RXJS

* **Observable:** represents the idea of an invokable collection of future values or events.
* **Observer:** is a collection of callbacks that knows how to listen to values delivered by the Observable.
* **Subscription:** represents the execution of an Observable, is primarily useful for cancelling the execution.
* **Operators:** are pure functions that enable a functional programming style of dealing with collections with operations like [map](https://rxjs.dev/api/index/function/map), [filter](https://rxjs.dev/api/index/function/filter), [concat](https://rxjs.dev/api/index/function/concat), [reduce](https://rxjs.dev/api/index/function/reduce), etc.
* **Subject:** is equivalent to an EventEmitter, and the only way of multicasting a value or event to multiple Observers.
* **Schedulers:** are centralized dispatchers to control concurrency, allowing us to coordinate when computation happens on e.g. setTimeout or requestAnimationFrame or others.

Observables

THE STRUCTURE OF THE OBSERVABLE CLASS & CONSTRUCTOR

The subscribe? parameter in the constructor below is a function that is called when the Observable is subscribed to.

This function is given a Subscriber parameter, to which new values can be `next`ed, or an `error` method can be called to raise an error, or complete` can be called to notify of a successful completion.

So, whenever you create an observable, the constructor has an optional parameter that accept a function with the subscribe parameter.

export class Observable<T> implements Subscribable<T> {

    constructor(subscribe?: (this: Observable<T>, subscriber: Subscriber<T>) => TeardownLogic) {

       if (subscribe) {

      this.\_subscribe = subscribe;

      }

   }

}

How To Create an Observable

// Create new observable with the new keyword because it a class

      var myObservable = new Observable(

        // This is the function that will be called when an observer subscribe to this observable.

        /// the "subscriber" parameter is from the observable constructor (can be any name). it has next(value), complete(), error(value) method that you can use to send/emit values/data to the observar that has subscribed to this observable

      function (subscriber){

        //normally, you will perform operations here like http request and then...

        subscriber.next("one");  // send value/data to the subscriber (it should be json data from http request in real app)

        subscriber.complete();   // notify the subscriber of an operation done.

        // subscriber.error("there was an error");

      }

      );

// NOTE: When creating an observable, it is a good idea to wrap any code in subscribe function with try/catch block that will deliver an Error notification if it catches an exception:

//

const observable = new Observable(function subscribe(subscriber) {

  try {

    subscriber.next(1);

    subscriber.next(2);

    subscriber.next(3);

    subscriber.complete();

  } catch (err) {

    subscriber.error(err); // delivers an error if it caught one

  }

});

How To Subscribe to an Observable

To subscribe to an observable simple means to execute the function in the observable.

Note: Each different subscription to the observable will call the function in it to execute from scratch

  // To Execute the function in the above Observable, you need to subscribe to it using the subscribe() method of observable

      // So subscribing to observable simply means, executing the function(subscriber){} in the observable

      // subscribe() method constructor accept an object of 3 optional functions: next(), complete() error()

      myObservable.subscribe(

      {

        // next(nextValue) ==> "nextValue" represent the value emitted in the observable above with  subscriber.next("one") which is "one"

        // so you grab the value here and do something with it

        next(nextValue){

          console.log(nextValue);

        },

        // complete()  ==> get notified and do something when the observable is done. thats when the above observable calls subscriber.complete();

        complete(){

          console.log("done");

        },

        // error(errorValue) ==> "errorValue" represent the error emitted/send by an observable when it calls  subscriber.error("there was an error");

        error(errorValue){

          console.log(errorValue);

        }

      }

    );

  /\* RESULTS:

  one

  done

  \*/

  /\*

  NOTE: you can store the above subscription to a variable and use it to unsusbribe from the observable when needed like:

  The subscibe() method will return Subscription object, which represents the ongoing execution. Just call unsubscribe() to cancel the execution.

  var mysubscriber = myObservable.subscribe({other code cut off..........});

  mysubscriber.unSubscribe();

  \*/

  // You can also just grab the value in the next() method from the observable when subscribing to it like

  // below x represent the value "one" from the above observable next() method

  myObservable.subscribe(x =>{

    console.log(x)

  });

  // RESULT: one

============================================================================

RxJs Observable Operators:::

shareReplay() operator

The shareReplay() allows you to execute an observable once and share it emitted values to all of it subscribers/observers. For example, you may want to make a single http request to fetch a data and share the data in different component. But hei, if different component observe/subscribe to the same observable in the service, each subscription will trigger a separate http request. This is bad, you only want to make single http request and share the data between different components. That’s where shareReplay() operator comes in.

