RXJS – tool that helps control data as it flows through time

Observables – wrapper of data. When observables change, their subscribers are notified of the change.

Const testObservable = Observable.create(observer => {

observer.next(data); //next() function emits data to the subscribers

observer.next(data2);

observer.complete(); complete() function closes the data stream

observer.next(data3); //not sent

})

The callback function notifies the subscriber of new data. When complete() is called, the observable is closed and no new data can be emitted (data3 is not sent).

Subscriptions and subscribers - Rxjs subscribe function takes a callback function which is called on the subscriber whenever the observable is changed. This allows the code to behave reactively.

Const testSubscription = testObservable.subscribe(data => {

console.log(data);

});

testSubscription is the subscription. testObservable is the observable. Whenever testObservable is changed, it will emit the data to testSubscription where it will be handled reactively (in this case, printed to the console).

**How to create Observables:**

of(value);

the of() helper function creates an observable wrapper of a raw value (string/int/etc)

from(value); //value = “hello”. This observable would pass each letter of the string “hello” individually.

from() takes an array, promise or iterable and emits each item from the observable individually.

fromEvent(document, “event”)

fromEvent() takes 2 arguments, the first is a DOM element (button/div/etc) and the second is the event you want to listen to (key press/click/etc)

interval(500) //emits a number after each 500ms interval.

interval() takes 1 argument, the time between data emissions, and emits a number each time that time interval is reached.

**Hot and cold observables**

Observables can either be hot (can have multiple subscriptions) or cold (can only have 1 subscription). Cold observables only calculate their values when a subscription is created. You can turn a cold observable hot by using cache.

The shareReplay(#) function caches the last value of an observable to allow multiple subscriptions access to this value.

**Subjects and behaviorSubjects:**

subjects are similar to hot observables but they can have new values pushed into them using the next() function. To add new values to a subscription, the subscription must be created before any values are added. Any subscription created after a value is pushed will not contain that value.

behaviorSubjects are similar to subjects although they store the last emitted value to be added to any new subscriptions. This behaves similarly to shareReplay(1). This means every subscription will always receive a value from the observable if one is available.

**Operators:**

Operators help you control the flow of data from your observables to your subscriptions. You can wrap operators in a pipe() function which allows for multiple sequential operators to be called. Order matters when placing operators in the pipe() function.

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

const modifiedObservable = sourceObservable.pipe(

map(n => Math.pow(n, 2)), //map()allows you to morph your data how you want

scan((acc, val) => acc + val), //scan() gives access to a cumulative value from all data emissions

filter(n => n > 5), //filter allows for filtering of your data

take(3) //take() defines how many data emissions will be sent before the observable completes.

)

This pipe will square the numbers, find the sum, filter any values not greater than 5 and only send the first 3 values.

**Other special use operators:**

tap()

The tap operator gives access to the data from within the pipe() function. You can access data this way from any point in the pipe() function.

**Backpressure operators**

debounceTime(500) //500ms delay

The debounce operator filters out all events and data emissions until they have stopped sending for the specified timeframe. This is useful when sending api calls after the user has stopped typing for a given delay.

throttleTime(500) //500ms delay

The throttle operator sends an event then waits the specified delay before sending the next event.

bufferCount(10) //buffers 10 events before emission

The buffer operator stores all events and emits them in an arraywhen they reach the given buffer count.

**Changing the subscription’s observables:**

switchMap(newObservable)

This function allows you to switch a subscription’s observable. Let’s say you have a users observable (full of all of your site’s users) and a user’s preference observable (full of a specific user’s preferences). You can swap the observable using nested observables, or you can use switchMap to swap the observables in the pipe() function.

combineLatest([observable1, observable2, …])

This function takes an array of observables, and emits the current state of each observable whenever data is emitted. Each observable must emit at least 1 value before a combined value is sent (as an array). Afterwards, if any observable emits a value, this function will return an array of all the current values of each observable.

merge(observable1, observable2, …)

merges the observables into a single data stream and emits values from these observables as they are sent over time.

**Handling errors:**

pipe(

catchError(err => of(‘something went wrong’)

when an error occurs, catchError helps you control what is sent.

retry(2)

if an error occurs, retry allows you to send another value in place of the value sent.

)

**Closing subscriptions:**

take(5)

sends the first 5 events and then closes the stream

takeWhile(n => n > 10)

sends events until the data does not meet the conditional statement, at which point the stream closes.

takeUntil(observable)

sends events until the observable (argument) sends an event at which point the original observable is completed.