RxJs

It provides a way to process different kinds of streams with the same API

For example

1. Events firing by click of a button: infinite stream that is usually handled by click listener
2. An array: a finite stream that is handled by a for loop

# Pull vs Push

### Pull:

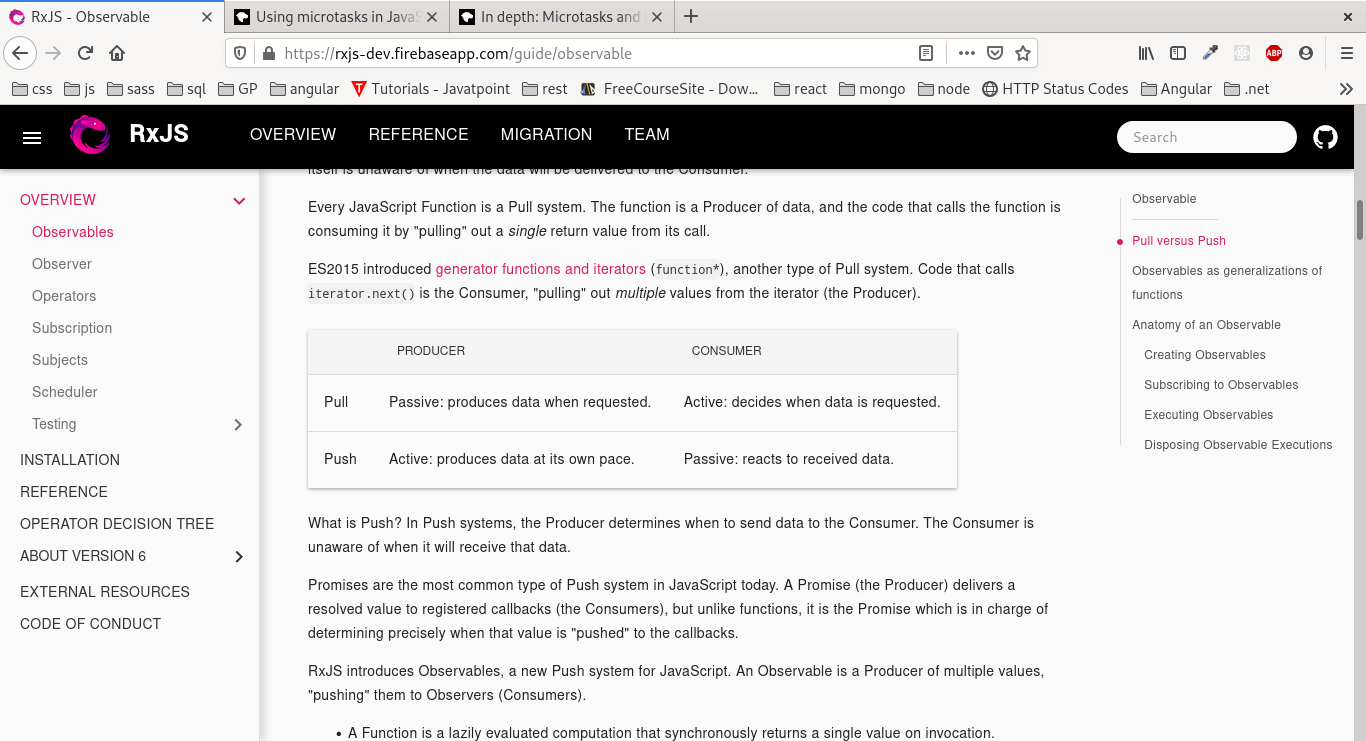
What is Pull? In Pull systems, the Consumer determines when it receives data from the data Producer. The Producer itself is unaware of when the data will be delivered to the Consumer.

Every JavaScript Function is a Pull system. The function is a Producer of data, and the code that calls the function is consuming it by "pulling" out a single return value from its call.

### Push:

What is Push? In Push systems, the Producer determines when to send data to the Consumer. The Consumer is unaware of when it will receive that data.

Promises are the most common type of Push system in JavaScript today. A Promise (the Producer) delivers a resolved value to registered callbacks (the Consumers), but unlike functions, it is the Promise which is in charge of determining precisely when that value is "pushed" to the callbacks.

RxJS introduces Observables, a new Push system for JavaScript. An Observable is a Producer of multiple values, "pushing" them to Observers (Consumers).

### Promise vs Observable

A Promise is a computation that may (or may not) eventually return a single value.

An Observable is a lazily evaluated computation that can synchronously or asynchronously return zero to (potentially) infinite values from the time it's invoked onwards.

We can unsubscribe from an observable but we cant stop a promise once it starts execution

