25)RxJs Error Handling – The catch and replace error handling strategy

In this new part we will talk about rxjs error handling.Here we are going to see how errros behave in rxjs. We are going to talk about how to catch error’s.,how to handle errors, how to attempt to recover from errors and how to retry a observable if something goes wrong. To see how rxjs errors work. Lets trigger a error.

in home component we are making request to, /api/courses route. go to backend, server.ts file. here we can see routes. We change this for /api/courses route-

export function getAllCourses(req: Request, res: Response) {

/\*

const error = (Math.random() >= 0.5);

if (error) {

console.log("ERROR loading courses!");

res.status(500).json({message: 'random error occurred.'});

}

else {

\*/

setTimeout(() => {

// res.status(200).json({payload:Object.values(COURSES)});

res.status(500).json({ message: 'random error occurred.' });

}, 200);

// }

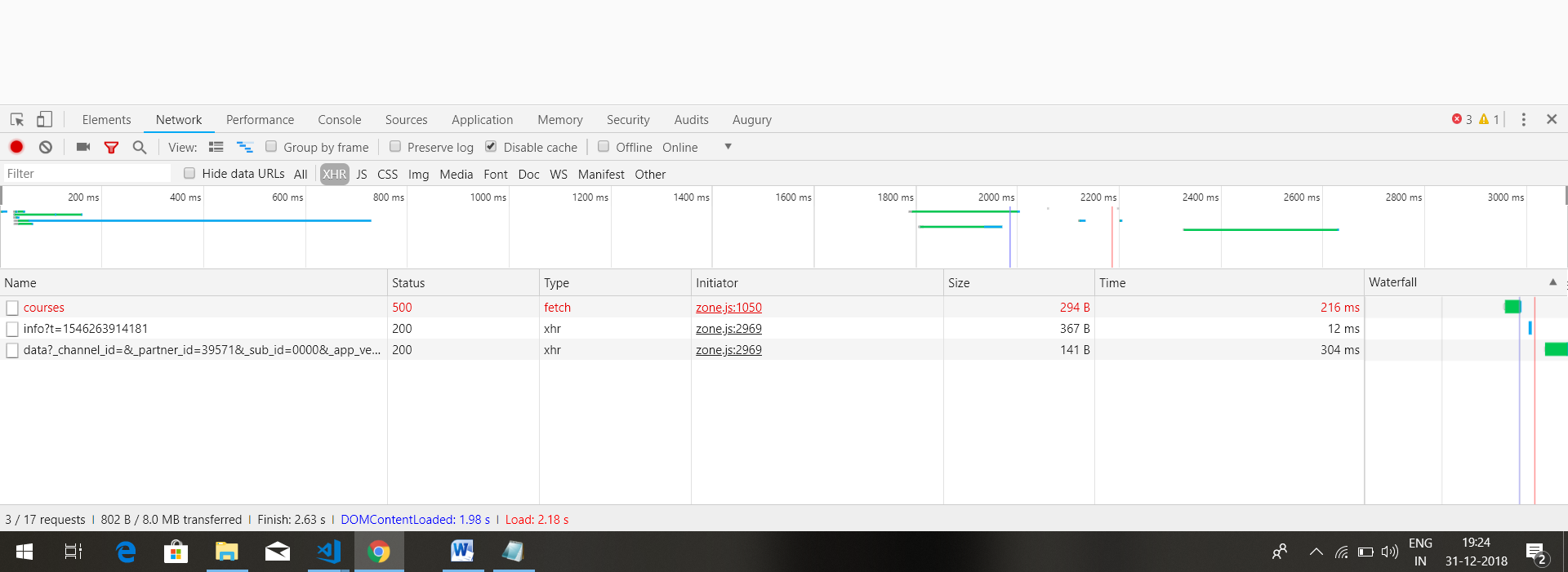
}

Later we will use above logic(commented out) to generate random errors to test our retry logic. But right now we want error to ocuur each time.

Now we are returning error instead of success response. Now restart server by running-

**npm run server**

now if we reload home component, we can see in network tab that we got an error and data is not populated(this data comes from success response).



Lets now try to catch this error and see how we can handle it. Multiple strategy are avalaible. We can either catch these error and try to recover from it by for example providing an alternate value, to the value we are getting from backened.

So instead of providing here the list of courses that receive from the backend , we can provide may be an alternative value that would still make sense in this concrete use case.

const http$ = createHttpObservable('/api/courses');

const courses$: Observable<Course[]> = http$

.pipe(

tap(() => console.log('HTTP Requested was executed')),

map(res => Object.values(res['payload'])),

shareReplay()

);

Here res[‘payload’] is list of courses that we get from backend.

We can also catch the error , log it to console and rethrow it to outer observable that is consuming this observable(consuming courses$ observable). and we can also retry the operation that just failed. So there we have 3 strategies that we are going to cover now.

ngOnInit() {

const http$ = createHttpObservable(`/api/courses`);

const courses$: Observable<Course[]> = http$

.pipe(

tap(() => console.log('HTTP request executed')),

map(res => Object.values(res['payload']) ),

shareReplay()

);

this.beginnerCourses$ = courses$

.pipe(

map(courses => courses

.filter(course => course.category === 'BEGINNER')));

this.advancedCourses$ = courses$

.pipe(

map(courses => courses

.filter(course => course.category === 'ADVANCED')));

}

First lets see how we can catch the error.we can catch the error by using **catchError** operator. This operator takes a function as argument. This function takes error as first argument and it is supposed to return an observable which is going to be used to continue the observable that error out(shown in blue color, this observable has errored out). So as we know according to observable contract , this observable(courses$) –

const courses$: Observable<Course[]> = http$

.pipe(

tap(() => console.log('HTTP Requested was executed')),

map(res => Object.values(res['payload'])),

shareReplay()

);

is going to emit value and then it is either going to complete or error out. Because this observable (which is shown above) has errored out with error on backend, this observable will not emit any further values. Then what is going to happen is catchError is going to catch error and it’s going to trigger function that is passed to it. The goal of this function is to provide an alternative observable that user of courses$(in this case our component is using this observable)can use in replacement of original observable that just failed. So output of this function is alternative error observable, that is only going to be consumed by component if http observable errors out. An alternative is that this function throws an error again and then courses$ observable is going to be error out just like http observable.

lets try first strategy, where we will try to recover from error by providing alternate value to component. For that we need to return observable in catchError.So let’s use **of** operator to return an observable that emits a single value.

