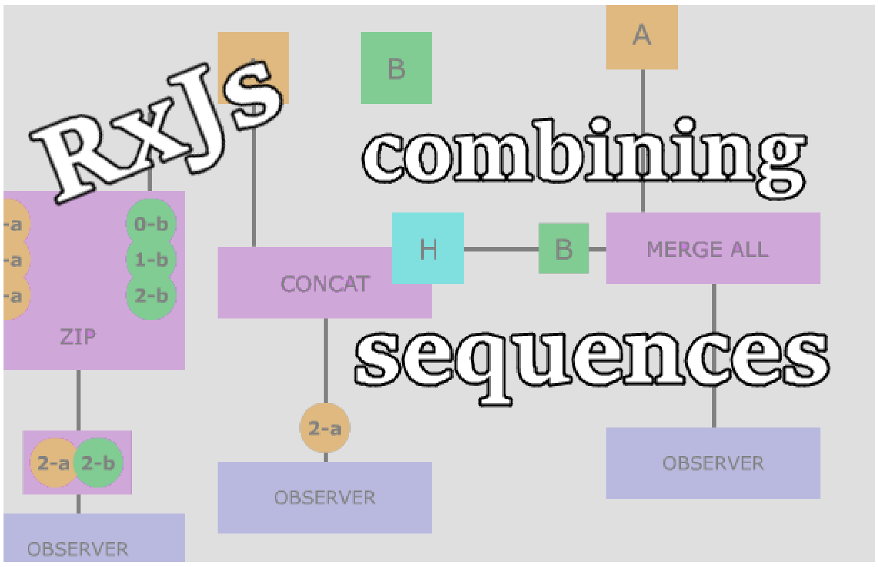
Learn to combine RxJs sequences with super intuitive interactive diagrams

Max Koretskyi aka Wizard Follow
Dec 14, 2017 · 14 min read



> **We're organizing the first Angular conference in Kiev, Ukraine as part of Angular In Depth community. Read more about it here.** I'll be happy to see you at the conference. We'll have experts from Angular team and the community and we made tickets affordable for most developers (around $150), even if you pay out of your own pocket.

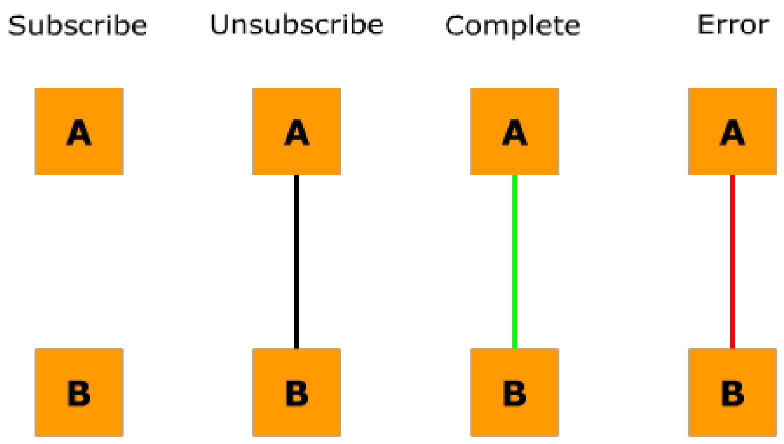
When working on a sufficiently complex application you usually have data coming from more than one data source. It can be some multiple external data points like Firebase or several UI widgets interacting with a user. Sequence composition is a technique that enables you to create complex queries across multiple data sources by combing relevant streams into one. RxJs provides a variety of operators that can help you do that and in this article we'll take a look at the most commonly used.

I’ve even become part time animation specialist to design and create most intuitive data flow diagrams that demonstrate the difference between all the operators. However, the diagrams are embedded as an animated GIF so it takes a while for all of them to load. Please be patient.

In the accompanying code I’ll be using lettable operators so if you’re not familiar with them you can do it here. I’ll also be using a custom `stream` operator that produces a stream of values asynchronously with the first item delivered synchronously upon subscription.

> I work as a developer advocate at **ag-Grid**. If you’re curious to learn about data grids or looking for the ultimate Angular data grid solution, give it a try with the guide “**Get started with Angular grid in 5 minutes**”. I’m happy to answer any questions you may have. **And follow me to stay tuned!**

And here is the legend for the type of diagrams I’ll be using throughout this article:



. . .