# Observables

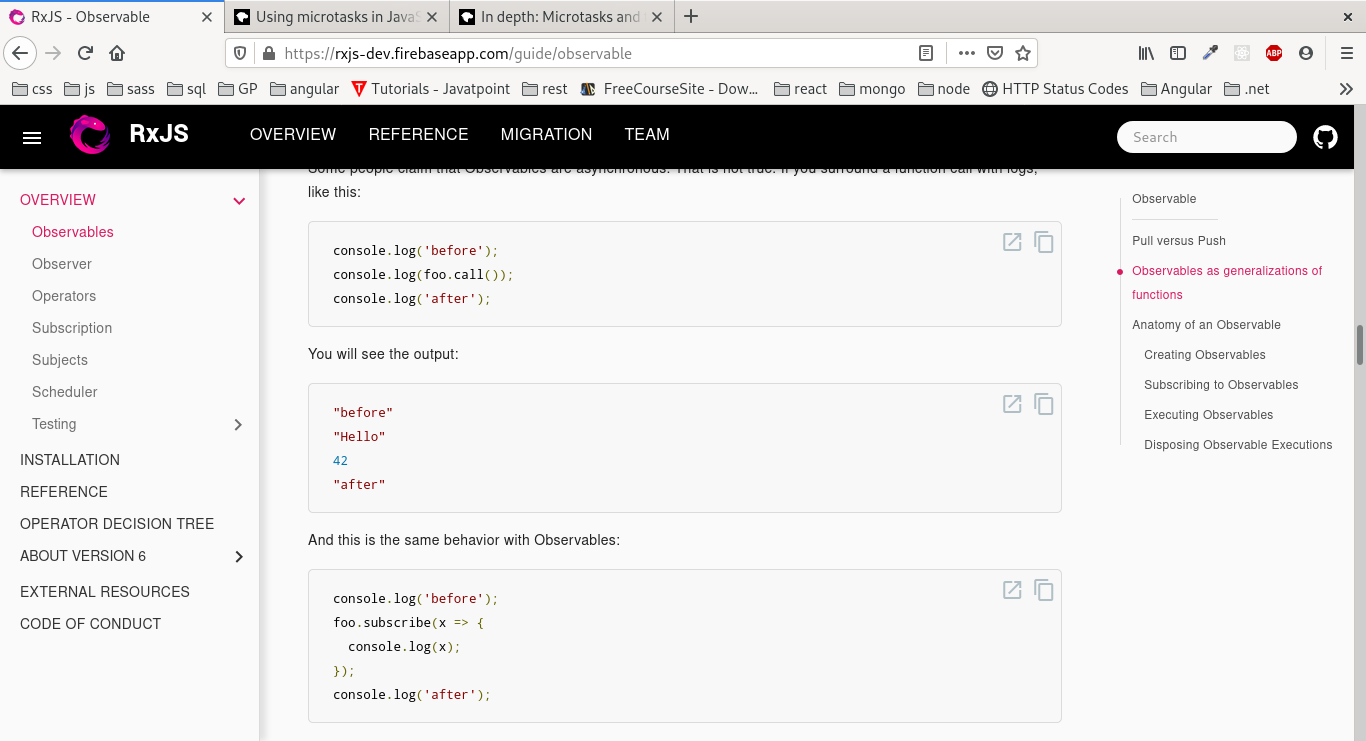
Contrary to popular claims, Observables are not like EventEmitters nor are they like Promises for multiple values. Observables may act like EventEmitters in some cases, namely when they are multicasted using RxJS Subjects, but usually they don't act like EventEmitters.

**Observables are like functions with zero arguments, but generalize those to allow multiple values.**

If you don't call the function, the console.log('Hello') won't happen. Also with Observables, if you don't "call" it (with subscribe), the console.log('Hello') won't happen. Plus, "calling" or "subscribing" is an isolated operation: two function calls trigger two separate side effects, and two Observable subscribes trigger two separate side effects.

As opposed to EventEmitters which share the side effects and have eager execution regardless of the existence of subscribers, Observables have no shared execution and are lazy.

Subscribing to an Observable is analogous to calling a Function.

**Observables are able to deliver values either synchronously or asynchronously.**

func.call() means "give me one value synchronously"

observable.subscribe() means "give me any amount of values, either synchronously or asynchronously"

**In an Observable Execution, zero to infinite Next notifications may be delivered. If either an Error or Complete notification is delivered, then nothing else can be delivered afterwards.**

# Observers

An Observer is a consumer of values delivered by an Observable. Observers are simply a set of callbacks, one for each type of notification delivered by the Observable: next, error, and complete.

Subscribers are also Observers because they implement the same interface

# How to subscribe to an observable:

1. Create a subscribe method and pass it as a parameter to Observable constructor
2. Use Observable.create (deprecated)
3. Using ‘of’ function

This takes a set of args and converts them to a stream so that each argument is fired separately

of("hello", 1, true, { book: allBooks[0] })

1. Using ‘from’ function

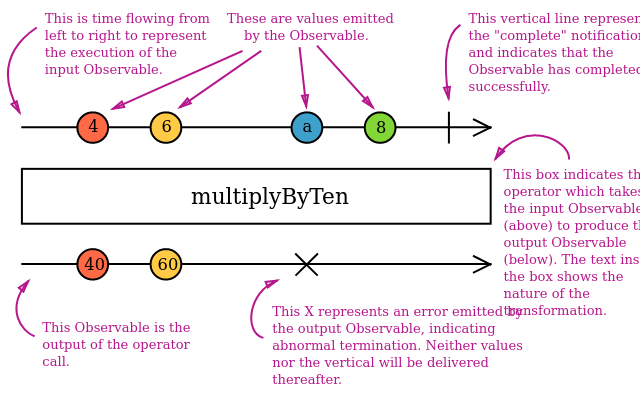
Takes an observable or a promise or an array(stream)

1. From ‘ajax’ request using ajax function

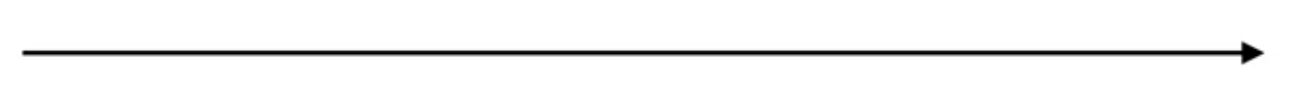
# Combining observables

1. concat(): will produce a new observable consist of all values from first observable followed by all values from second observable

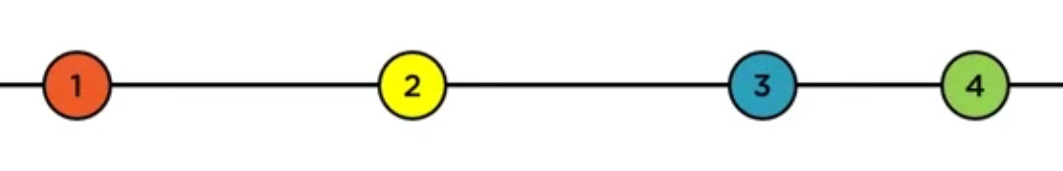
# Marble Diagram



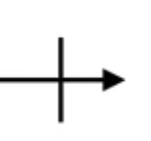
* The black horizontal arrow represents the source observable over time



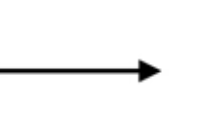
* The bubbles on the arrow represent the values which are produced over time by the observable



* The vertical line at the end of arrow represents the successful completion of observable