Creates an Observable that emits some values you specify as arguments, immediately one after the other, and then emits a complete notification.

<https://reactivex.io/rxjs/class/es6/Observable.js~Observable.html#static-method-of>

this values needs to be array of courses. We could return empty array here. So empty list of corses will be displayed. In order to show this works.lets provide here some data. in sever folder go to , db-data.ts file. here copy one course object from array and pass it to array that we used in **of** operator.

ngOnInit() {

const http$ = createHttpObservable('/api/courses');

const courses$: Observable<Course[]> = http$

.pipe(

tap(() => console.log('HTTP Requested was executed')),

map(res => Object.values(res['payload'])),

shareReplay(),

catchError(error => of([

{

id: 0,

description: "RxJs In Practice Course",

iconUrl: 'https://s3-us-west-1.amazonaws.com/angular-university/course-images/rxjs-in-practice-course.png',

courseListIcon: 'https://angular-academy.s3.amazonaws.com/main-logo/main-page-logo-small-hat.png',

longDescription: "Understand the RxJs Observable pattern, learn the RxJs Operators via practical examples",

category: 'BEGINNER',

lessonsCount: 10

}

])

)

);

this.beginnerCourses$ = courses$

.pipe(

map((courses: Course[]) => courses.filter(course => course.category === 'BEGINNER'))

);

this.advancedCourses$ = courses$

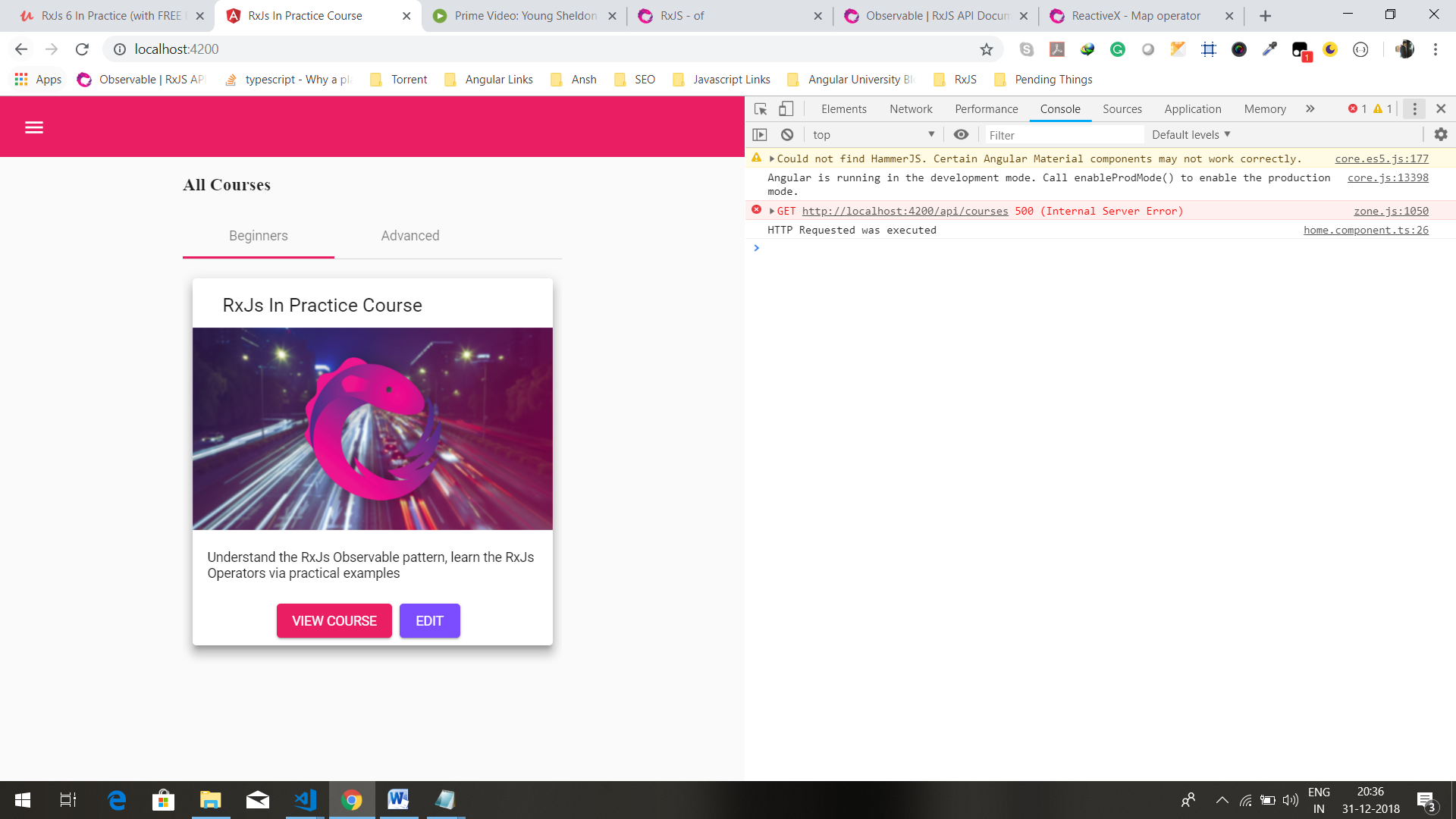
.pipe(

map((courses: Course[]) => courses.filter(course => course.category === 'ADVANCED'))

);

}

Output-



Now we can se that we have only one course. Also in network tab you can see that your request has failed. But instead of having empty screen, we can see that the alternative error observable has kicked in and then catchError operator subscribe to alternative error observable. the error observable was subscribed to and it’s value started showing up in course’s stream. Note that observable that we return in catchError operator does need to be built using **of** operator, it can potentially any observable that for example fetches data from an offline database when network is down. When this recovery observable that gets created by these catchError function, erros out, then the outer observable ,course$ will also error out.in same way when this recovery observable emits its first value and completes(because we used **of** operator), outer observable(courses$) also completes.