**Example. The problem of why you need shareReplay() operator**

  // inside app.component.ts

  export class AppComponent {

    constructor(private Http:HttpClient){

    }

    ngOnInit(): void {

      // create a get request that returns a data from server

      // remember the get() method returns observable of the data so

      // the "request" var below is of type Observable<any>

    var request: Observable<any> = this.Http.get("https://jsonplaceholder.typicode.com/todos/1");

    // First subscription

    // this will trigger the above get method to make a get request to the server

    request.subscribe(

      (data)=>{

        console.log(data);

      }

    );

        // Second subscription

    // this will trigger the above get method to make another get request to the server

    request.subscribe(

      (data)=>{

        console.log(data);

      }

    );

    /\* RESULTS FROM THE CONSOLE:

    { userId: 1, id: 1, title: "delectus aut autem", completed: false }

    { userId: 1, id: 1, title: "delectus aut autem", completed: false }

    \*/

    /\* BUT IF YOU CHECK THE BROWSER NETWORK TAB

    you can see from below image that two http request have been made. But I just want to make a single http request

    and share the return data betweeen all the subscribers.

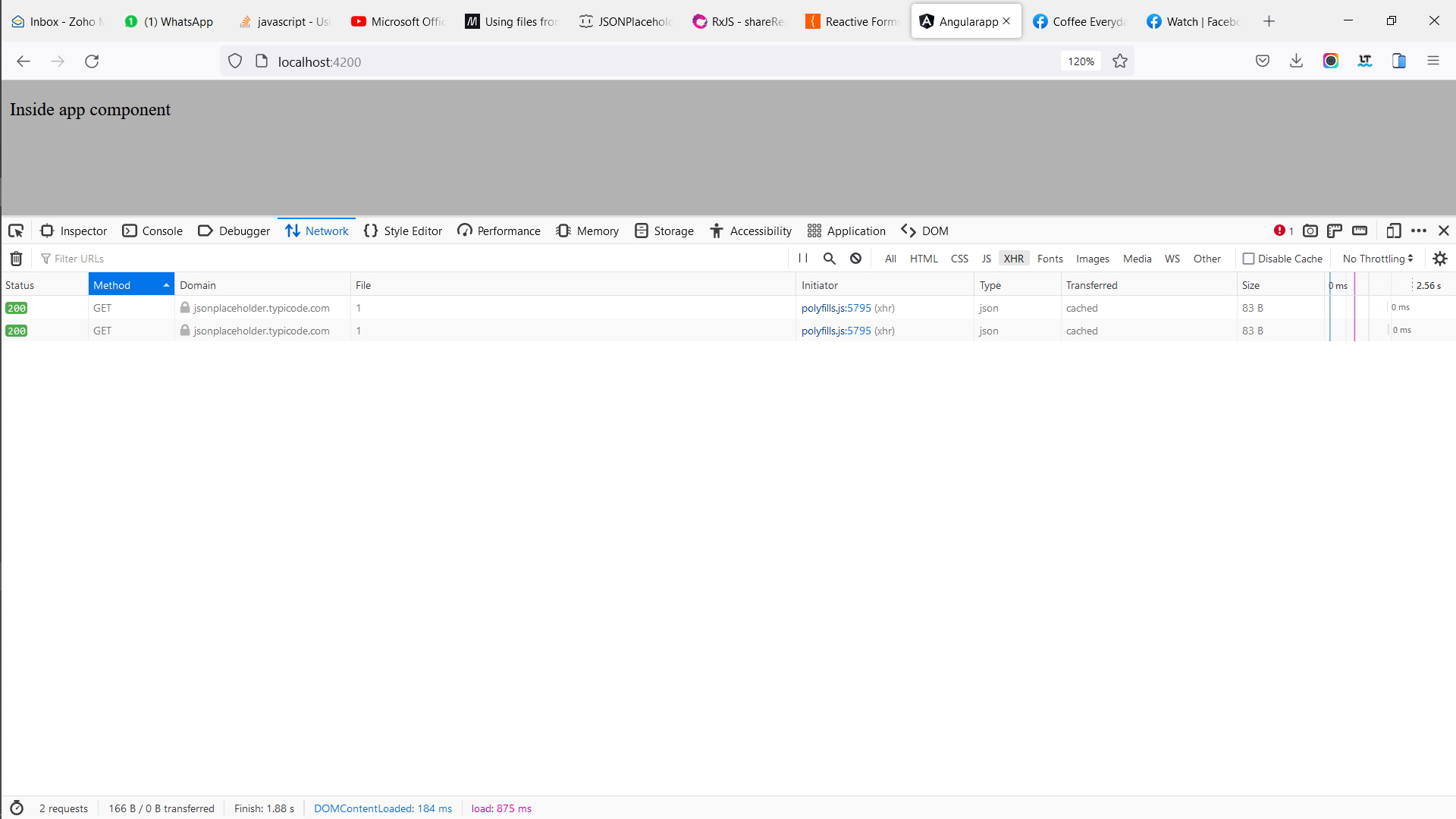
    Thats where shareReplay() operator comes in.