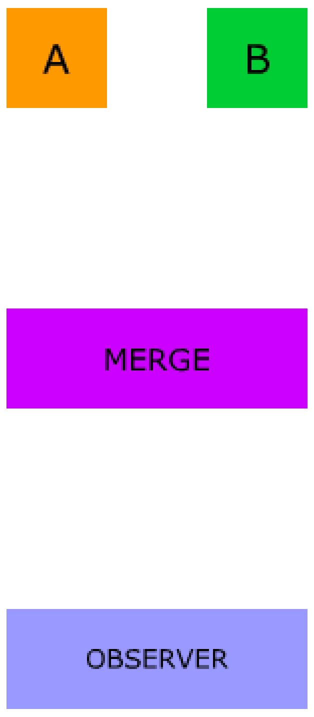
Merging multiple sequences concurrently

The first operator we’ll take a look at is **merge**. This operator combines a number of observables streams and concurrently emits all values from every given input stream. As values from any combined sequence are produced, those values are emitted as part of the resulting sequence. Such process is often referred to as flattening in documentation.

The stream completes when all input streams complete and will throw an error if any of the streams throws an error. It will never complete if some of the input streams don't complete.

Use this operator if you're not concerned with the order of emissions and is simply interested in all values coming out from multiple combined streams as if they were produced by one stream.

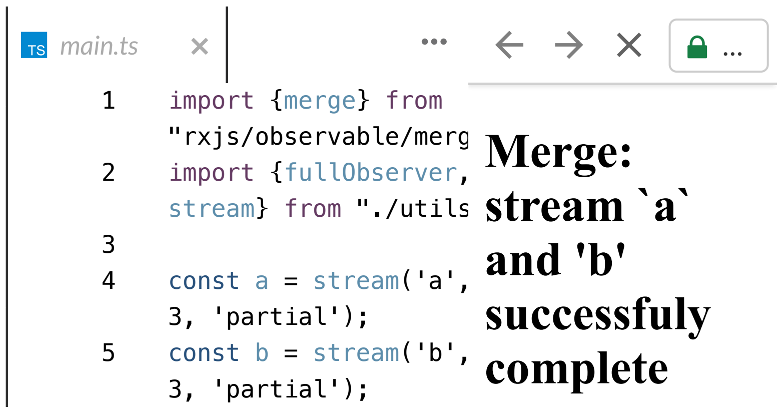
In the diagram below you can see the `merge` operator combining two streams `A` and `B` each producing 3 items and the values falling through to the resulting sequence as they occur.



Here is the code example that demonstrates the setup shown by the above diagram:

```
const a = stream('a', 200, 3, 'partial');
const b = stream('b', 200, 3, 'partial');
merge(a, b).subscribe(fullObserver('merge'));

// can also be used as an instance operator
a.pipe(merge(b)).subscribe(fullObserver('merge'));
```

And stackblitz editable demo:



```
import {merge} from
"rxjs/observable/merg
import {fullObserver,
stream} from "./utils

const a = stream('a',
3, 'partial');
const b = stream('b',
3, 'partial');
```

combining-sequences-merge Editor Preview Both

• • •

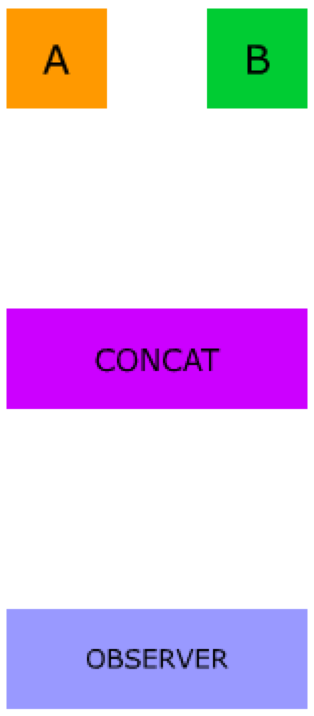
Concatenating multiple sequences sequentially

The next composition method is **concat**. It concatenates streams by subscribing and emitting values from each input stream sequentially having only one active subscription at a time. Once the current stream completes it subscribes to next sequence and passes its values on through to the resulting sequence.

The stream completes when all input streams complete and will throw an error if some of the input streams throw an error. It will never complete if some of the input streams don't complete which also means that some streams will never be subscribed.

Use this operator if the order of emissions is important and you want to first see values emitted by streams that you pass first to the operator. For example, you may have an observable sequences that delivers values from a cache and another sequence that delivers values from a remote server. Use `concat` if you want to combine them and ensure that the value from cache is delivered first.

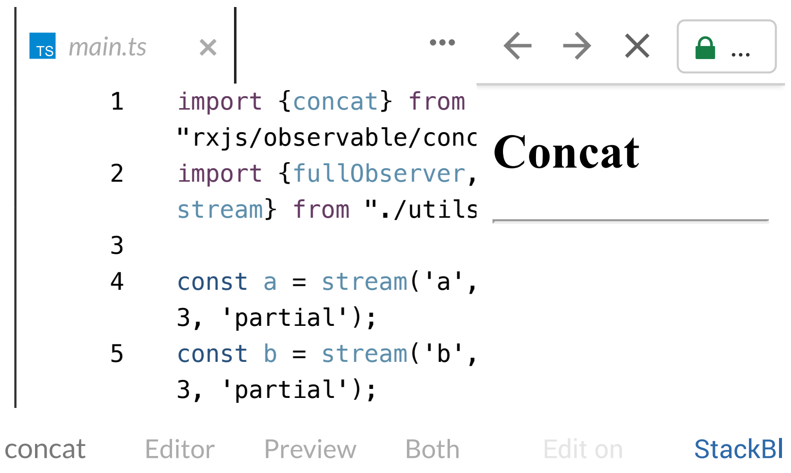
In the diagram below you can see the `concat` operator combining two streams `A` and `B` each producing 3 items and the values falling through to the resulting sequence first from the `A` and then from the `B`.



