MAY 13, 2020 / #REACTIVE PROGRAMMING

How to Understand RxJS Operators by Eating a Pizza: zip, forkJoin, & combineLatest Explained with Examples

Samuel Teboul

What is RxJS?

> *Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change - **Wikipedia***

> RxJS is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code - **RxJS docs**

The essential concepts in RxJS are

- **An Observable** is a stream of data
- **Observers** can register up to 3 callbacks:

2. *error* is called at most 1 time when an error occurred

3. *complete* is called at most 1 time on completion

- **Subscription** "kicks off" the observable stream

Without subscribing the stream won't start emitting values. This is what we call a **cold observable.**

It's similar to subscribing to a newspaper or magazine... you won't start getting them until you subscribe. Then, it creates a 1 to 1 relationship between the producer (observable) and the consumer (observer).

MfSs4WItSaYoHHK6TS7MlN1O5pSZsN98hA6af6L0j_MHh5F7bL8_Vm3fiya9Vw3Xwr4E(

What are RxJS operators?

Operators are pure functions that enable a functional programming style of dealing with collections with operations. There are two kinds of operators:

- Creation operators
- Pipeable operators: transformation, filtering, rate limiting, flattening

Subjects are a special type of Observable that allows values to be **multicast** to many Observers. While plain Observables are **unicast** (each subscribed Observer owns an independent execution of the Observable), Subjects are multicast. This is what we call a **hot observable.**

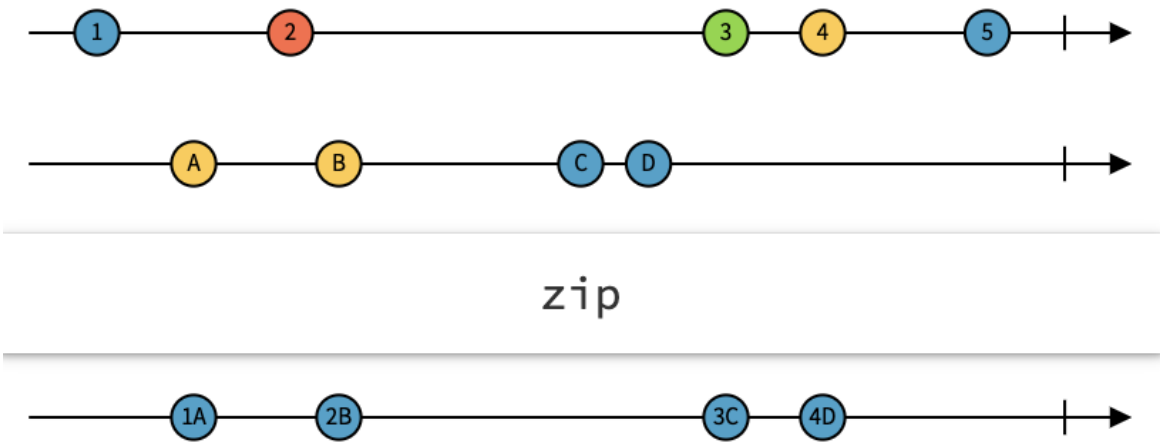
In this article, I will focus on the `zip`, `combineLatest` and `forkJoin` operators. These are RxJS combination operators, which means that they enable us to join information from multiple observables. Order, time, and structure of emitted values are the primary differences among them.

Let's look at each one individually.

zip()

- `zip` doesn't start to emit until each inner observable emits at

- `zip` emits values as an array



Let's imagine that you are with Mario and Luigi, two of your best friends, at the best Italian restaurant in Rome. Each one of you orders a drink, a pizza, and a dessert. You specify to the waiter to bring the drinks first, then the pizzas, and finally the desserts.