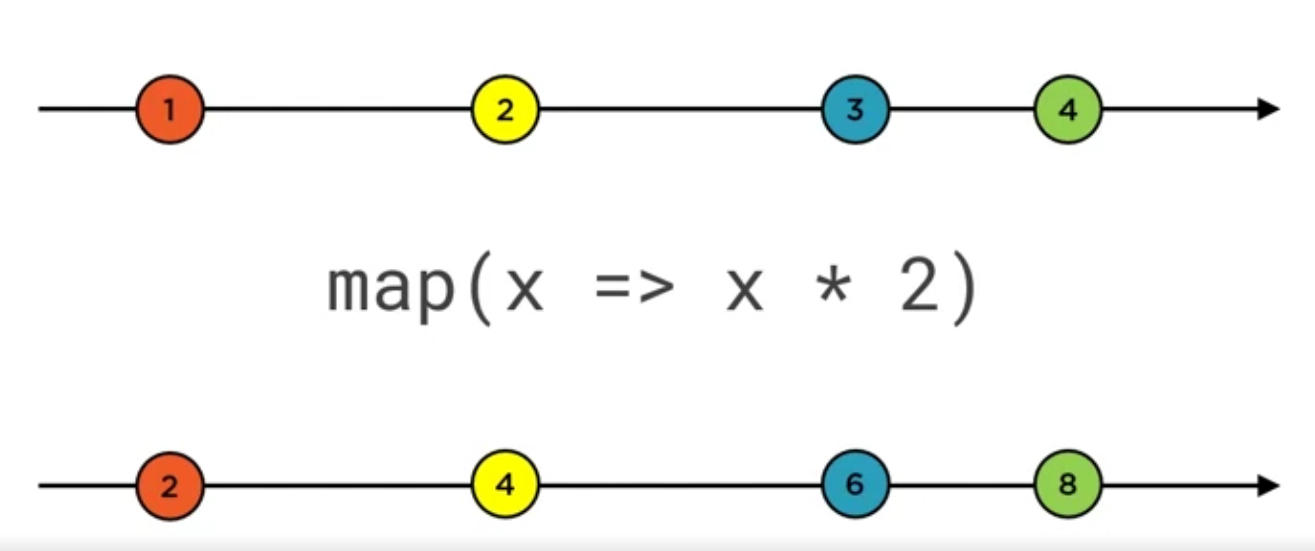
* Observable that will never complete



* Observable that failed to complete(has an error)



* The result of operator is represented in the bottom horizontal line



The colors associate the source value and result value of map operator

# Higher order observables

Observables most commonly emit ordinary values like strings and numbers, but surprisingly often, it is necessary to handle Observables of Observables, so-called higher-order Observables. For example, imagine you had an Observable emitting strings that were the URLs of files you wanted to see.

const fileObservable = urlObservable.pipe(

map(url => http.get(url)),

);

# Operators

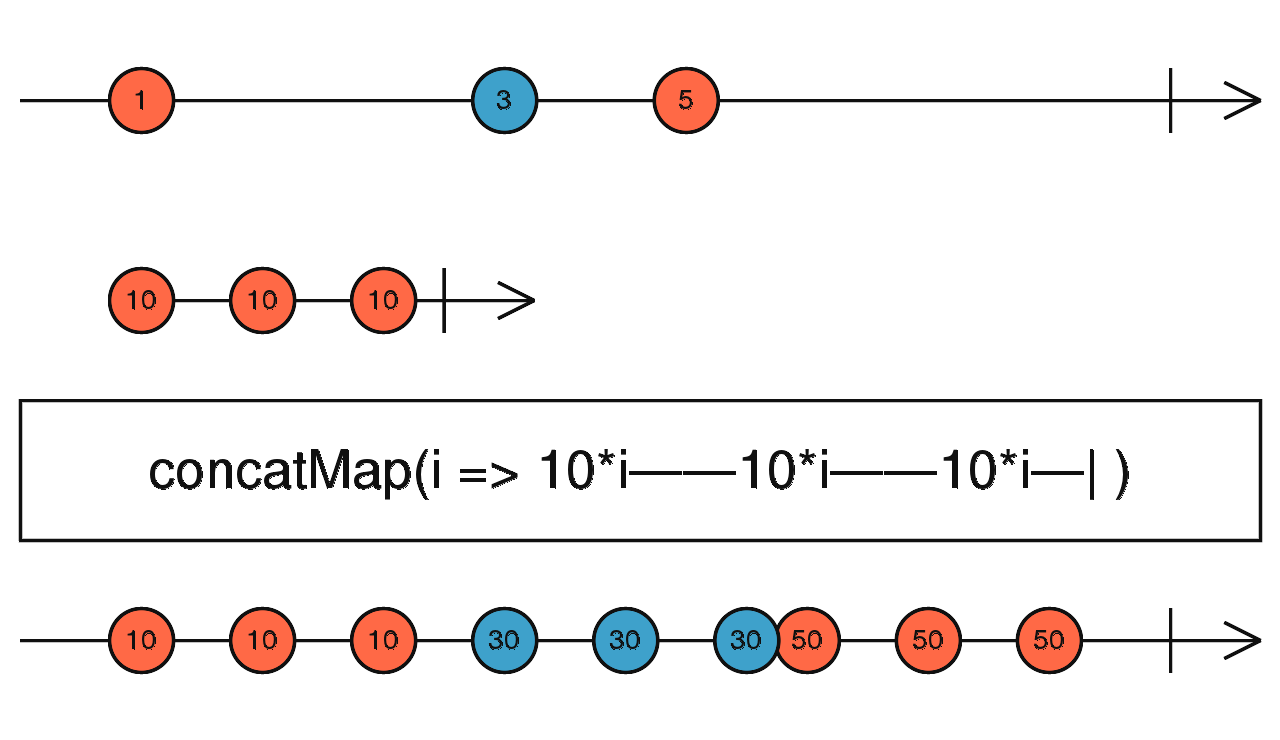
They add more functionalities to observables

It’s a function that returns a new function which returns an observable when called