Here is the code example that demonstrates the setup shown by the above diagram:

```
const a = stream('a', 200, 3, 'partial');
const b = stream('b', 200, 3, 'partial');
concat(a, b).subscribe(fullObserver('concat'));

// can also be used as an instance operator
a.pipe(concat(b)).subscribe(fullObserver('concat'));
```

And stackblitz editable demo:



```
import {concat} from
"rxjs/observable/conc
import {fullObserver,
stream} from "./utils

const a = stream('a',
3, 'partial');
const b = stream('b',
3, 'partial');
```

Concat

. . .

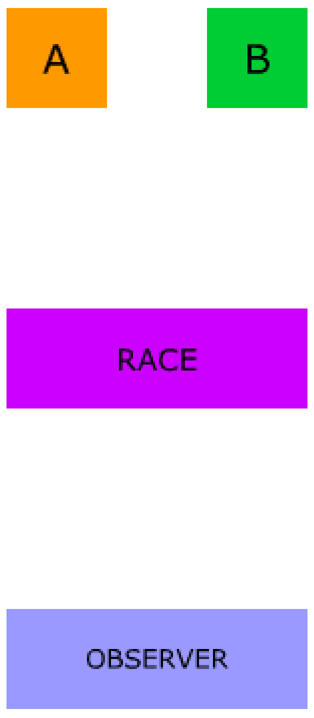
Combining sequences ambigously

The next operator `race` introduces a pretty interesting concept. It doesn't combine sequences per se but is rather used to select an observable sequences that is the first to produce values. As soon as one of the sequences starts emitting values the other sequences are unsubscribed and completely ignored.

The resulting stream completes when the selected input stream completes and will throw an error if this one stream errors out. It will also never complete if this inner stream doesn't complete.

This operator can be useful if you have multiple resources that can provide values, for example, servers around the world, but due to network conditions the latency is not predictable and varies significantly. Using this operator you can send the same request out to multiple data sources and consume the result of the first that responds.

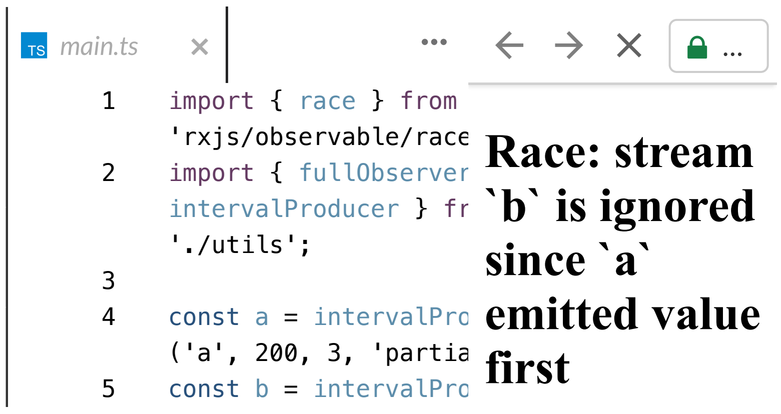
In the diagram below you can see the `race` operator combining two streams `A` and `B` each producing 3 items but only the values from the stream `A` are emitted since this stream starts emitting values first.



Here is the code example that demonstrates the setup shown by the above diagram:

```
const a = intervalProducer('a', 200, 3, 'partial');
const b = intervalProducer('b', 500, 3, 'partial');

race(a, b).subscribe(fullObserver('race'));

// can also be used as an instance operator
a.pipe(race(b)).subscribe(fullObserver('race'));
```

And stackblitz editable demo:



```
import { race } from
'rxjs/observable/race
import { fullObserver
intervalProducer } fr
'./utils';

const a = intervalPro
('a', 200, 3, 'partia
const b = intervalPro
```

Race: stream \`b\` is ignored since \`a\` emitted value first

combining-sequences-race-b-is-ignored   Editor   Preview

. . .

Combing unknown number of sequences with higher-order observables

The operators I've shown above, either as a static or an instance version, can only be used to compose a known number of sequences. But what if you don't know all the sequences beforehand and want to merge sequences that can be evaluated lazily at run time. In fact, this is a very common situation when working with asynchronous code. For example, a network call for some resource can result in a number of other requests determined by the resulting value of the original request.