    It helps you to make single http request and share it with multiple subscribbers/observers

    \*/

}

}



**Example. How to use the shareReplay() operator**

To use it, simply include it in the pipe() operator

    // inside app.component.ts

    export class AppComponent {

      constructor(private Http:HttpClient){}

      ngOnInit(): void {

        // create a get request that returns a data from server

      var request: Observable<any> = this.Http.get("https://jsonplaceholder.typicode.com/todos/1").pipe(

        // meaning run this observable once and share it emitted value between all it subcribers/observers

        shareReplay()

      );

      // First subscription

      // this will trigger the above get method to make a get request to the server

      request.subscribe(

        (data)=>{

          console.log(data);

        }

      );

          // Second subscription

      // no http request will be made it will just grab the observable emitted values

      request.subscribe(

        (data)=>{

          console.log(data);

        }

      );

      /\* RESULTS FROM THE CONSOLE:

      { userId: 1, id: 1, title: "delectus aut autem", completed: false }

      { userId: 1, id: 1, title: "delectus aut autem", completed: false }

      \*/

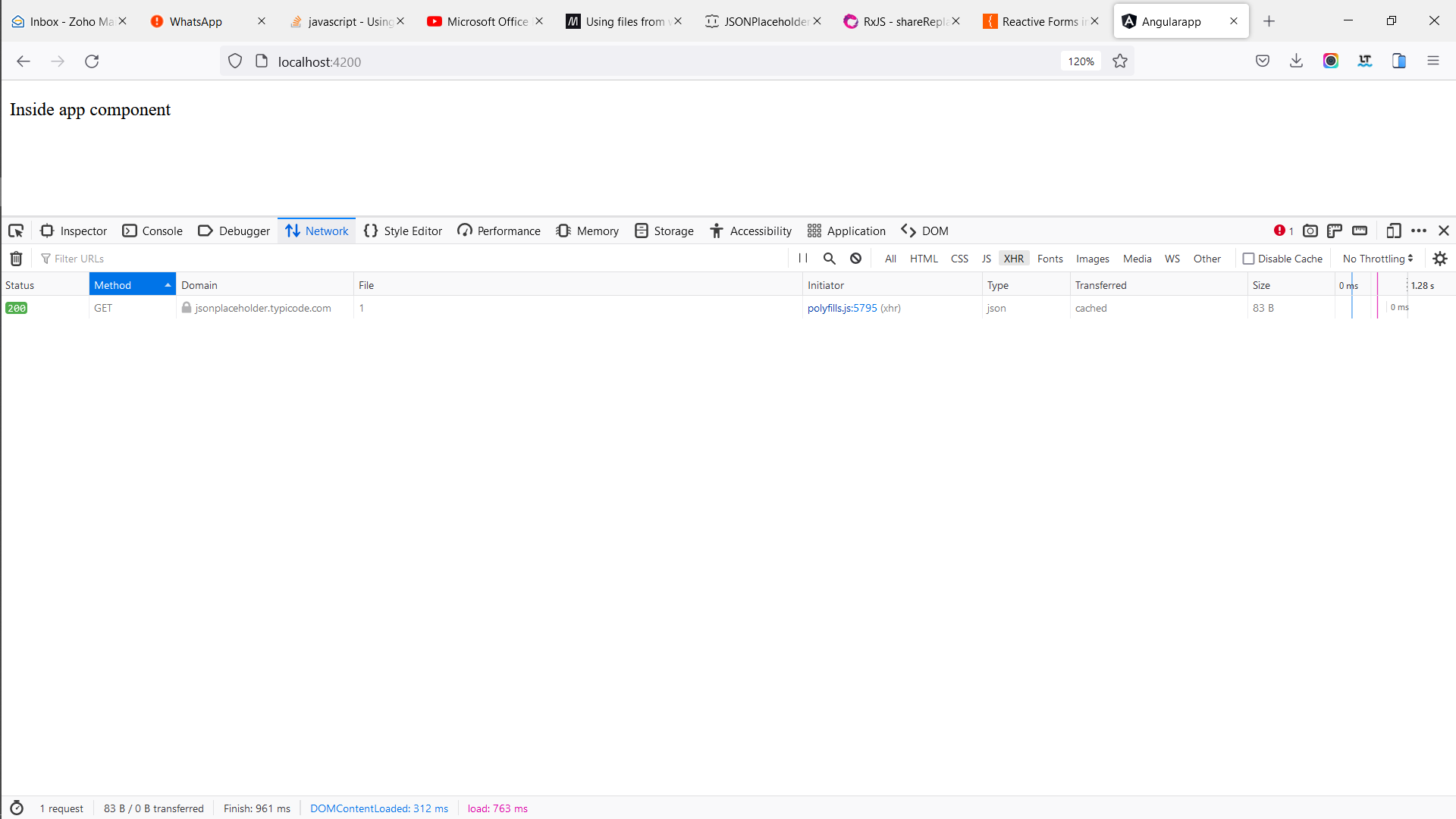
      /\* IF YOU CHECK THE BROWSER NETWORK TAB YOU CAN SEE ONLY ONE SINGL HTTP REQUEST EVEN THOUGH