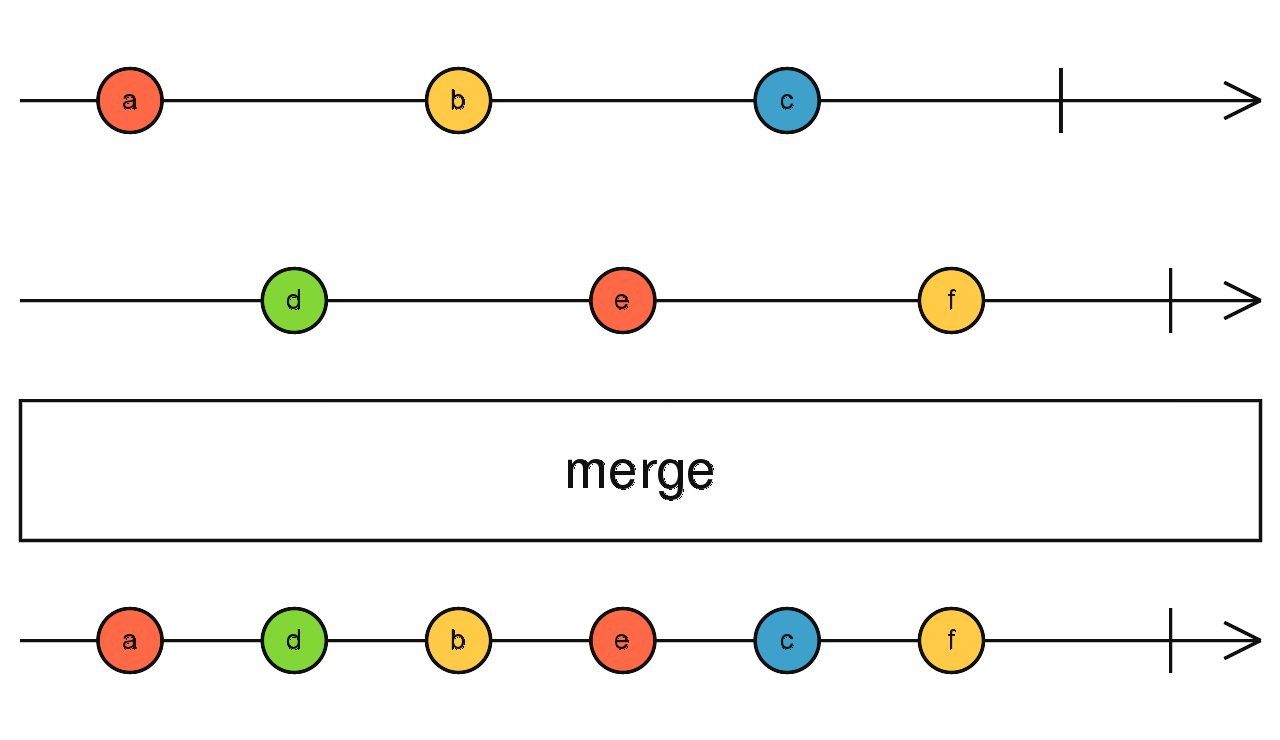
A Pipeable Operator is a function that takes an Observable as its input and returns another Observable. It is a pure operation: the previous Observable stays unmodified.

## Categories

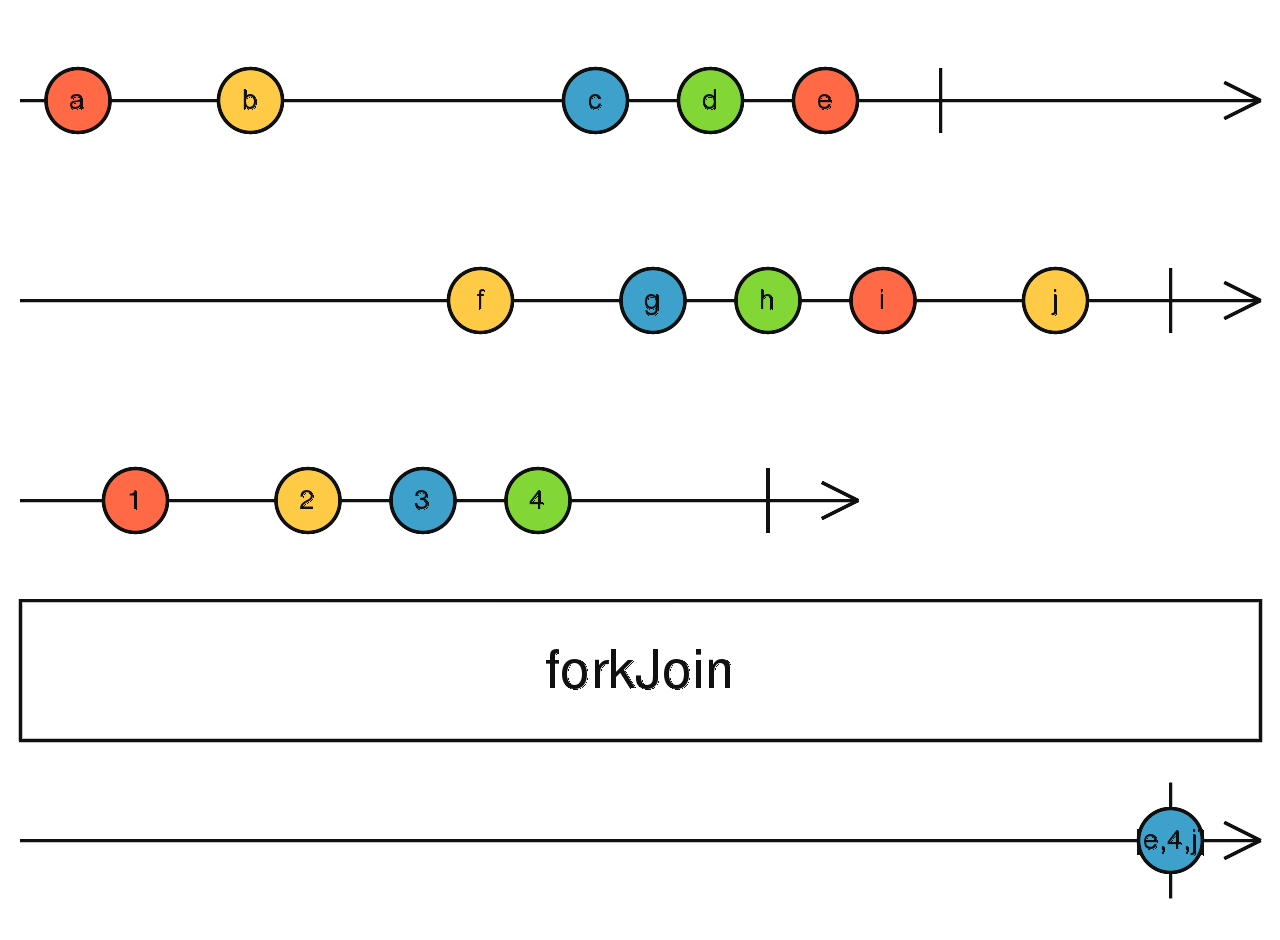
* Creation
  + ajax
  + from
  + of
  + interval
  + timer
* Transformation: the result will be different from the source
  + concatMap: Maps each value to an Observable, then flattens all of these inner Observables using [concatAll](https://rxjs-dev.firebaseapp.com/api/operators/concatAll).