RxJs has a variations of the operators we've seen above that take a sequence of sequences, so called higher-order observables or Observable-of-Observables. The operators expect emissions from such observables to be sequences and work with them according to the rules we saw in the first chapter.

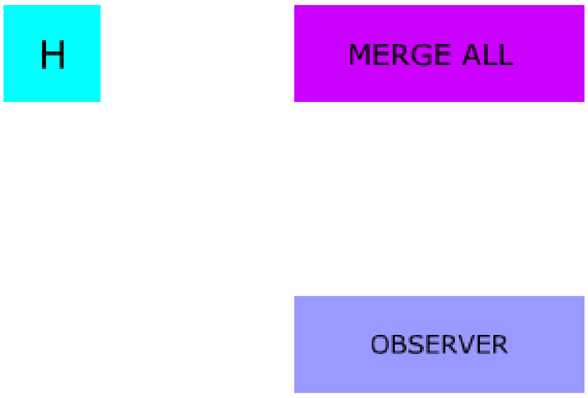
All such operators emit an error of any of the inner streams produce an error and can only be used as instance operators. Now let's take a look at them one by one.

. . .

MergeAll

This operator combines all emitted inner streams and just as with plain `merge` concurrently produces values from each stream.

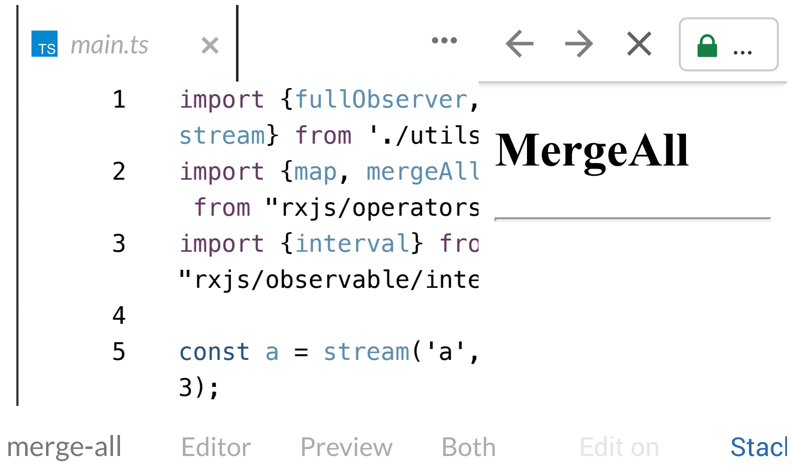
In the diagram below you can see the `H` higher-order stream that produces two inner streams `A` and `B`. The `mergeAll` operator combines values from these two streams and then passes them through to the resulting sequence as they occur.



Here is the code example that demonstrates the setup shown by the above diagram:

```
const a = stream('a', 200, 3);
const b = stream('b', 200, 3);
const h = interval(100).pipe(take(2), map(i => [a, b][i]));

h.pipe(mergeAll()).subscribe(fullObserver('mergeAll'));
```

And stackblitz editable demo:



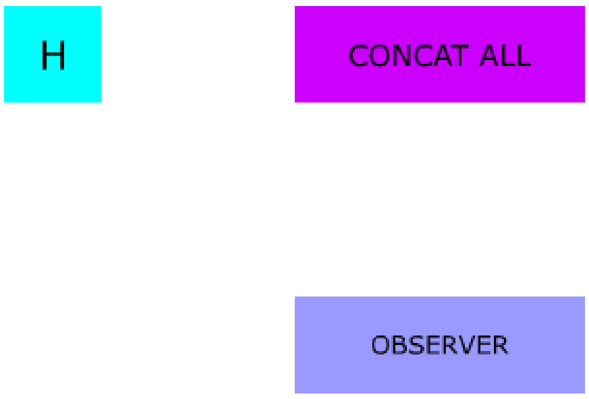
```
import {fullObserver,
stream} from './utils
import {map, mergeAll
 from "rxjs/operators
import {interval} fro
"rxjs/observable/inte

const a = stream('a',
3);
```

. . .

ConcatAll

This operator combines all emitted inner streams and just as with plain `concat` sequentially produces values from each stream.

In the diagram below you can see the `H` higher-order stream that produces two inner streams `A` and `B`. The `concatAll` operator takes values from the `A` stream first and then from the stream `B` and passes them through the resulting sequence.