This situation can be represented with 3 different observables, representing the 3 different orders. In this specific situation, the waiter can use the `zip` operator to bring (emit) the different order items by category.

```
const you$ = ['Cola Zero', 'Margherita Pizza', 'Tiramisu'];
const mario$ = ['Sprite', 'Carbonara Pizza', 'Fruits salad'];
const luigi$ = ['Pepsi', 'Quattro Formaggi Pizza', 'Ice cream'];

const waiter$ = zip(
  from(you$),
  from(mario$),
  from(luigi$)
);

waiter$.subscribe(
  next => console.log(next),
  error => console.log(error),
  () => console.log('completed!')
);
// ["Cola Zero", "Sprite", "Pepsi"]
// ["Margherita Pizza", "Carbonara Pizza", "Quattro Formaggi Pizza"]
// ["Tiramisu", "Fruits salad", "Ice cream"]
// completed!
```

Warning

If you go back to the same Italian restaurant with your girlfriend, but she doesn't want to eat, this is what will happen:

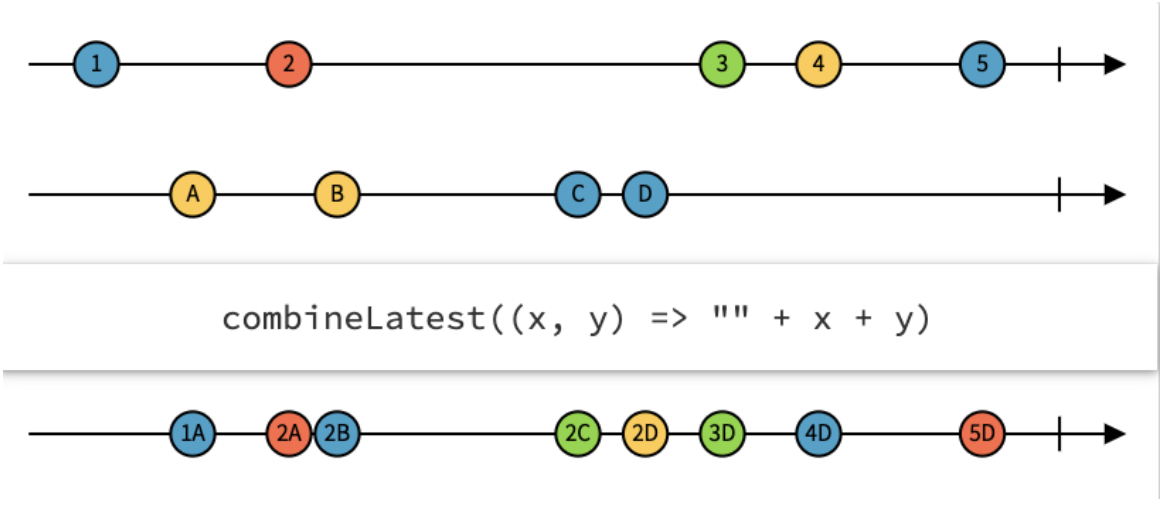
```
const you$ = ['Cola Zero', 'Margherita Pizza', 'Tiramisu'];
const girlfriend$ = ['Sprite'];

const waiter$ = zip(
  from(you$),
  from(girlfriend$)
);

waiter$.subscribe(
  next => console.log(next),
  error => console.log(error),
  () => console.log('completed!')
);

// ["Cola Zero", "Sprite"]
// completed!
```

Why?

Because, when the `waiter$` emits the drinks, the `girlfriend$` observable is complete and no more value can be collected from it. Hopefully, the `waiter$` can use another operator for us so we don't break up with our girlfriend ?

combineLatest()

- `combineLatest` doesn't start to emit until each inner observable emits at least one value
- When any inner observable emits a value, emit the last emitted value from each



At the exact same restaurant, the smart `waiter$` now decide to use `combineLatest` operator.

```
const you$ = ['Cola Zero', 'Margherita Pizza', 'Tiramisu'];
const girlfriend$ = ['Sprite'];

const waiter$ = combineLatest(
  from(girlfriend$),
  from(you$),
);

waiter$.subscribe(
  next => console.log(next),
  error => console.log(error),
  () => console.log('completed!')
);

// ["Sprite", "Cola Zero"]
// ["Sprite", "Margherita Pizza"]
// ["Sprite", "Tiramisu"]
// completed!
```