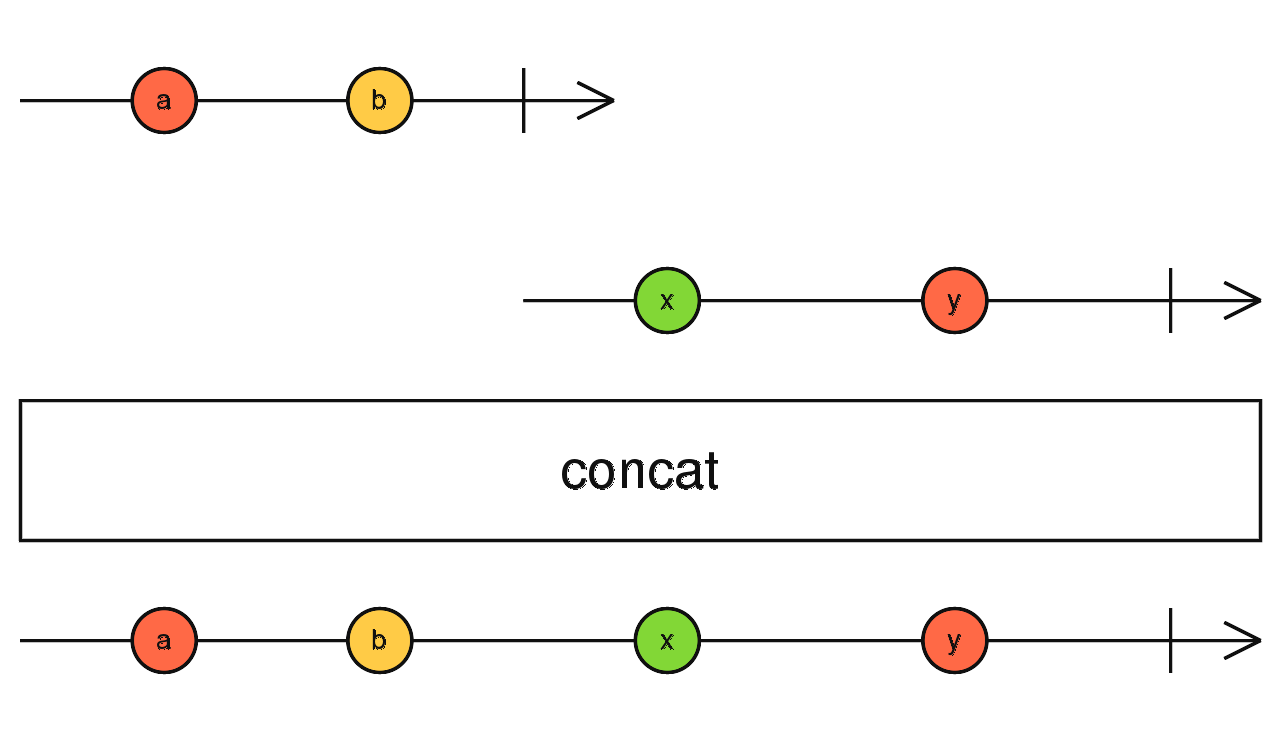
* + map:
* Filtering: produces a subset of the source stream
  + filter
  + skip
  + take
  + takeUntil
* Utility: how or when values are produced, without changing what values are produced
  + tap
  + delay
  + observeOn
* Conditional: similar o filtering
  + findIndex
  + find
* Aggregate: produce a single value as a result of all the stream for example: min, max, sum
* Join:These are Observable creation operators that also have join functionality -- emitting values of multiple source Observables.
  + Merge:Flattens multiple Observables together by blending their values into one Observable.



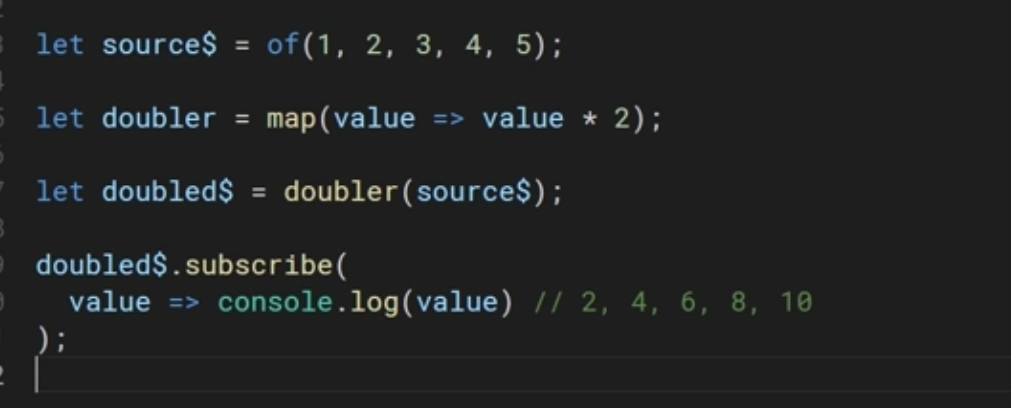
* + ForkJoin: Wait for Observables to complete and then combine last values they emitted.



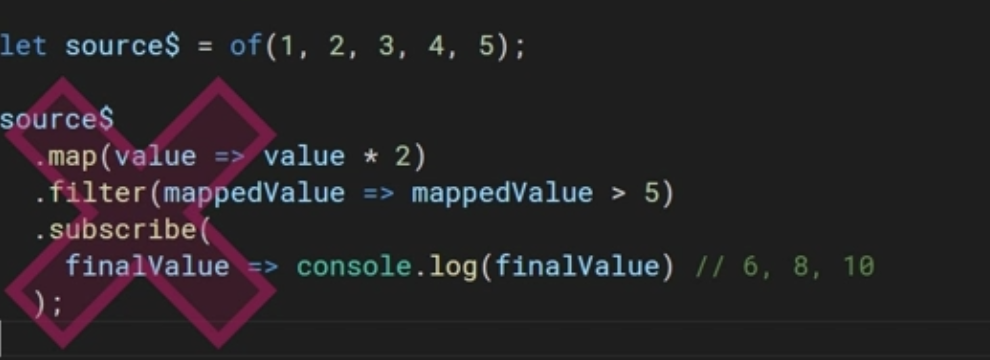
* + Concat: Concatenates multiple Observables together by sequentially emitting their values, one Observable after the other. concat joins multiple Observables together, by subscribing to them one at a time and merging their results into the output Observable.