Here is the code example that demonstrates the setup shown by the above diagram:

```
const a = stream('a', 200, 3);
const b = stream('b', 200, 3);
const h = interval(100).pipe(take(2), map(i => [a, b][i]));

h.pipe(concatAll()).subscribe(fullObserver('concatAll'));
```

And stackblitz editable demo:

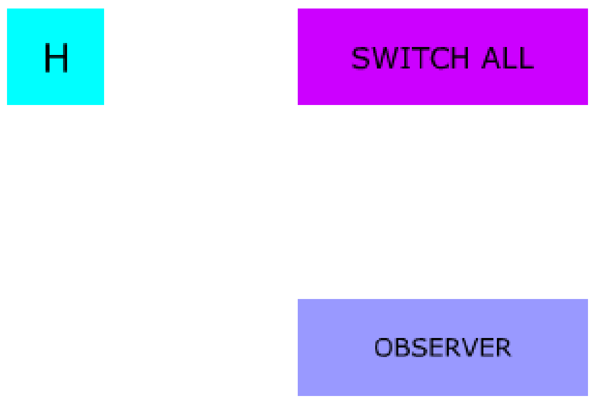
```
import {fullObserver,
stream} from './utils
import {concatAll, ma
take} from "rxjs/oper
import {interval} fro
"rxjs/observable/inte

const a = stream('a',
3);
```

• • •

SwitchAll

Sometimes receiving values from all inner observable sequences is not what we need. In some scenarios, we may only be interested in the the values from the most recent inner sequence. A good example of such functionality is search. As a user types in some text, the request is sent to a server and since it's asynchronous the result is returned as an observable. What if the user updates the text in the search-box before the result is returned? The second request is sent and so by now two searches have been sent to the server. However, the first search contains the results that we are no longer interested in. Furthermore, if the result for the first search was merged together with result for the second search, the user would be very surprised. We don't want that and so this is where `switchAll` operator comes in. It subscribes and produces values only from the most recent inner sequence ignoring previous streams.

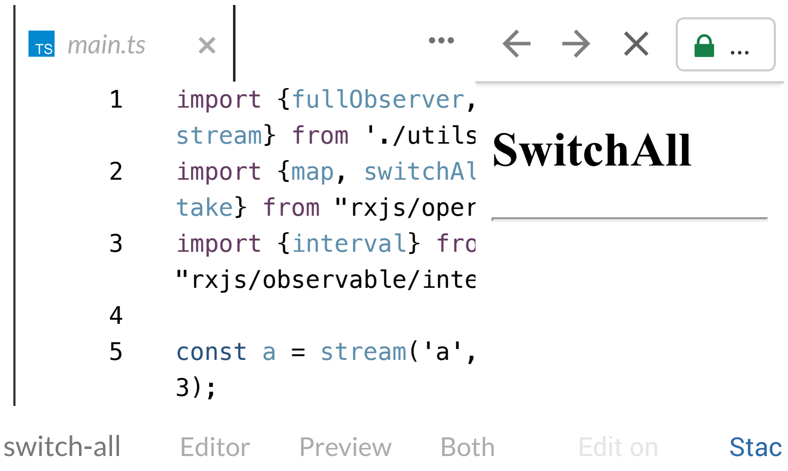
In the diagram below you can see the `H` higher-order stream that produces two inner streams `A` and `B`. The `switchAll` operator takes values from the `A` stream first and then from the stream `B` and passes them through the resulting sequence.



Here is the code example that demonstrates the setup shown by the above diagram:

```
const a = stream('a', 200, 3);
const b = stream('b', 200, 3);
const h = interval(100).pipe(take(2), map(i => [a, b][i]));

h.pipe(switchAll()).subscribe(fullObserver('switchAll'));
```

And stackblitz editable demo:



```
import {fullObserver,
stream} from './utils
import {map, switchAl
take} from "rxjs/oper
import {interval} fro
"rxjs/observable/inte

const a = stream('a',
3);
```

. . .

concatMap, mergeMap and switchMap

Interestingly, the mapping operators `concatMap` , `mergeMap` and `switchMap` are used much more often than their counterparts `concatAll` , `mergeAll` and `switchAll` that operate on the stream of observables. Yet, if you think about it, they are almost the same thing. All `*Map` operators consist of two parts—producing a stream of observables through mapping and applying combination logic on the inner streams produced by this higher order observable.

Let's take a look at the following familiar code that demonstrates how `mergeAll` operator works:

```
const a = stream('a', 200, 3);
const b = stream('b', 200, 3);
const h = interval(100).pipe(take(2), map(i => [a, b][i]));

h.pipe(mergeAll()).subscribe(fullObserver('mergeAll'));
```

Here `map` operator produces a stream of observables and `mergeAll` combines values from these observables and so we can easily replace `map` and `mergeAll` with the `mergeMap` like this:

```
const a = stream('a', 200, 3);
const b = stream('b', 200, 3);
const h = interval(100).pipe(take(2), mergeMap(i => [a, b]
[i]));

h.subscribe(fullObserver('mergeMap'));
```

The result will be the exactly the same. The same holds true for both `concatMap` and `switchMap` —try it on your own.

