30)Building a RxJs Custom operator

Here we will write our own rxjs operator. we are going to write debug rxjs operator which is going to help us a lot to debug our rxjs programms. Sometimes it is not easy to understand what is going on, when we apply multiple rxjs operators. It’s not always easy to understand what is going on just by reading the observable chain. So sometimes to better understand the program and especially to troubleshoot the problem we often use  **tap** operator for producing debugging logging statements. For example in case of this observable, we would like to log on console, the value that we receive in tap operator.

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

.pipe(

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

startWith(''),

tap(search => console.log(search)),

debounceTime(100),

distinctUntilChanged(),

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

);

This search value is whatever we type in input box.

In case of course observable we might want to log what we are receiving from backend. This is course observable-

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

}

Lets log what we are receiving from backend-

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

.pipe(

tap(course => console.log(course))

);

Now as our program gets more complex we would get a lot of tap operator throughout our observable chains and also we would get a lot of output in console. In order to avoid having large volumes of information in console, we would end up, after solving the problem that we were investigating, going back and commenting out or deleting statements that we added to produce these logs on console. Ideally we would like to keep the logging statement that helped us to understand a given part of the program because we might want to use them later on but we would like to have a way to turn them off like we do in logging system in backend where we have multiple logging levels. So that is what we are going to be implementing here in rxjs. We are going to be creating here a new operator that is going to be called debug operator that is going to allow us to turn on and off logging statements according to a predefine logging level that could be error, info, debug or trace.

In common folder create debug.ts file. inside this file , we will define our new operator. lets see what argument this operator needs. So how would we use it our program.

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

.pipe(

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

startWith(''),

// tap(search => console.log(search)),

debug(RxJsLoggingLevel.INFO, 'search'),

debounceTime(100),

distinctUntilChanged(),

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

);

Instead of tap, we would apply here debug operator(that we are about to implement) and we would start by defning logging level, so we would need constant that we would call **RxJsLoggingLevel** and our here we would be able to choose several options such as for example debug,trace, info. Lets say for example, this particular message we would like to log at info level. The second argument will be string that we want to add to the log. The text in this argument is going to show up in console followed by value emitted by the observable.

Lets see how we are going to implement this debug operator.

First we define constant debug. So this will be operator itself. We are going to be assigning to this constant a function but it is going to be a very special type of function. its going to be higher order function. the operator as we have seen is going to take 2 arguments- first is going to be logging level which is number. The second argument is going to be the message which is going to be a string. This debug function that we are defining here will need to return another function . that is why it is called higher order function, it’s a function that returns a function. in order to understand why debug is higher order function, lets go back to example observable chain that we have-

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

.pipe(

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

startWith(''),

// tap(search => console.log(search)),

debug(RxJsLoggingLevel.INFO, 'search'),

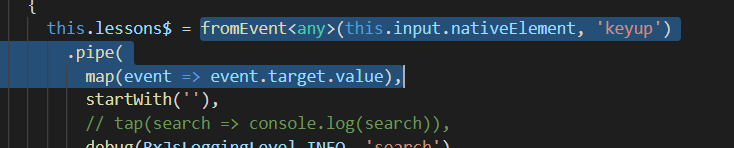
debounceTime(100),

distinctUntilChanged(),

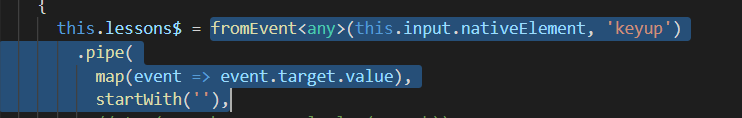
switchMap(search => this.loadLessons(search))

);

So as we can see at each point of observable chain what we are getting back here is an observable and input to this operator is going to be another observable. for example, here we have highlighted an observable-



which is returning stream of search keywords i.e text that we type in input box. This observable is then passed as input to **startWith** and startWith is going to take this observable (highlighted above) and it is going to output a new observable. so result of calling startWith, is also a observable (which is highlighted below).



so the operator is essentially higher order function that takes as an input a observable and gives us back another observable. so lets annonate here input of this function as being an observable of type any, the output of this function is going to be output of debug operator. in order to better understand that this is source observable, lets rename input observable as source. Now output of this function needs to an observable that logs message that we have passed in here and that returns us back the original observable unchanged. So we are going to take source observable we are going to pipe it and we are going to apply here, just like we are doing before, the **tap** operator. we are going to use tap operator implementing our logging logic. In tap operator lets assign here a value to emitted stream value(we will it val) then in function passed to tap we implement our logging logic. So what we want to log on console is message that we have received here in call to debug operator. but we don’t want to add only the message, we want to concate the message with value received from observable. what we have here at this stage, is skeleton of debug operator.

import { Observable } from "rxjs";

import { tap } from "rxjs/operators";

export const debug = (level: number, message: string) =>

(source: Observable<any>) => source

.pipe(

tap(val => {

console.log(message + ':' + val);

})

);

Lets now implement the logging logic.