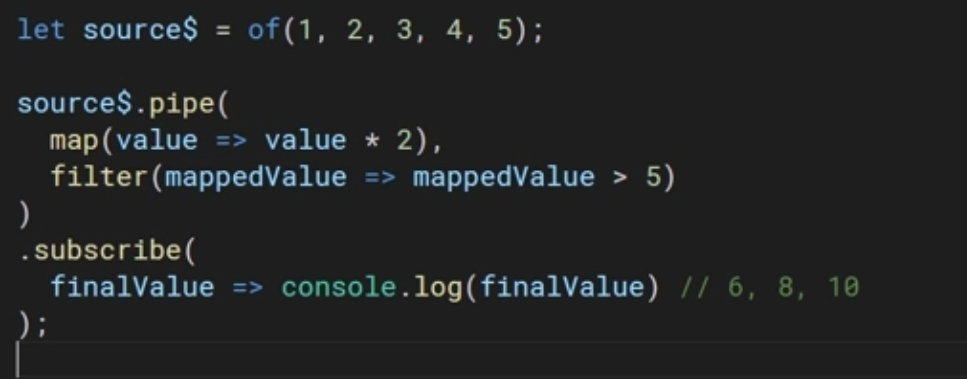
## Manual chanining:



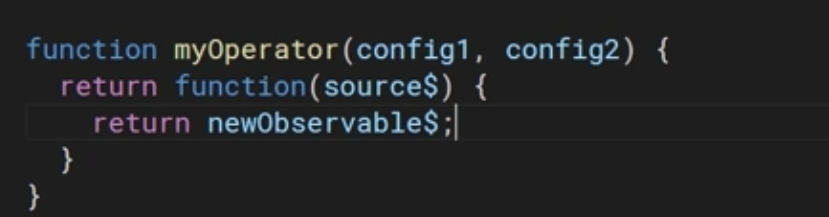
## Old method:



## New method after v5:



## Structure of operator:



It’s a function that could take configuration as parameters, and returns another function

The returned function takes the source observable and should return a new observable

## Custom operators

Why do we need it?

1. When we have a set of operators that we use together a lot, we can create a new one of the existing operators and use it
2. When we need new functionality that isn’t available in existing operators

## Notes:

Observable is only executed when there is a subscription to it, otherwise it wont fire.

# Subjects

They are both observables and observers

They are multicasted which means a single value is published to all subscribers at the same time before the next value is published

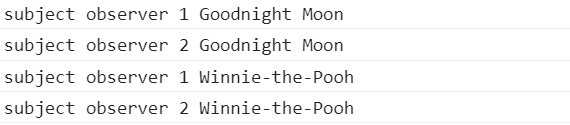
It contains an array of observers which it loops and pushes the single value to each of them

**Subjects are like EventEmitters: they maintain a registry of many listeners.**

For example in observables all values are sent to the first subscriber first.

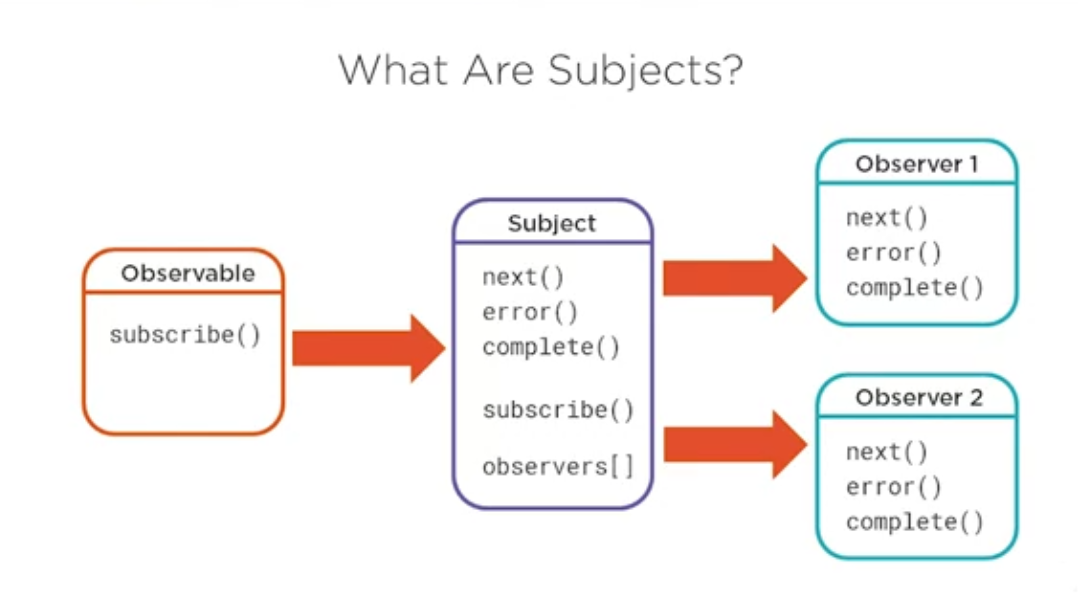


But in case of subjects each value is pushed to all observers



Internally to the Subject, subscribe does not invoke a new execution that delivers values. It simply registers the given Observer in a list of Observers, similarly to how addListener usually works in other libraries and languages.

## Multicasting



A subject can sit between an observable and observers to multicast each single value to all observers simultaneously

Note: in this case make sure you subscribe to the observable before the observables subscribe to the subject

Because the execution of subject begins whether or not it has subscribers

If an observer subscribed after the subject has completed, it will get the completion message or get the error which stopped the execution

## Cold vs Hot observables

### Cold (Unicast)

The value producer is created inside the observable

For example: if it produced a series of number, the array of numbers will be created inside the observable.

One observer per execution, which starts when subscribe is called

### Hot (multicast)

They wrap a value producer which exist outside the observable

Multiple observers receive the same value over the same time

Example: fromEvents

## Multicase operators

### muticast()

It takes a subject as a parameter and returns a connectable observable which has a method called connect() to begin the execution

### refCount()

triggers the execution when number of observers is > 0

Calling connect() manually and handling the Subscription is often cumbersome. Usually, we want to automatically connect when the first Observer arrives, and automatically cancel the shared execution when the last Observer unsubscribes.