• • •

Combing sequences by pairing their values

The previous operators allowed us to flatten multiple sequences and deliver values from those sequences unchanged through the resulting stream as if they all come from this one sequence. The set of operators we'll take a look next still take multiple sequences as an input, but differ in that they pair values from each sequence to produce a single combined value for the output sequence.

Each operator can take an optional so-called projection function as the last parameter that defines how the values from the resulting sequence should be combined. In my examples I'll be using the default projection function that simply joins values using comma as a separator. I'll show how to provide a custom projection function in the end of the section.

• • •

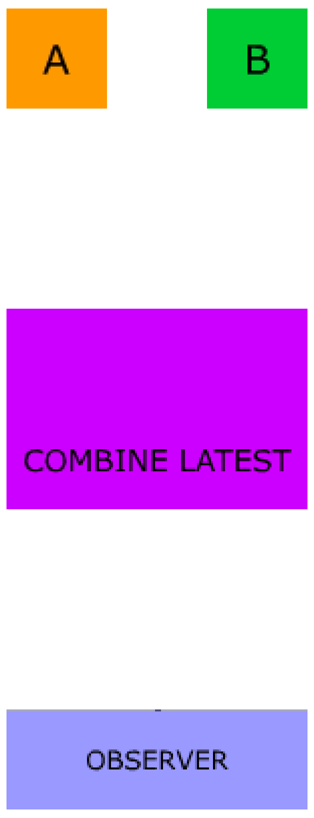
CombineLatest

The first operator we'll review is `combineLatest`. It allows you to take the most recent value from input sequences and transform those into one value for the resulting sequence. RxJs caches last value for each input sequence and once all sequences have produced at least one value it computes a resulting value using projection function that takes the latest values from the cache, then emits the output of that computation through the result stream.

The resulting stream completes when all inner streams complete and will throw an error if any of the inner streams throws an error. It will never complete if any of the inner streams doesn't complete. On the other hand, if any stream does not emit value but completes, resulting stream will complete at the same moment without emitting anything, since it will be now impossible to include value from completed input stream in resulting sequence. Also, if some input stream does not emit any value and never completes, `combineLatest` will also never emit and never complete, since, again, it will wait for all streams to emit some value.

This operator can be useful if you need to evaluate some combination of state which needs to be kept up-to-date when part of the state changes. A simple example would be a monitoring system. Each service is represented by a sequence that returns a Boolean indicating the availability of said service. The monitoring status is green if all services are available so the projection function should simply perform a logical AND.

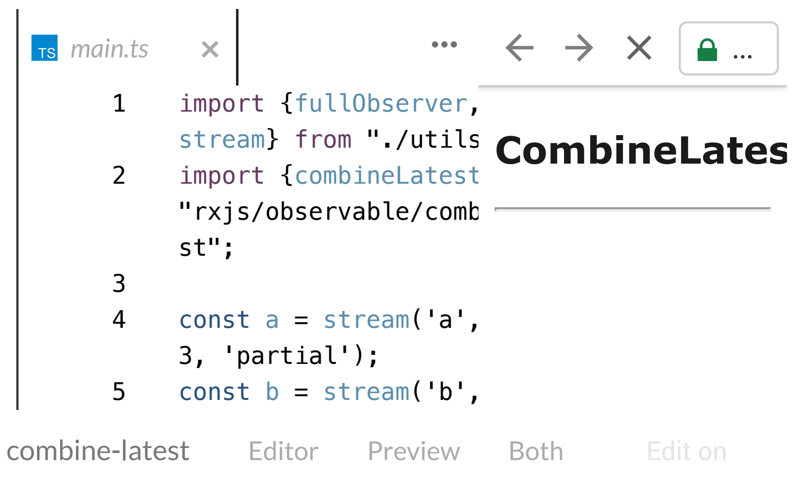
In the diagram below you can see the `combineLatest` operator combining two streams `A` and `B`. As soon as all streams have emitted at least one value each new emission produces a combined value through the result stream:



Here is the code example that demonstrates the setup shown by the above diagram:

```
const a = stream('a', 200, 3, 'partial');
const b = stream('b', 500, 3, 'partial');

combineLatest(a, b).subscribe(fullObserver('latest'));
```

And stackblitz editable demo:



```
import {fullObserver,
stream} from "./utils
import {combineLatest
"rxjs/observable/comb
st";

const a = stream('a',
3, 'partial');
const b = stream('b',
```

CombineLates

. . .