So this is a replacement observable replaces the error out observable. this error handling strategy that we just covered is **recovery observable error handling strategy**. Let’s now look at another strategies. we will now cover how to handle the error and then rethrow it and we will also see how to retry the failed observable.

Now all is extra

Code-

Util.ts-

import { Observable } from 'rxjs';

export function createHttpObservable(url: string) {

return Observable.create(observer => {

const controller = new AbortController();

const signal = controller.signal;

fetch(url, {signal})

.then(response => {

console.log('first then block');

// if (response.ok) {

// return response.json();

// } else {

// observer.error(`Request failed with status code: ${response.status}` );

// }

return response.json();

})

.then(body => {

console.log('second then block');

observer.next(body);

observer.complete();

})

.catch(err => {

console.log('control is in catch block of observable');

observer.error(err);

});

return () => controller.abort();

});

}

Home.component.ts-

ngOnInit() {

const http$ = createHttpObservable(`/api/courses`);

const courses$: Observable<Course[]> = http$

.pipe(

tap(() => console.log('HTTP request executed')),

map(res => Object.values(res['payload']) ),

shareReplay(),

catchError(err => of([{

id: 0,

description: "RxJs In Practice Course",

iconUrl: 'https://s3-us-west-1.amazonaws.com/angular-university/course-images/rxjs-in-practice-course.png',

courseListIcon: 'https://angular-academy.s3.amazonaws.com/main-logo/main-page-logo-small-hat.png',

longDescription: "Understand the RxJs Observable pattern, learn the RxJs Operators via practical examples",

category: 'BEGINNER',

lessonsCount: 10

}]))

);

this.beginnerCourses$ = courses$

.pipe(

map(courses => courses

.filter(course => course.category === 'BEGINNER')));

this.advancedCourses$ = courses$

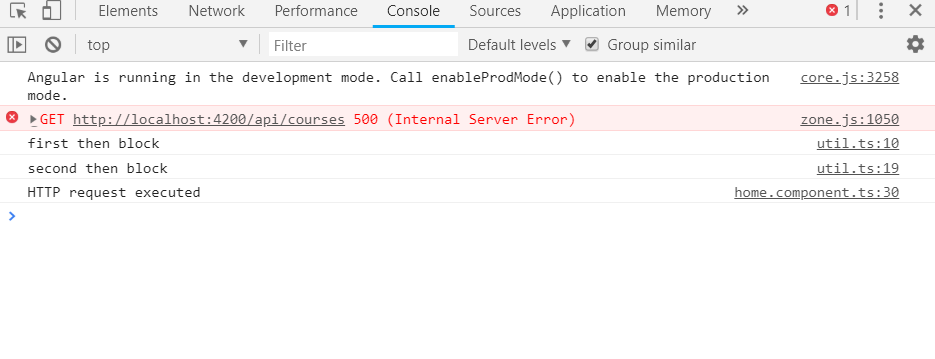
.pipe(

map(courses => courses

.filter(course => course.category === 'ADVANCED')));

}

In console we can see that we get error. This is to inform us that our request is failed. Code-



Here we have added console statements in then calls in http observable. if we run the program we will see that both console statements in http observables are printed and statement in tap operator is also printed. Errors occurs after that.it occusrs when map operators tries to read a value. this means there is no error in fetch api, error occurs when we try to read value in map operator.this is just a observation. Nothing was said about it in lecture.

Other strange thing that I have noticed is if I place some console.log statements in catchError block, they are executed 2 times.it is because of nature of shareReplay.

Perform this little experiment to know more-

ngOnInit() {

const source = interval(1000);

const source1 = source.pipe(

tap(() => console.log('tap1')),

shareReplay(),

tap(() => console.log('tap2')),

);

const sub1 = source1.subscribe((val) => console.log(`sub1 ${val}`));

setTimeout(() => {

const sub2 = source1.subscribe((val) => console.log(`sub2 ${val}`));

},5000);

}

Here first tap will be executed for each subscribe while tap before sharerReplay, will be executed once no matter how man subscriptions we have. So it means each subscribe creates a separate stream. But using shareReplay we can share those streams.

Here we are sharing those streams , so second subscribe will have values from 6. It is not starting its own stream(in that case it would have values starting from 0), it sharing stream which is already emitting values.

code-

Util.js-

import { Observable } from 'rxjs';

export function createHttpObservable(url: string) {

return Observable.create(observer => {

const controller = new AbortController();

const signal = controller.signal;

fetch(url, {signal})

.then(response => {

console.log('first then block');

return response.json();

// if (response.ok) {

// console.log('Inisde if block of promise');

// return response.json();

// } else {

// console.log('inisde else block of promise');

// observer.error(`Request failed with status code: ${response.status}` );

// }

})

.then(body => {

console.log('second then block');

observer.next(body);

observer.complete();

})

.catch(err => {

console.log('control is in catch block of observable');

// observer.error(err);

});

return () => controller.abort();

});

}

Home component.ts-

import { Component, OnInit } from "@angular/core";

import { Course } from "../model/course";

import { interval, Observable, of, timer, noop, throwError } from "rxjs";

import {

catchError,

delayWhen,

map,

retryWhen,

shareReplay,

tap

} from "rxjs/operators";

import { createHttpObservable } from "../common/util";

@Component({

selector: "home",

templateUrl: "./home.component.html",

styleUrls: ["./home.component.css"]

})

export class HomeComponent implements OnInit {

beginnerCourses$: Observable<Course[]>;

advancedCourses$: Observable<Course[]>;

constructor() {}

ngOnInit() {

const http$ = createHttpObservable(`/api/courses`);

const courses$: Observable<Course[]> = http$

.pipe(

tap(() => console.log('HTTP request executed')),

map(res => Object.values(res['payload'])),

shareReplay(),

catchError(err => {

console.log('inside catch error block');

console.log(`Error occured, ${err}`);

return of([{

id: 0,

description: "RxJs In Practice Course",

iconUrl: 'https://s3-us-west-1.amazonaws.com/angular-university/course-images/rxjs-in-practice-course.png',

courseListIcon: 'https://angular-academy.s3.amazonaws.com/main-logo/main-page-logo-small-hat.png',

longDescription: "Understand the RxJs Observable pattern, learn the RxJs Operators via practical examples",

category: 'BEGINNER',

lessonsCount: 10

}]);