Warning

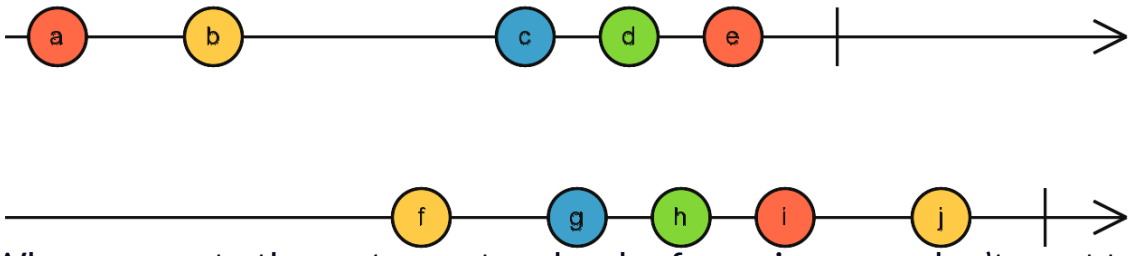
With `combineLatest`, the **order** of the provided inner observables does matter.

If `you$` is provided first to `waiter$`, it will emit only one value: `["Tiramisu", "Sprite"]`.

This is happening because `combineLatest` doesn't start to emit until each inner observable emits at least one value. `girlfriend$` starts emitting when the first inner observable emits its last value. Then, `combineLatest` emits the last values collected from both inner observables.

forkJoin()

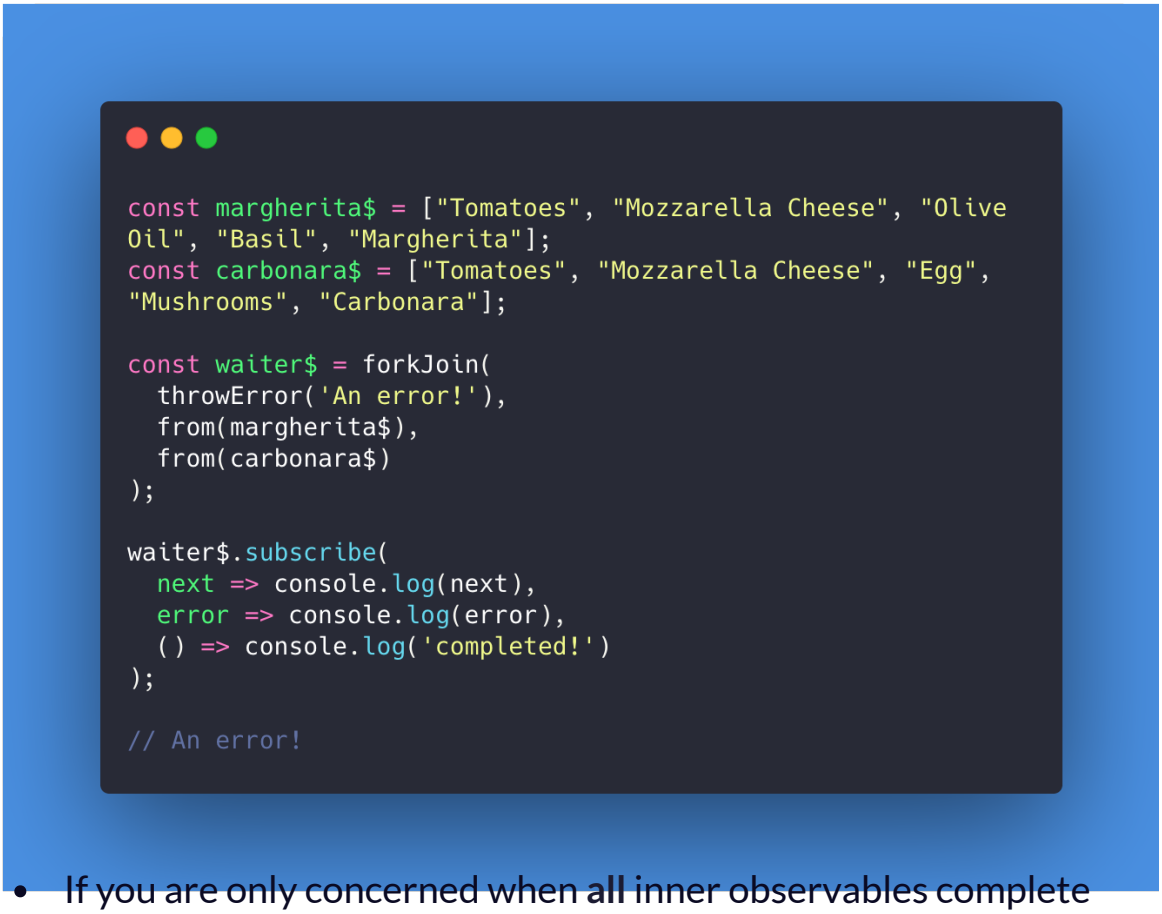
- `forkJoin` emits the last emitted value from each inner observables after they **all** complete
- `forkJoin` will never emit if one of the observables doesn't complete