Zip

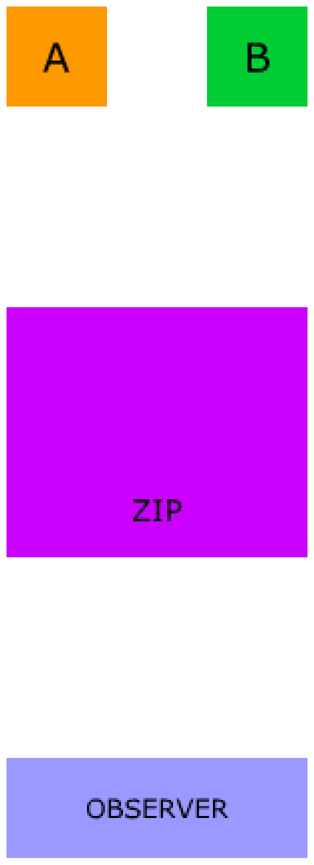
This operator is another interesting merge feature that in some way resembles the mechanics of a zipper on clothing or a bag. It brings together two or more sequences of *corresponding* values as a tuple (a pair in case of two input streams). It waits for the corresponding value to be emitted from all input streams, then transforms them into a single value using projection function and emits the result. It will only publish once it has a pair of fresh values from each source sequence so if one of the source sequences publishes values faster than the other sequence, the rate of publishing will be dictated by the slower of the two sequences.

The resulting stream completes when any of the inner streams complete and the corresponding matched pairs are emitted from other streams. It will never complete if any of the inner streams doesn't complete and will throw an error if any of the inner streams errors out.

This operator can be conveniently used to implement a stream that produces a range of values with an interval. Here is the basic example with the projection function returning values only from the `range` stream:

```
zip(range(3, 5), interval(500), v => v).subscribe();
```

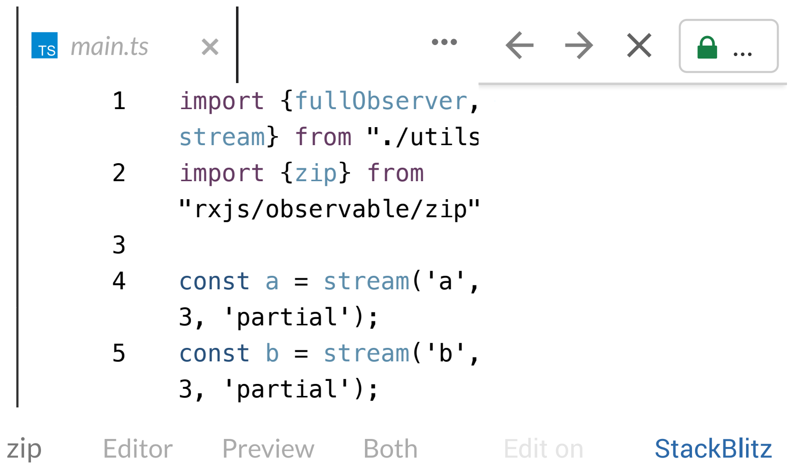
In the diagram below you can see the `zip` operator combining two streams `A` and `B`. As soon as a corresponding pair is matched the resulting sequence produces a combined value:



Here is the code example that demonstrates the setup shown by the above diagram:

```
const a = stream('a', 200, 3, 'partial');
const b = stream('b', 500, 3, 'partial');

zip(a, b).subscribe(fullObserver('zip'));
```

And stackblitz editable demo:



```
import {fullObserver, stream} from "./utils
import {zip} from "rxjs/observable/zip"

const a = stream('a', 3, 'partial');
const b = stream('b', 3, 'partial');
```

forkJoin

Sometimes you have a group of streams and only care about the final emitted value of each. Often such sequences have only a single emission. For example, you may want to make multiples network requests and only want to take an action when a response has been received for all of them. It is in some way similar to Promise.all functionality. However, if you have a stream that emits more than one item, those items will be ignored except for the last value.

The resulting stream emits only one time when all of the inner streams complete. It will never complete if any of the inner streams doesn't complete and will throw an error if any of the inner streams errors out.

In the diagram below you can see the `forkJoin` operator combining two streams `A` and `B`. As soon as a corresponding pair is matched the resulting sequence produces a combined value:

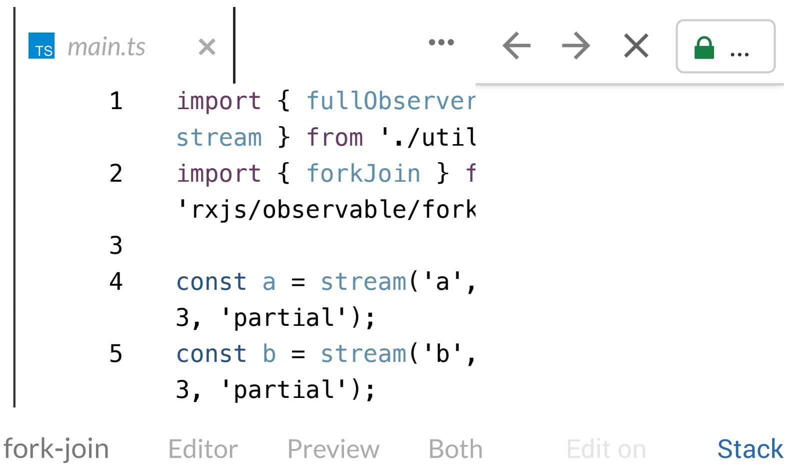
A
B
FORK JOIN
OBSERVER

Here is the code example that demonstrates the setup shown by the above diagram:

```
const a = stream('a', 200, 3, 'partial');
const b = stream('b', 500, 3, 'partial');
```

```
forkJoin(a, b).subscribe(fullObserver('forkJoin'));
```