}));

this.beginnerCourses$ = courses$

.pipe(

map(courses => courses

.filter(course => course.category === 'BEGINNER')));

this.advancedCourses$ = courses$

.pipe(

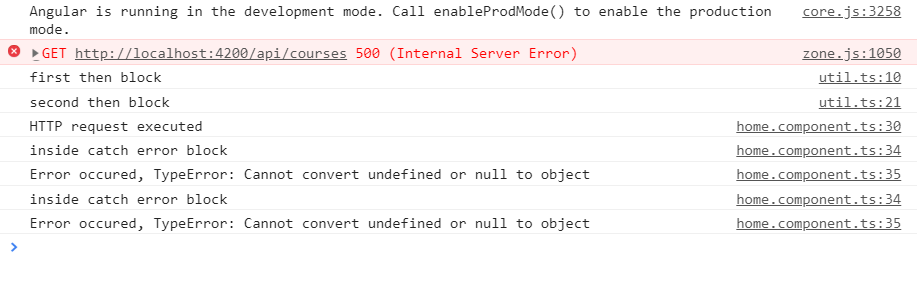
map(courses => courses

.filter(course => course.category === 'ADVANCED')));

}

}

Output-



Here you can see that statements in catchError are excuted 2 times. In next lecture we rethrow error, there also catchError block was executed 2 times.

Question-

1)catchError block is executed 2 times

<https://www.udemy.com/rxjs-course/learn/v4/questions/5964514>

26)The Catch and Rethrow RxJs Error Handling Strategy and the finalize Operator

Here we are going to cover a couple of commonly used error handling strategies for rxjs. We are going to cover catch and rethrow error handling strategies and we are going to see how to perform cleanup operations that are done whether the observable fails or completes. First we are going to improve here definition of our http Observable. if we go to http observable-

export function createHttpObservable(url: string) {

return Observable.create(observer => {

const controller = new AbortController();

const signal = controller.signal;

fetch(url, {signal})

.then(response => {

return response.json();

})

.then(body => {

observer.next(body);

observer.complete();

})

.catch(err => {

observer.error(err);

});

return () => controller.abort();

});

}

We can see that althrough we are supporting cancellaton , our error handling logic is not as good as it could be and this is due to behaviour of fetch api. So the fetch api is going to call here the backend and is going to return us here a response(in then block) then we have here catch block in case something fails. But the problem is catch block is only going to be triggered in case of fatal error. This means in case of network failure or in case of DNS error, something that browser cannot recover from. However there are other situations such as for ex, this case here when we are returning 500 internal server error from client, that we would also like to error out our observable.

so we are going to error out observable not only in case of fatal errors but we are also going to error out observable if the http response was correctly returned from the server but it is an error response. In order to check in which case we are, we are going to be using ok flag from response object.so our response is ok, if flag is true. If it is not true, then we are going to error out our observable. we call error method on observer and pass it a error message. Code-

Util.js-

import { Observable } from 'rxjs';

export function createHttpObservable(url: string) {

return Observable.create(observer => {

const controller = new AbortController();

const signal = controller.signal;

fetch(url, {signal})

.then(response => {

console.log('first then block');

if (response.ok) {

console.log('Inisde if block of promise');

return response.json();

} else {

console.log('inisde else block of promise');

observer.error(`Request failed with status code: ${response.status}` );

}

})

.then(body => {

console.log('second then block');

observer.next(body);

observer.complete();

})

.catch(err => {

console.log('control is in catch block of observable');

observer.error(err);

});

return () => controller.abort();

});

}

Here if status is not ok then we just error out the observable.

So now we have improved here the error handling of our http observer.

Question-

1)why catchError block was getting executed in last lecture

<https://www.udemy.com/rxjs-course/learn/v4/questions/5965154>

Lets now try out new error strategy which is going to be the catch and rethrow strategy. So instead of returning recovery observable ,we re going to handle error locally(in catchError block).

First lets log the error. this could have also been a call to a messaging service that displays here an error message to user on top of screen.. Either way error is going to be handled locally at the level of catchError block. But ide is giving us an error. This is because catchError operator needs to return an observable, the recovery observable. in this case we do not want to write an observable that tries to recover from this error situation, instead of that we want to create one observable that does not emit any value and instead it immediately errors our with exact same error that we got as argument in catchError operator. In order to produce such observable we have a utility method called **throwError**. This method will create one observable that will error out immediately with this error without emitting any value.

Home.component.ts-

import {Component, OnInit} from '@angular/core';

import { Course } from '../model/course';

import {interval, Observable, of, timer, throwError} from 'rxjs';

import {catchError, delayWhen, map, retryWhen, shareReplay, tap, finalize} from 'rxjs/operators';

import { createHttpObservable } from '../common/util';

@Component({

selector: 'home',

templateUrl: './home.component.html',

styleUrls: ['./home.component.css']

})

export class HomeComponent implements OnInit {

beginnerCourses$: Observable<Course[]>;

advancedCourses$: Observable<Course[]>;

constructor() {

}

ngOnInit() {

const http$ = createHttpObservable('/api/courses');

const courses$: Observable<Course[]> = http$

.pipe(

tap(() => console.log('HTTP Requested was executed')),

map(res => Object.values(res['payload'])),

shareReplay(),

catchError(err => {

console.log('Error Occured');

return throwError(err);

})

);

this.beginnerCourses$ = courses$

.pipe(

map((courses: Course[]) => courses.filter(course => course.category === 'BEGINNER'))