      WE HAVE 2 SUBSCRIBERS FOR THE SAME OBSERVABLE

      \*/

  }

  }



tap() operator

The tap() operator returns an observable which is the same as the emitted value of the observable that the tap() operator is being used on. It mainly used for logging the emitted value of the observable to the console or other things that won’t affect the observable emitted value.

That’s because inside the tap() operator, if you modify the emitted value of the observable, it will not affect the original value emitted. For example. If the observable emit a value of 2 and you grab it inside the tap() operator and add 5 to it, the component listening to the observable will still receive the value of 2.

NOTE: The purpose of the tap() operator is mainly for debugging and also to do things that will not affect the observable emitted value.

NOTE: The tap() operator is the opposite of the map() operator. Map() can modify observable emitted values but tap() cannot.

**Tap()** operator object

The tap() operator takes in an object the same as observers. The object is next(val), error(val), complete() object emitted by the observable.

ngOnInit(): void {

        //create observables that emit 1 2 3 4 values

        var myObservable = new Observable(

          (subscriber)=>{

            subscriber.next(1);

            subscriber.next(2);

            subscriber.next(3);

            subscriber.next(4);

          }

        ).pipe(

         tap(

         {

           // object to grab each value the above "myObservable" emit

          next: (data)=>{

              console.log(data);

          },

           // object to grab the errorr above "myObservable" throws

          error: (err)=>{

          },

           // get notification of the above "myObservable" completion

          complete: ()=>{

          }

         }

         )

        );

        myObservable.subscribe();

        /\* RESULT

          1

          2

          3

          4

        \*/

      }

**How to use the tap() operator**

ngOnInit(): void {

        //create observables that emit 1 2 3 4 values

        var myObservable = new Observable(

          (subscriber)=>{

            subscriber.next(1);

            subscriber.next(2);

            subscriber.next(3);

            subscriber.next(4);

          }

        ).pipe(

         tap( data =>{

          // cast the emited value to number and add 2 to it

          // this will not affect the original value emitted by the observable

          // because tap() operator can't modify the emitted values by the observable

           (data as number) + 2;

         }

         )

        );

        // subscribe the observable

        myObservable.subscribe(data =>{console.log(data);} );