And stackblitz editable demo:



```
import { fullObserver
stream } from './util
import { forkJoin } f
'rxjs/observable/fork

const a = stream('a',
3, 'partial');
const b = stream('b',
3, 'partial');
```

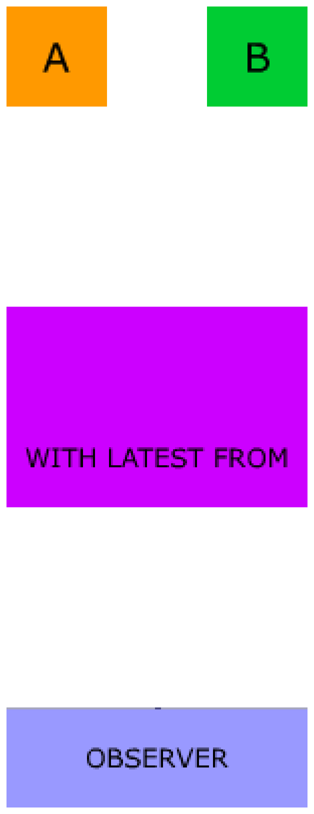
• • •

WithLatestFrom

The last operator we'll take a look in this article is `withLatestFrom`. This operator used when you have one guiding stream but also need latest values from other streams. While the similar `combineLatest` operator emits a new value whenever there's a new emission from any of the input streams, `withLatestFrom` emits a new value only if there's a new emission from the guiding stream.

Just as with `combineLatest` it still waits for at least one emitted value from each stream and may complete without a single emission when the guiding stream completes. It will never complete if the guiding stream doesn't complete and will throw an error if any of the inner streams errors out.

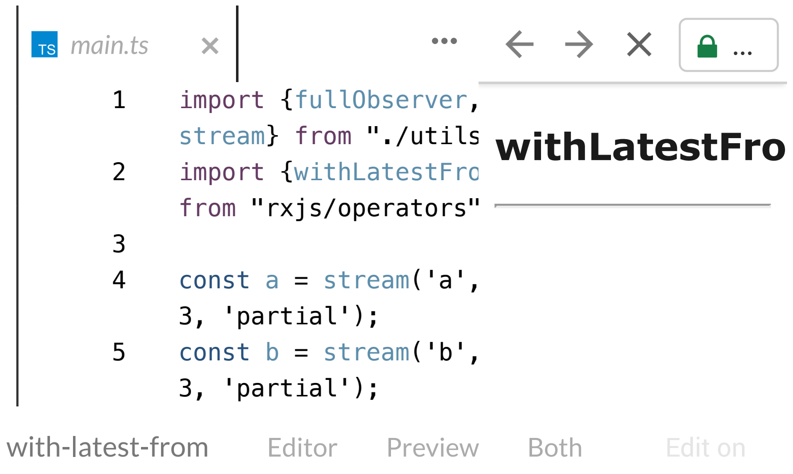
In the diagram below you can see the `withLatestFrom` operator combining two streams `A` and `B` with the stream `B` being the guiding stream. Every time the stream `B` emits a new value the resulting sequence produces a combined value using latest value from the stream `A`:



Here is the code example that demonstrates the setup shown by the above diagram:

```
const a = stream('a', 3000, 3, 'partial');
const b = stream('b', 500, 3, 'partial');

b.pipe(withLatestFrom(a)).subscribe(fullObserver('latest'));
```

And stackblitz editable demo:



```
import {fullObserver,
stream} from "./utils
import {withLatestFro
from "rxjs/operators"

const a = stream('a',
3, 'partial');
const b = stream('b',
3, 'partial');
```

withLatestFro

• • •

Projection function

As mentioned in the beginning of the section all operators that combine values by pairing take an optional projection function. This function defines the transformation for the resulting value. Using this function you can choose to only emit a value from a particular input sequence or to join values in any way you want:

```
// return value from the second sequence
zip(s1, s2, s3, (v1, v2, v3) => v2)

// join values using dash as a separator
zip(s1, s2, s3, (v1, v2, v3) => `${v1}-${v2}-${v3}`)

// return single boolean result
zip(s1, s2, s3, (v1, v2, v3) => v1 && v2 && v3)
```

. . .

If you want to have all diagrams in one place, see the gist by Pierre Criulanscy

. . .

Thanks for reading! If you liked this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story.

For more insights follow me on Twitter and on Medium.