31)The RxJs Debug Operator – Implementation Conclusion

In last lecture we have setup skeleton of our custom rxjs debug operator. we are now going to implement the logging logic. In this current version, logging message would always be printed out as the observable gets executed. So what we want to do is we will conditionally output this message depending on the logging level of the message and of the logging level of application.in order to do that we need to define multiple loging levels. Trace is most detailed logging level, DEBUG is going to be level that we will be using often.INFO is for informational messages only and ERROR is for errors only. Each of calls to debug operator needs to specify what is the logging level of message. Then in implementation of our debug operator we need to compare the application logging level with logging level of application .we are going to output the logging message to console only if the logging level of message is greater than or equal to rxjs application logging level, which we are yet to define. We will do that by defining a variable at level of file debug.ts, but we are not going to export it. We are going to make it private to this file. we call this variable rxjsLoggingLevel. So this is variable that we need to compare with logging level of message. We output the message only if level of message is greater than or equal to the rxjs global logging level. Now as this variable is private to this file , we need to give rest of application a away to modify this variable. So we define setter for this variable and we export it. Code-

Debug.ts-

import { Observable } from "rxjs";

import { tap } from "rxjs/operators";

export enum RxJsLoggingLevel {

TRACE,

DEBUG,

INFO,

ERROR

}

let rxjsLoggingLevel = RxJsLoggingLevel.INFO;

export function setRxJsLoggingLevel(level: RxJsLoggingLevel) {

rxjsLoggingLevel = level;

}

export const debug = (level: number, message: string) =>

(source: Observable<any>) => source

.pipe(

tap(val => {

if (level >= rxjsLoggingLevel) {

console.log(message + ":" , val);

}

})

);

Course.component.ts-

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

import {ActivatedRoute} from "@angular/router";

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

import {

debounceTime,

distinctUntilChanged,

startWith,

tap,

delay,

map,

concatMap,

switchMap,

withLatestFrom,

concatAll, shareReplay, throttle, throttleTime

} from 'rxjs/operators';

import {merge, fromEvent, Observable, concat, interval} from 'rxjs';

import {Lesson} from '../model/lesson';

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

import { RxJsLoggingLevel, debug } from '../common/debug';

@Component({

selector: 'course',

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

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

})

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}`)

.pipe(

// tap(course => console.log(course))

debug(RxJsLoggingLevel.INFO, 'course value')

);

}

ngAfterViewInit()

{

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

.pipe(

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

startWith(''),

// tap(search => console.log(search)),

debug(RxJsLoggingLevel.INFO, 'search'),

debounceTime(100),

distinctUntilChanged(),

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

debug(RxJsLoggingLevel.INFO, 'Lessons value')

);

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

.pipe(

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

throttleTime(500)

)

.subscribe(console.log);

}

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

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

.pipe(

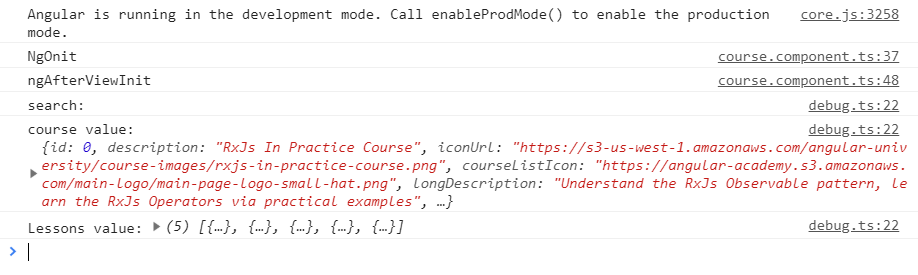
map(res => res['payload'])

);

}

}

Output-



When we start typing in input box –



We can see when we start typing we can see multiple logging statements on console. That really helps us to debug what is going on in here. We can see each and every stream. However as we can see this can be lot of information. For example here-

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

.pipe(

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

startWith(''),

// tap(search => console.log(search)),

debug(RxJsLoggingLevel.INFO, 'search'),

debounceTime(100),

distinctUntilChanged(),

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

debug(RxJsLoggingLevel.INFO, 'Lessons value')

);

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

.pipe(

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

throttleTime(500)

)

.subscribe(console.log);

Here output of search string and the output here of lessons values, these should b eon probabaly different logging levels other than info. Lets change it. this.lessons$ = fromEvent<any>(this.input.nativeElement, 'keyup')