);

this.advancedCourses$ = courses$

.pipe(

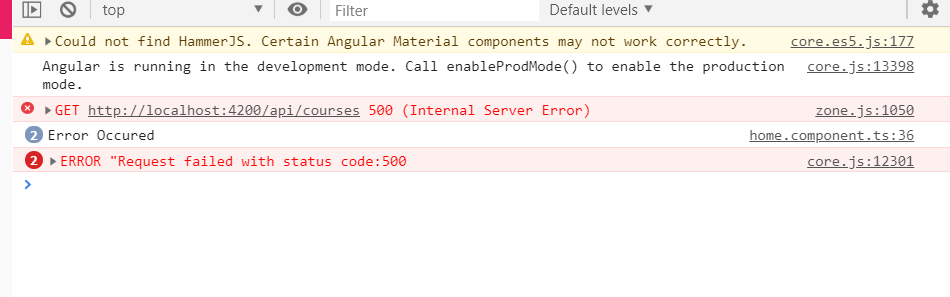
map((courses: Course[]) => courses.filter(course => course.category === 'ADVANCED'))

);

}

}

If we refresh the page, we can see in network tab that request failed. In console we can see that error is logged twice on console.



Here we can see that Catcherror block was executed twice. We can also see that error was indeed rethrown, because if we switch back to console, we can see that angular ‘s default error handler did indeed catch the exact same message that we logged there.

Here we can see that catchError block is executed twice, this is because we have 2 derived observables beginnerCourses and advancedCourses from courses$ observable.

So this shows that catch and rethrow strategy is working as expected, lets now talk about clean up logic. So lets say we have observable courses$ that might complete or that might fail, in both cases we want to do some kind of clean up operation, this could be to close network connection, realease an memory resource or some other common cleanup operation. In rxjs we can implement that type of clean up logic by using finalize operator. This operator is going to take a function that is going to be invoked in one of 2 cases(complete or error out). This function will be executed when courses$ observable completes or when it errors out. Lets try it out –

import { Component, OnInit } from "@angular/core";

import { Course } from "../model/course";

import { interval, Observable, of, timer, noop, throwError } from "rxjs";

import {

catchError,

delayWhen,

map,

retryWhen,

shareReplay,

tap,

finalize

} from "rxjs/operators";

import { createHttpObservable } from "../common/util";

@Component({

selector: "home",

templateUrl: "./home.component.html",

styleUrls: ["./home.component.css"]

})

export class HomeComponent implements OnInit {

beginnerCourses$: Observable<Course[]>;

advancedCourses$: Observable<Course[]>;

constructor() {}

ngOnInit() {

const http$ = createHttpObservable(`/api/courses`);

const courses$: Observable<Course[]> = http$

.pipe(

tap(() => console.log('HTTP request executed')),

map(res => Object.values(res['payload'])),

shareReplay(),

catchError(err => {

console.log('inside catch error block');

console.log(`Error occured, ${err}`);

return throwError(err);

}),

finalize(() => {

console.log('Finalized Executed');

})

);

this.beginnerCourses$ = courses$

.pipe(

map(courses => courses

.filter(course => course.category === 'BEGINNER')));

this.advancedCourses$ = courses$

.pipe(

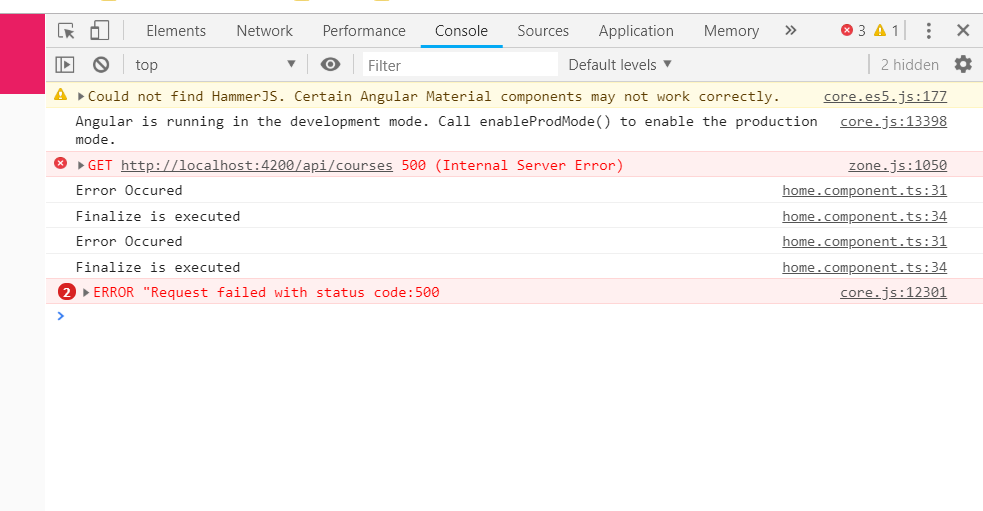
map(courses => courses

.filter(course => course.category === 'ADVANCED')));

}

}

Output-



As we can see that catchError and finalize is execuedc2 times. This because we subscribe to 2 observables that is defined by courses$ observable.

Question-

1)i want to execute catchError and finaloze block only once

<https://www.udemy.com/rxjs-course/learn/v4/questions/5965076>

If this is not the behaviour that you are looking for, if you want the error handler(catchError block) to get executed only once instead, then what we need to do is , we need to move catchError block up the observable chain. So instead of doing the the catchError here on output of shareReplay,that we are going to have this shared between 2 subscription(we are talking about output of shareReplay is going to be shared between 2 observables) instead we can take catchError block and move it up immediately after invoking the http observable. so in this way the whole observable chain is going to be bypassed and we will not get to this mapping operation(map operator) and we will not get here to shareRelay operator. Lets try this logic. Code-

const courses$: Observable<Course[]> = http$

.pipe(

catchError(err => {

console.log('Error Occured');

return throwError(err);