When the number of subscribers increases from 0 to 1, it will call connect() for us, which starts the shared execution.

refCount makes the multicasted Observable automatically start executing when the first subscriber arrives, and stop executing when the last subscriber leaves.

### publish()

similar to multicast but doesn’t require to pass a subject and creates one behind the scenes

### share()

similar to refCount but will re-subscribe to the source observable if the refCount goes to zero then increases again

so it re-triggers the execution of the source

## Types of subjects

### Async subject:

Publishes only the last value of an observable  
publishLast() operator

### Behavior subject

Emit an initial seed value if a subscriber has subscribed before the source emits a value. If the source has already emitted a value then it will publish it.  
publishBehavior() operator

### Replay subject

Emit multiple number of values to all observers. If an observer subscribed after all values were published, it will replay all these values and re-publish them to the last observer.  
publishReplay() operator

# Schedulers

Controls when an observable begins execution and when notifications are delivered to its observers

## It consists of three components:

## A Scheduler is a data structure. It knows how to store and queue tasks based on priority or other criteria.

## A Scheduler is an execution context. It denotes where and when the task is executed (e.g. immediately, or in another callback mechanism such as setTimeout or process.nextTick, or the animation frame).

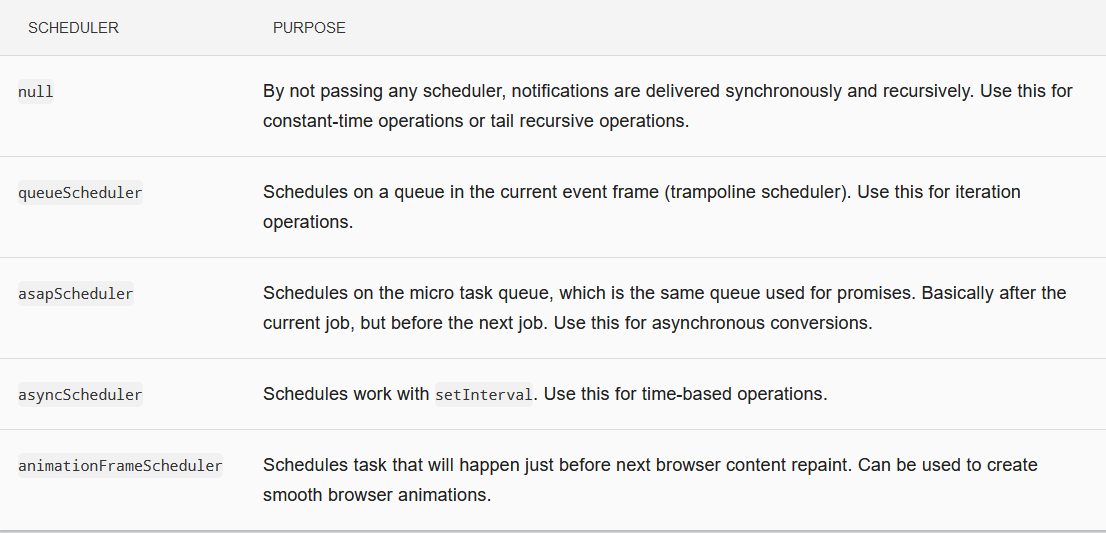
## A Scheduler has a (virtual) clock. It provides a notion of "time" by a getter method now() on the scheduler. Tasks being scheduled on a particular scheduler will adhere only to the time denoted by that clock.

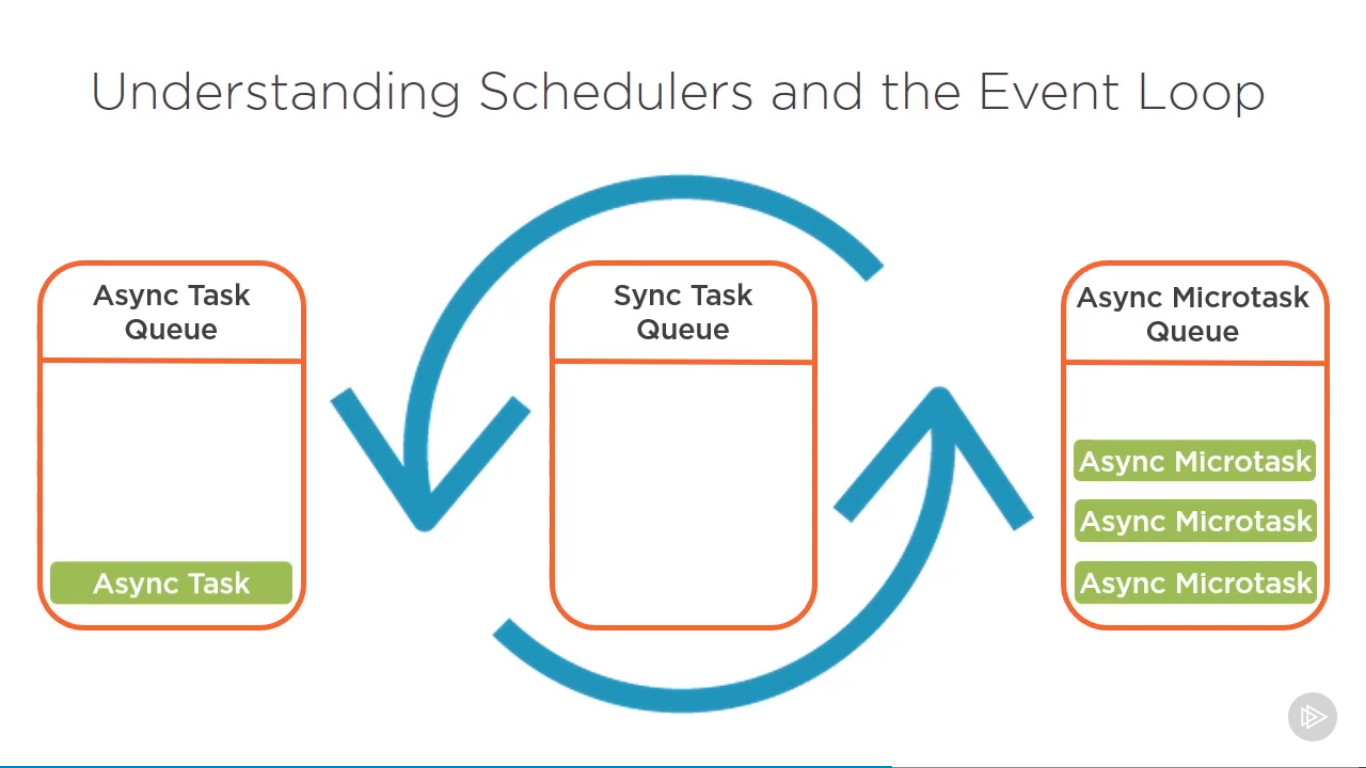
**A Scheduler lets you define in what execution context will an Observable deliver notifications to its Observer.**

