping operators and their names, we need to first focus on understanding the underlying

Observable combination strategies concat, merge, switch and exhaust.

We also need to realize that there is a higher order mapping operation taking place, where values are being transformed into separated Observables, and those Observables are getting subscribed to in a hidden way *by the mapping operator itself*.

Choosing the right operator is all about choosing the right inner Observable combination strategy. Choosing the wrong operator often does not result in an immediately broken program, but it might lead to some hard to troubleshoot issues over time.

I hope that you have enjoyed this post, if you are looking to learn more about RxJs, you might want to check out our other RxJs posts in the [RxJs Series](#).