}),

tap(() => console.log('HTTP Requested was executed')),

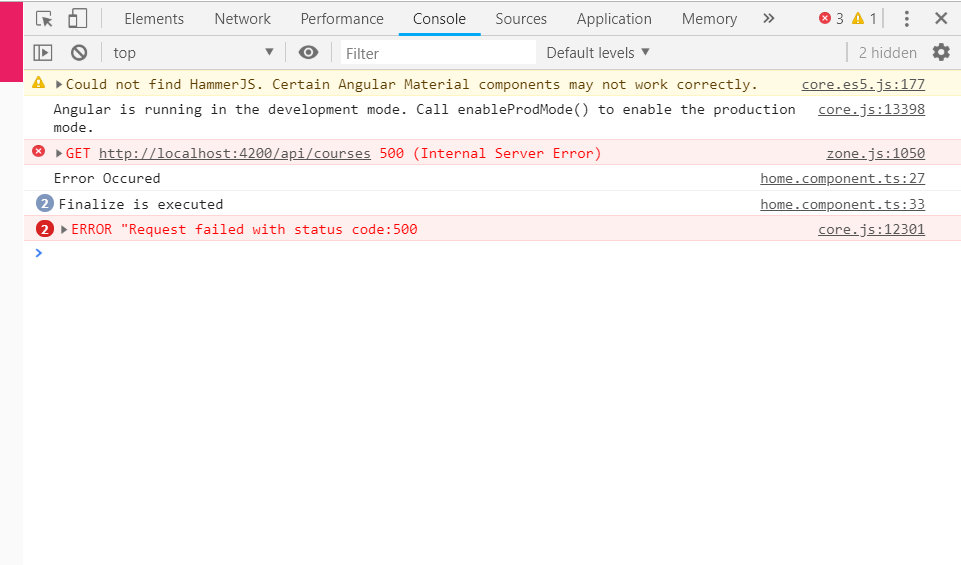
map(res => Object.values(res['payload'])),

shareReplay(),

finalize(() => console.log('Finalize Eexecuted'))

);

Output-



Now catchError block is executed once. but finalize block is executed twice. This is expected because we got here 2 subscriptions to courses$ observable. so 2 times observable was finished because both subscriptions were terminated.

My observation-

In createObservable method(in util.js) we error out our observable if status is 500. So in home component control directly moves to catchError operator. hence the above result.

Now consider situation, where we are have same setup as above in home.component, but in createHttpObservable method we are not erroring out our observable(setup we have before this lecture). we move to home component(but response structure is different, we do not payload property) in map operator we try to access payload property which does not exist, so we get error on console, because we have moved catchError operot block up in the operator chain.so there is nothng to handle our error and it is logged on console. Now if catchError block is after map then , control should have moved to catchError block and error is handled and we would have nothing on console.(this was happening in last lecture).

The reason that catchError is executed once after moving it up in observable chain is behaviour of shareReplay operator. I did some google on shareReplay. Here is link-

<https://www.learnrxjs.io/operators/multicasting/sharereplay.html>

this operator does not create new stream for each subscription. It makes other sunsvcription to share the stream created by first subscription.

lecture continues-

now,finalize is executed twice. This is expected because we have 2 subcrptions to courses$ observable. so 2 times this observable was finished because both subscriptions were terminated. If we want finalized to be handled once per http request then we would move the finalize block and place it before shareReplay operator and this is one thing that is general to catchError and to finalize operator, we don’t have to apply them only once per observable chain. If we have multiple steps in observable chain that might error out, then we might want to apply different error handling strategies at different points in observable chain. Maybe some errors can be recovered and we can use recovery observable or other errors might be unrecoverable and then in that case we want to handle them locally and rethrow them. So we should apply here the catchError block as close to source of error as possible. Lets try our code-

const courses$: Observable<Course[]> = http$

.pipe(

catchError(err => {

console.log('inside catch error block');

console.log(`Error occured, ${err}`);

return throwError(err);

}),

finalize(() => {

console.log('Finalized Executed');

}),

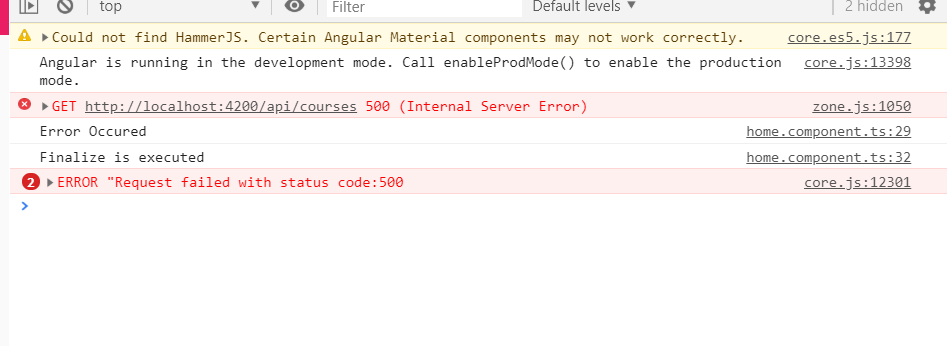
tap(() => console.log('HTTP request executed 1')),

map(res => Object.values(res['payload'])),

shareReplay()

);

Output-



Question-

1)still error is thrown 2 times as shown in console

<https://www.udemy.com/rxjs-course/learn/v4/questions/5969010>

Here we can see that catchError was invoked only once and finalize was executed only once as expected. This means that even though these observables has 2 separate subscriptions , we are sharing here execution of http observable (one highlighted)between 2 subscriptions using shareReplay.

const courses$: Observable<Course[]> = http$

.pipe(

catchError(err => {

console.log('Error Occured');

return throwError(err);

}),

finalize(() => console.log('Finalize is executed')),

tap(() => console.log('HTTP Requested was executed')),

map(res => Object.values(res['payload'])),

shareReplay()

);

So that is why we have one only 1 http request that is either going to error out or complete. And with that we have covered the catch and rethrow strategy and finalize operator. Lets try a new error strategy, that is going to be retry http request.

27)The retry Rxjs Error handling Strategy.screensflow

Here we will cover new rsjs error handling strategy- retry. for that remove code for strategy that we covered in last lecture. Sometimes a backend request may fail due to excessive load on server but it might succeed if we retry after lets say couple of seconds. First we need to simulate this thik on backed.