You may have already used schedulers in your RxJS code without explicitly stating the type of schedulers to be used. This is because all Observable operators that deal with concurrency have optional schedulers. If you do not provide the scheduler, **RxJS will pick a default scheduler by using the principle of least concurrency**. This means that the scheduler which introduces the least amount of concurrency that satisfies the needs of the operator is chosen. For example, for operators returning an observable with a finite and small number of messages, RxJS uses no Scheduler, i.e. null or undefined. For operators returning a potentially large or infinite number of messages, queue Scheduler is used. For operators which use timers, async is used.

## Built in

* Queue scheduler: to execute observables synchronously
* Async scheduler: wont block js event loop while observable execute
* Asap scheduler: they are added to the micro task queue in event loop
* test scheduler: for unit testing observables



the async micro task queue has more priority than the async task queue and all of the micro tasks are added to the sync task queue at once, rather than waiting for next event loop to add the next microtask

# Testing

## Test schedulers

To test async code synchronously, in virtual time

only works with code that uses the asyncScheduler, doesnt work with asapScheduler or code that gets added to microTask queue

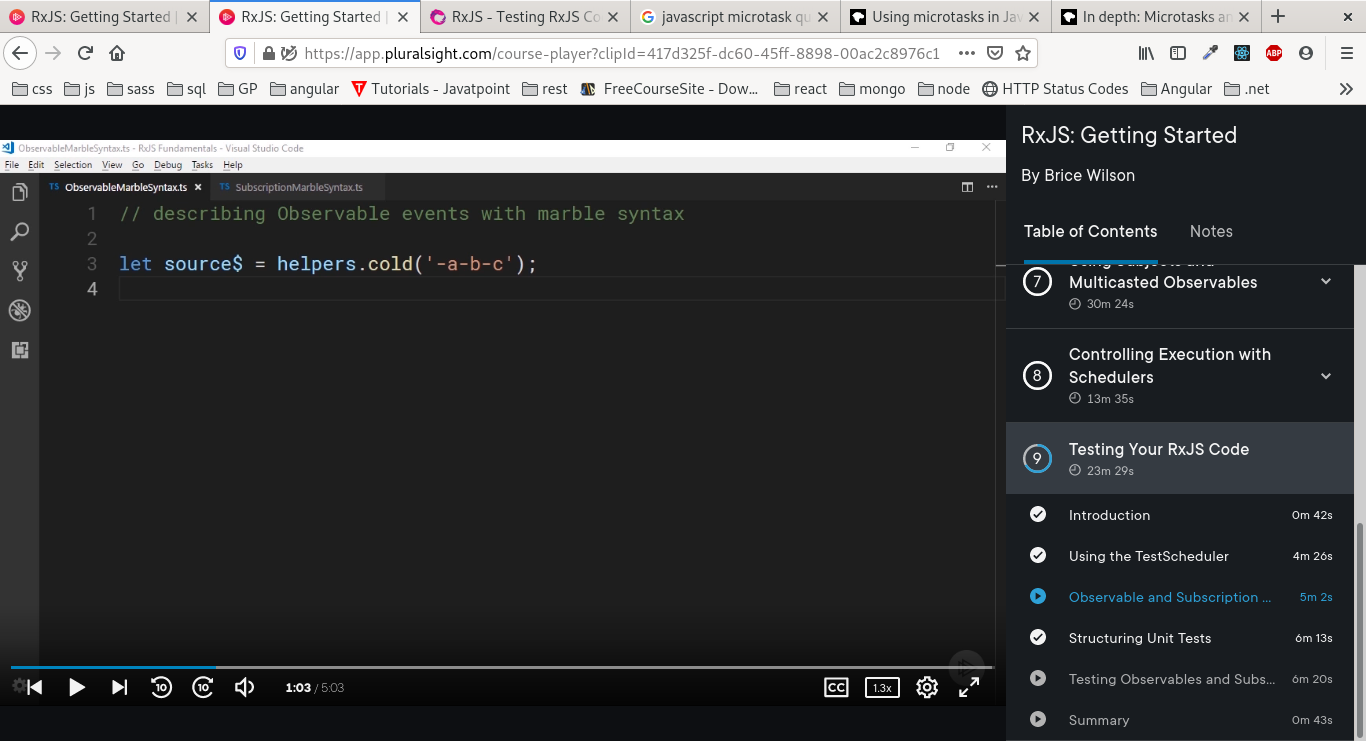
its constructor takes a function that gets used in equality tests; the function could be passed from the testing framework being used.

### run()

Accepts a callback function as a parameter that will perform the test

the recommended way to use it.

### Marble diagram for testing observable

This creates a new cold observable based on the pattern passed

1. each character in string represents a single frame in the diagram
2. a frame is the unit used to represent virtual time in a testScheduler
3. each frame represents 1 virtual millisecond (vms)
4. each dash (-) represents the passing of 1 vms
5. the values a,b,c will be produced with 1 vms between them

* ‘--a-4---c-8|’
  1. the observable will complete directly after 8 is produced
* ‘ --a-4 12ms c-8#’
  1. white spaces in beginning are ignored
  2. 12ms: instead of writing 12 dashes we can write the time between two spaces
  3. the hash-tag(#) an error will be produced
* ‘-a-^-b-(cde)---f|’
  1. ^: represents a subscription, only applies to hot observables

it means the subscriber wont receive the value ‘a’ because it hadn't subscribed yet

* 1. (cde): will occur synchronously in the same frame

### Marble diagram for testing subscriptions

* ‘^---!’: will begin then end after 3 ms
* ‘--^-’: will subscribe after 2ms and will never unsubscribe
* ‘^ 10ms !’