When you go to the restaurant and order for a pizza, you don't want to know all the steps about how the pizza is prepared. If the cheese is added before the tomatoes or the opposite. You just want to get your pizza! This is where `forkJoin` comes into play.

Warning

- If one of the inner observables throws an error, all values are lost
- `forkJoin` doesn't complete



```
const margherita$ = ["Tomatoes", "Mozzarella Cheese", "Olive
Oil", "Basil", "Margherita"];
const carbonara$ = ["Tomatoes", "Mozzarella Cheese", "Egg",
"Mushrooms", "Carbonara"];

const waiter$ = forkJoin(
  throwError('An error!'),
  from(margherita$),
  from(carbonara$)
);

waiter$.subscribe(
  next => console.log(next),
  error => console.log(error),
  () => console.log('completed!')
);

// An error!
```

- If you are only concerned when **all** inner observables complete successfully, you can catch the error from the **outside**
- Then, `forkJoin` completes

- If you don't care that inner observables complete successfully or not you must catch errors from every single inner observable
- Then, forkJoin completes

```
const margherita$ = ["Tomatoes", "Mozzarella Cheese", "Olive Oil",
"Basil", "Margherita"];
const carbonara$ = ["Tomatoes", "Mozzarella Cheese", "Egg",
"Mushrooms", "Carbonara"];

const waiter$ = forkJoin(
  throwError('An error!'),
  from(margherita$),
  from(carbonara$)
).pipe(
  ca                    or => of(error))
);

waiter$.subscribe(
  next => console.log(next),
  error => console.log(error),
```

```
const margherita$ = ["Tomatoes", "Mozzarella Cheese", "Olive Oil",
"Basil", "Margherita"];
const carbonara$ = ["Tomatoes", "Mozzarella Cheese", "Egg",
"Mushrooms", "Carbonara"];

const waiter$ = forkJoin(
  throwError('An error!').pipe(
    catchError(error => of(error))
  ),
  from(margherita$).pipe(
    catchError(error => of(error))
  ),
  from(carbonara$).pipe(
    catchError(error => of(error))
  )
);

waiter$.subscribe(
  next => console.log(next),
  error => console.log(error),
  () => console.log('completed!')
);

// ["An error!", "Margherita", "Carbonara"]
// completed!
```

`$` to catch the errors from inner observables individually.

Wrap up

We covered a lot in this article! Good examples are important to better understand RxJS operators and how to choose them wisely.

For combination operators like `zip`, `combineLatest`, and `forkJoin` the order of inner observables that you provide is also critical, as it can drives you to unexpected behaviours.

There is much more to cover within RxJS and I will do it in further articles.

I hope you enjoyed this article! ?

Samuel Teboul

Read more posts by this author.

If you read this far, tweet to the author to show them you care.

Tweet a thanks

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

Trending Guides

JavaScript split()

Span HTML