In get-courses.route, we have modified our request so, sometimes we get error and some times we get good response. Code-

Getcourses.route.ts-

export function getAllCourses(req: Request, res: Response) {

const error = (Math.random() >= 0.5);

if (error) {

console.log('ERROR loading courses!');

res.status(500).json({message: 'random error occurred.'});

}

else {

setTimeout(() => {

res.status(200).json({payload: Object.values(COURSES)});

}, 200);

}

There is 50 percent chance that our backend will throw error. Restart the server to see changes.

Now what we wnt to do is, if our request fails we want to wait for 2 seconds and retry the request. So if request fails we wait for 2 seconds then try again, if it fails again then we wait for 2 seconds and try again etc. we can implement this retry logic by using rxjs **retryWhen** operator. This operator receives here as the first argument , an errors observable. so this is an observable that is going to emit an error each time that the stream that we are retrying throws an error. We are talking about this stream-

http$

.pipe(

tap(() => console.log('HTTP request executed 1')),

map(res => Object.values(res['payload'])),

shareReplay()

When ever http stream throws an error, the stream will finish. So it will not complete successfully, it will error out. What retryWhen is going to do is to create a brand new stream, a brand new http stream and it is going to subscribe to that stream and it will do it successfully until the stream does not error out. So this is errors observables(that we pass as argument to function, which is passed to retryWhen), that emits values each time when one of streams(we are talking about above stream, code posted) getting retried errors out. And this function (passed as an argument to retryWhen) needs to return an Observable. this observable is going to tell retryWhen, when to retry. retryWhen operator will retyrn when observbel returned by this operator emits value. So we can retry immediate retry whenever we get failure, so if that is the case so just return errors observable from function. like this-

const courses$: Observable<Course[]> = http$

.pipe(

tap(() => console.log('HTTP request executed 1')),

map(res => Object.values(res['payload'])),

shareReplay(),

retryWhen(errors => errors)

);

However in practice we usually don’t want to retry immediately after a failure because many of these http request failures are due to intermittent problems, so we want to wait, lets say for example 2 seconds before trying to perform http request again. So in order to do that we just need to modify the observable that we are returning from retryWhen. So we use delayWhen operator to say that we want to delay the values emitted by this observable(errors) by lets say 2 seconds. code-

[Home.component.ts-](http://Home.component.ts-)

ngOnInit() {

const http$ = createHttpObservable(`/api/courses`);

const courses$: Observable<Course[]> = http$.pipe(

tap(() => console.log('HTTP Requested was executed')),

map(res => Object.values(res['payload'])),

shareReplay(),

retryWhen(errors => errors.pipe(

delayWhen(() => interval(2000))

))

);

this.beginnerCourses$ = courses$

.pipe(

map(courses => courses

.filter(course => course.category === 'BEGINNER')));

this.advancedCourses$ = courses$

.pipe(

map(courses => courses

.filter(course => course.category === 'ADVANCED')));

}

we can do this by using timer function. so we are going to say that whenever errors observable emits a value ,we are going to return here(in function passed to delayWhen) a observable that we are going to build using timer operator, that is going to emit value after 2 seconds. So this way each time we have error we are going to wait for 2 seconds before emitting a value back to retryWhen.

to see how delayWhen and timer works go to this link-

<https://reactivex.io/rxjs/class/es6/Observable.js~Observable.html#instance-method-delay>

if we simple use delay instead of delayWhen, what we would do in this case would be to delay whole error stream by a total of 2 seconds. This is not what we are looking for. We want after each error , wait for 2 seconds. So the correct way to do that is to use delayWhen and return after the error gets emitted an observable, that’s going to emit a value after 2 seconds and only after that time has elapsed then retryWhen is going to subscribe gain to this observable(below)

http$

.pipe(

tap(() => console.log('HTTP request executed 1')),

map(res => Object.values(res['payload'])),

shareReplay(),

and its going to trigger a new http request. Now run this code. In netwrork tab you can see that request fails, then it is made gain until it is successful.

And with this we have covered the retry error handling strategy. Lets now take this playground that we have here with several examples and use it to quickly cover a series of commonly used operators.

Question-

<https://www.udemy.com/rxjs-course/learn/v4/questions/5970900>

Lets say you want to retry only 3 times that too after interval of 2 seconds, then use this-

const courses$: Observable<Course[]> = http$

.pipe(

tap(() => console.log('HTTP Requested was executed')),

map(res => Object.values(res['payload'])),

shareReplay(),

retryWhen(errors => errors.pipe(

delayWhen(() => interval(2000)),

take(3)

))

);

28)The startwith RxJs Operator – simplifying the course component

Here we will simplify our course component. We are going to refactor it so that it uses the startWith operator. The goal of the operator is to initialize a stream with a given initial value. we built lessons$ stream by conacatting these 2 streams-

this.lessons$ = concat(initialLessons$, serachLessons$);

this logic works great but lets see how we can do this with a bit less code. So intead of using concat here, what we re going to do,we assign to lessons$ output of our typeahead Initial code-

course.component.ts-

export class CourseComponent implements OnInit, AfterViewInit {

courseId: string;

course$: Observable<Course>;

lessons$: Observable<Lesson[]>;

@ViewChild('searchInput') input: ElementRef;

constructor(private route: ActivatedRoute) {}

ngOnInit() {

this.courseId = this.route.snapshot.params['id'];

this.course$ = createHttpObservable(`/api/courses/${this.courseId}`);

}

ngAfterViewInit()

{

const serachLessons$ = fromEvent<any>(this.input.nativeElement, 'keyup')

.pipe(

map(event => event.target.value),

debounceTime(100),

distinctUntilChanged(),

switchMap(search => this.loadLessons(search))

);

const initialLessons$ = this.loadLessons();

this.lessons$ = concat(initialLessons$, serachLessons$);

}

loadLessons(search = ``): Observable<Lesson[]> {

return createHttpObservable(`/api/lessons?courseId=${this.courseId}&pageSize=100&filter=${search}`)

.pipe(

map(res => res['payload'])

);

}

Refactored code-

Course.component.ts-

ngOnInit() {

this.courseId = this.route.snapshot.params['id'];

this.course$ = createHttpObservable(`/api/courses/${this.courseId}`);

}

ngAfterViewInit()

{

this.lessons$ = fromEvent<any>(this.input.nativeElement, 'keyup')

.pipe(

map(event => event.target.value),

startWith(''),

debounceTime(100),

distinctUntilChanged(),

switchMap(search => this.loadLessons(search))

);

}

Now you can run your code and see that same thing is happening with lot less code now.