        /\* RESULT

        You still get the original value emitted by the observable and not the changes

        made in the above tap() operator

          1

          2

          3

          4

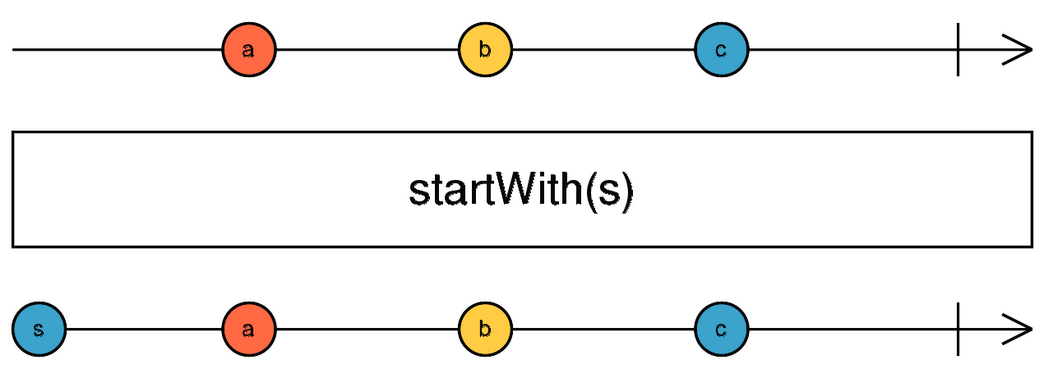
        \*/

      }

startWith() operator

The startWith(list\_of\_values) operator allows you to emit the values passed into it as an argument to an observer before emitting the main Observable values that the startWith() operator is used on.

Example below, the main observable is to emit a,b,c but the startWith(s) means the observable should emit the value “s” before a, b, c. So at the end, the observable will emit s,a,b,c



So, If you want an Observable to emit a specific sequence of items before it begins emitting the main items expected from it, apply the StartWith() operator to it.

**Example:**

ngOnInit(): void {

        //create observables that emit 1 2 3 4 values

        var myObservable = new Observable(

          (subscriber)=>{

            subscriber.next(3);

            subscriber.next(4);

            subscriber.next(5);

          }

        ).pipe(

          // First emit below values, then begin emitting the above observable values

          startWith(0,1,2),

        );

        // subscribe the observable

        myObservable.subscribe(data =>{console.log(data);} );

        /\* RESULT

          0

          1

          2

          3

          4

          5

        \*/

      }

============================================================================

Subject

Subject are observables but they just multicast values to different observers. But plain observables unicast values to observers

**Difference between subject and observables**

// inside app.component.ts

    export class AppComponent {

      constructor(){

      }

      ngOnInit(): void {

        // Create an observable

      var myObservble:Observable<number> = new Observable<number>(

        (subscription)=>{

          subscription.next(1);

          subscription.next(2);

        }

      );

      // first subscription

      myObservble.subscribe(

        {

          next:(data)=>{

            console.log(`first subscription value is: ${data}`);

          }

        }

      );

      // secon subscription

      myObservble.subscribe(

        {

          next:(data)=>{

            console.log(`second subscription value is: ${data}`);

          }

        }

      );

        // third subscription

      myObservble.subscribe(

        {

          next:(data)=>{

            console.log(`third subscription value is: ${data}`);

          }

        }

      );

      /\* RESULT:

      From below, you can see that the plain observable finish emiting/serving all it values 1 & 2 to the

      "first subscription". Then it moves on to the "second subscription" and started the execution of the observable from scratch and

      emited/served all of it values 1 & 2. Then it moves on to "third subscriptio" and started the execution of the observable from scratch and

      emited/served all of it values 1 & 2.  So each single subscription execute the observable from scratch and if the observable was to make http

      request, then it will make 3 request.

      first subscription value is:  1

      first subscription value is:  2

      second subscription value is: 1

      second subscription value is: 2

      third subscription value is:  1

      third subscription value is:  2

      \*/

      // Now Create a subject

      var mySubject:Subject<number> = new Subject<number>();

      // first subscription

      mySubject.subscribe({

        next:(data)=>{

          console.log(`first subject-subscription value is: ${data}`);

        }

      });

          // first subscription

          mySubject.subscribe({

            next:(data)=>{

              console.log(`second subject-subscription value is: ${data}`);

            }

          });

              // first subscription

      mySubject.subscribe({

        next:(data)=>{

          console.log(`third subject-subscription value is: ${data}`);

        }

      });

      // let your subject emit values to it subscribers

      mySubject.next(1);

      mySubject.next(2);

      /\* RESULT:

    From below, you can see that the observable executed only once, then emitted it first value to all of it subscribers,

    Then it moves on to emit it second value to all of it subscribers. If it was an http request, then it will make a single request

    So subject are muticast and observables are unicast.

    first subject-subscription value is: 1

    second subject-subscription value is: 1

    third subject-subscription value is: 1

    first subject-subscription value is: 2

    second subject-subscription value is: 2

    third subject-subscription value is: 2

      \*/

      }

  }