**Promise vs Observable**

Promise

Not cancellable

Return Single Value (Asynchronous)

Its eager return, the executor function is called as soon as the promise is created.

const promise = new Promise(resolve => {

resolve(1);

resolve(2);

resolve(3);

});

promise.then(result => console.log(result));

Would return only 1

**Observable**

Cancellable (Can unsubscribe)

Return multiple values (Asynchronous)

Lazy return, the subscriber function is only called when a client subscribes to the observable

const observable = new Observable(observer => {

observer.next(1);

observer.next(2);

observer.next(3);

});

observable.subscribe(result => console.log(result));

Would return 1, 2 and 3

**map**

Apply projection with each value from source.

const source = from([1, 2, 3, 4, 5]);

//add 10 to each value

const example = source.pipe(map(val => val + 10));

**mergeMap Vs switchMap**

**mergeMap** will have multiple active inner subscriptions, we use this cases of writing to database, and it does not care about the order of execution of each subscription.

You might be using take(n), to limit the subscription to avoid memos leaks in long lived subscriptions.

**switchMap** will have only one active inner subscription, as and when a new subscription is added, the old one get cancelled, this can be used in scenarios like type aheads, cos as and when anew value comes old value is a replacement of new, so old subscription is not required.

**take, takeUntil, takeWhile and takeLatest**

**take** allows to return n number of values from the source

Const source = of(1, 2, 3, 4, 5);

const example = source.pipe(take(1));

const subscribe = example.subscribe(val => console.log(val));

-> Output 1

**takeUntil** allows to take values from source until specific time interval

const source = interval(1000);

const timer$ = timer(5000);

const example = source.pipe(takeUntil(timer$));

const subscribe = example.subscribe(val => console.log(val));

-> output: 0,1,2,3 // takes values until 5 seconds

**takeLast** takes the n latest value for the source

const source = of(1, 2, 3, 4, 5);

const example = source.pipe(takeLast(2));

const subscribe = example.subscribe(val => console.log(val));

-> Output 4, 5

**takewhile** will return until certain condition is true

**combineAll**

// emit every 1s, take 2

const source$ = interval(1000).pipe(take(2));

// map each emitted value from source to interval observable that takes 5 values

const example$ = source$.pipe(

map(val =>

interval(1000).pipe(

map(i => `Result (${val}): ${i}`),

take(5)

)

)

);

Combine the result of both the maps together and respond when ever a value emits from any of the sources

["Result (0): 0", "Result (1): 0"]

["Result (0): 1", "Result (1): 0"]

—— from 0 to 4 for inner subscription

**combineLatest**

This is used when we need to get a calculated value of two different events of sources, for example get the values from two click events and produce a result.

// elem refs

const redTotal = document.getElementById('red-total');

const blackTotal = document.getElementById('black-total');

const total = document.getElementById('total');

const addOneClick$ = id =>

fromEvent(document.getElementById(id), 'click').pipe(

// map every click to 1

mapTo(1),

// keep a running total

scan((acc, curr) => acc + curr, 0),

startWith(0) // the calculation start with value 0. The initial value will set to 0 before any event calculation

);

combineLatest(addOneClick$('red'), addOneClick$('black')).subscribe(

([red, black]: any) => {

redTotal.innerHTML = red;

blackTotal.innerHTML = black;

total.innerHTML = red + black;

}

);

Other scenarios is we make two back end calls and both return the data, then we do some combining the data once and send to the component, in case of ngrx effects class.

**concat**

Used when order of subscription is required,

import { of, concat } from 'rxjs';

concat(

of(1, 2, 3),

// subscribed after first completes

of(4, 5, 6),

// subscribed after second completes

of(7, 8, 9)

)

// log: 1, 2, 3, 4, 5, 6, 7, 8, 9

.subscribe(console.log);

**concatAll**

Delay till inner subscription / observables completes

// RxJS v6+

import { take, concatAll } from 'rxjs/operators';

import { interval, of } from 'rxjs/observable/interval';

const obs1 = interval(1000).pipe(take(5));

const obs2 = interval(500).pipe(take(2));

const obs3 = interval(2000).pipe(take(1));

//emit three observables

const source = of(obs1, obs2, obs3);

//subscribe to each inner observable in order when previous completes

const example = source.pipe(concatAll());

/\*

output: 0,1,2,3,4,0,1,0

How it works...

Subscribes to each inner observable and emit values, when complete subscribe to next

obs1: 0,1,2,3,4 (complete)

obs2: 0,1 (complete)

obs3: 0 (complete)

\*/

const subscribe = example.subscribe(val => console.log(val));

**concatMap**

concatMap won’t subscribe to the next observable until the first one completes, but in the case of mergeMap, it immediately subscribe to the inner observables

**merge**

Convert multiple observable to single observable

// RxJS v6+

import { mapTo } from 'rxjs/operators';

import { interval, merge } from 'rxjs';

//emit every 2.5 seconds

const first = interval(2500);

//emit every 2 seconds

const second = interval(2000);

//emit every 1.5 seconds

const third = interval(1500);

//emit every 1 second

const fourth = interval(1000);

//emit outputs from one observable

const example = merge(

first.pipe(mapTo('FIRST!')),

second.pipe(mapTo('SECOND!')),

third.pipe(mapTo('THIRD')),

fourth.pipe(mapTo('FOURTH'))

);

//output: "FOURTH", "THIRD", "SECOND!", "FOURTH", "FIRST!", "THIRD", "FOURTH"

const subscribe = example.subscribe(val => console.log(val));

**mergeAll**

Collect and subscribe to all observables

const myPromise = val =>

new Promise(resolve => setTimeout(() => resolve(`Result: ${val}`), 2000));

//emit 1,2,3

const source = of(1, 2, 3);

const example = source.pipe(

//map each value to promise

map(val => myPromise(val)),

//emit result from source

mergeAll()

);

/\*

output:

"Result: 1"

"Result: 2"

"Result: 3"

\*/

const subscribe = example.subscribe(val => console.log(val));

**withLatestFrom**

provide last value for another observable

//emit every 5s

const source = interval(5000);

//emit every 1s

const secondSource = interval(1000);

const example = source.pipe(

withLatestFrom(secondSource),

map(([first, second]) => {

return `First Source (5s): ${first} Second Source (1s): ${second}`;

})

);

/\*

"First Source (5s): 0 Second Source (1s): 4"

"First Source (5s): 1 Second Source (1s): 9"

"First Source (5s): 2 Second Source (1s): 14"

...

\*/

const subscribe = example.subscribe(val => console.log(val));

**toPromose()**

Convert an observable to promise type.

**BehaviorSubject**

Emits the current value to the new subscription, need to provide a initial value as well.

const subject = new BehaviorSubject(123);

// two new subscribers will get initial value => output: 123, 123

subject.subscribe(console.log);

subject.subscribe(console.log);

// two subscribers will get new value => output: 456, 456

subject.next(456);

// new subscriber will get latest value (456) => output: 456

subject.subscribe(console.log);

// all three subscribers will get new value => output: 789, 789, 789

subject.next(789);

**ReplaySubject**

Emits old value to the new subscription and future values.

**Subject**

const sub = new Subject();

sub.next(1);

sub.subscribe(console.log);

sub.next(2); // OUTPUT => 2

sub.subscribe(console.log);

sub.next(3); // OUTPUT => 3,3 (logged from both subscribers)