Here we have initialized this stream with a value-

fromEvent<any>(this.input.nativeElement, 'keyup')

.pipe(

map(event => event.target.value),

startWith(''),

debounceTime(500),

distinctUntilChanged(),

switchMap(searchTerm => this.loadLessons(searchTerm))

);

29)Rxjs Throttling vs Debouncing – Understand the differences

In this lecure we re going to introduce throttling in rxjs. We are going to introduce throttle function and we are going to introduce throttleTime operator. Lets first discuss throttling in general and how it compares to debouncing, that we have used in course component(in typeahead). These are 2 closely related notions that are often mixed up. Consider this example-

fromEvent<any>(this.input.nativeElement, "keyup")

.pipe(

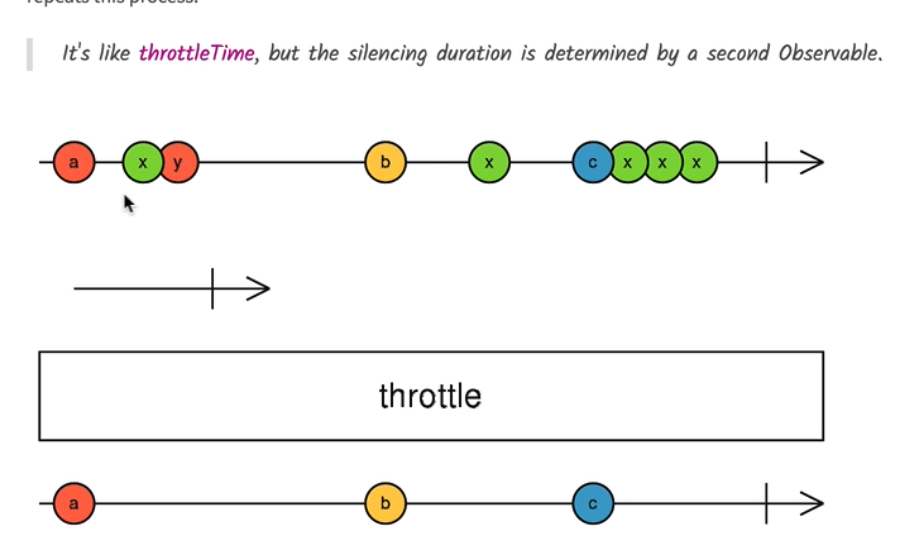
map(event => event.target.value),

debounceTime(400)

).subscribe(console.log);

So we only get value on console, once the input that the user is typing , is stable. So we are only going to get something once we have here a value that has been stable for atleast 400 ms. So it means if user keeps on typing quickly then there will never be output, because value that we are debouncing is not yet stable. So debouncing is about waiting for a value to become stable. This is very different from notion of throttling.

Throttling is somewhat similar to debouncing in the sense that we are also trying to reduce the number of values in our stream. But the way we are doing it is very different. lets have a look at throttle marble diagram-



Here we have input stream that is emitting lot of values. Sometimes there is period of silence but in certain occasions the stream is really emitting values in fast succession. Values are very close to each other some times. This could be web socket connection that is sending values continuously to frontend(like to website containing exchange rate of given currency). This is an example of a stream for which we would like to limit the output rate. So its not really useful for frontend to receive this information , updated more than say once in a minute. **Throttle is used for limiting the output by limiting the number of values that gets emitted in certain interval**. Throttle operator allows us to implement that and for that it uses a auxiliary timer observable that is going to be used to determine when we should emit a value from input stream.

So in diagram, first line is a input stream which contains the values that we want to rate limit and we have here second auxiliary timer observable(second line of marble diagram) , that is going to emit values at certain points in time. whenever the auxiliary timer observable emits a value, then we should also emit a value in output. Lets see in marble diagram. Here input stream emited first value and we have outputted the same value. then we are throttling here this input stream. So we are going to wait lets say for example for one second to elapse before emitting a second value. one second will elapsed lest say for ex after arrow in auxiliary timer observable. so what will happen now is that we are waiting for next value to show up before putting it in the outut. Then once we output b, we are going to start counting again one second, one second will end here after emitting x, so we have filtered here x value and once second interval of one second finishes then we are going to emit next value c and we are going to start couting again one second , these 3 occurences of x all occured inside this 1 second interval, so they are all ignored.

note that use of this auxiliary observable makes the use of throttling mechanism very flexible. We can have this observable increase the throttling rate or decrease it according to some external conditions. What determines the throttling rate is the values that are emitted by this(auxilary) observable, this observable can have any logic that we want it does not have to be a periodic interval. Lets see what happens if we use throttling instead of debounce in our example above.

fromEvent<any>(this.input.nativeElement, 'keyup')

.pipe(

map(event => event.target.value),

throttle(() => interval(500))

)

.subscribe(console.log);

This operator receives a function and this function needs to return an observable. lets do a simple case where we are throttling and limiting the rate of these observables to maximum of one value per half second. for this we have to periodic emit a value and this value is emitted once per half second.

in the console we end up having here one value each half second maximum with throttling we have the guarantee that we have our output rate limited in time but we don’t have guarantee that our output is the latest value of stream. if we start typing hello, we will see h on console, ths is because rest words were type before half second and we stopped typing while throttling interval was still ongoing. So we can see that in case of type ahead we really want to use debouncing instead of throttling.

Now lets have a look here an easier way to implement this logic. Here we used generic throttle operator and we have created here observable that is triggering throttling manually by using here the interval function. an alternate way that achieves the same result and it’s a bit easier to read is to use throttleTime operator. It will internally create our interval. Code-

fromEvent<any>(this.input.nativeElement, 'keyup')

.pipe(

map(event => event.target.value),

throttleTime(500)

)

.subscribe(console.log);

}