.pipe(

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

startWith(''),

// tap(search => console.log(search)),

debug(RxJsLoggingLevel.TRACE, 'search'),

debounceTime(100),

distinctUntilChanged(),

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

debug(RxJsLoggingLevel.DEBUG, 'Lessons value')

);

So default logging level is going to be info, so this means now we will get only one logging statement on console, which is because of this-

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

.pipe(

// tap(course => console.log(course))

debug(RxJsLoggingLevel.INFO, 'course value')

);

Now in lessons observables, we see no output statements as expected.but lets say that bow we have found here an issue with typeahead and we will like to have some logging information. We can do this by increasing the logging level of application by using setter that we have defined in debug.ts. lets increase logging level to debug level. Now we will see only console logs corresponding to second debug in lessons observable. so we do not get message corresponding to trace level.

32)The Rxjs forkJoin Operator –In-Depth Explanation and Practical example

The forkJoin operator allows us to launch several tasks in parallel wait for those tasks to complete and then we can get back the result of each task and use those combined results togather. actually you get array in subscribe, each element of array is value of different observable. the We are going to be using course component to give an example of how forkJoin might be used with http request. Right now course.component is-

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

import {ActivatedRoute} from "@angular/router";

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

import {

debounceTime,

distinctUntilChanged,

startWith,

tap,

delay,

map,

concatMap,

switchMap,

withLatestFrom,

concatAll, shareReplay, throttle, throttleTime

} from 'rxjs/operators';

import {merge, fromEvent, Observable, concat, interval} from 'rxjs';

import {Lesson} from '../model/lesson';

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

import { RxJsLoggingLevel, debug, setRxJsLoggingLevel } from '../common/debug';

@Component({

selector: 'course',

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

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

})

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()

{

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

.pipe(

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

startWith(''),

debounceTime(100),

distinctUntilChanged(),

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

);

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

.pipe(

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

throttleTime(500)

)

.subscribe(console.log);

}

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

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

.pipe(

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

);

}

}

So we get courseId from route parameter then we use make http request to search lessons of that course depending upon what we typed in input box. Initially we want all lessons, so we used startWith operator to search with empty string.

Now lets see how we can use forkJoin operator in this component. Lets do the following. We are going to launch here one course observable that is going to fetch course from this http url. Then we are going to create a second observable that’s going to be lessons observable., this observable is going to fetch the list of lessons fro backend. So we define lessons$ variable and call loadLessons function. lets say we would like to trigger these 2 http request at the same time, send them to backend at the same time , have the backend serve each request in parallel and then wait for result of both course observable and lessons observable to return from backend and only at that moment we would like to trigger some extra logic when we have course and lessons. This type of logic can be easily implemented by using **forkJoin** operator. name of operator comes from the fact that we are forking 2 streams then we are joining results of 2 streams togather when both requests are completed. Then we subscribe to forkJoin. What we get in subscribe is tupul value that contains both the output of course$ observable and output of lessons$ observable. lets have a look what that tupul value looks like, in order to do that we are going to pipe this forkJoin observable with tap operator, so that the tupul value that we get here is a typescript type that looks like an array. This is very similar to array with 2 elemnts. First element is going to be course which is output of first observable and second value is going to be value emitted by lessons$ observable. so this is going to be array of lessons. Then we log these value s into console. Code-

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

import {ActivatedRoute} from "@angular/router";

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

import {

debounceTime,

distinctUntilChanged,

startWith,

tap,

delay,

map,

concatMap,

switchMap,

withLatestFrom,

concatAll, shareReplay, throttle, throttleTime

} from 'rxjs/operators';

import {merge, fromEvent, Observable, concat, interval, forkJoin} from 'rxjs';

import {Lesson} from '../model/lesson';

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

import { RxJsLoggingLevel, debug, setRxJsLoggingLevel } from '../common/debug';

@Component({

selector: 'course',

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

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

})

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'];

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

const lessons$ = this.loadLessons();

forkJoin(course$, lessons$)

.pipe(

tap(([course, lessons]) => {

console.log('course', course);

console.log('lessons',lessons);

})

)

.subscribe();

}

ngAfterViewInit()

{

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

.pipe(

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

startWith(''),

debounceTime(100),

distinctUntilChanged(),

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

);

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

.pipe(

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

throttleTime(500)

)

.subscribe(console.log);

}

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

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

.pipe(

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

);

}

}

in output we can see these 2 outputs. So these http requests were performed in parallel but the tupul value was only emitted when both of these observables were completed. If one of these observable is not completed then no tupul value will ever be emitted.

If by some reasons one of these observable emits multiple values and then completes, then the value that we are going to get here is last value emitted before completion. So **forkJoin** operator is ideal for example hadling parallel http request or performing long running calculations in parallel that might emit multiple values and then eventually emit a final value and then they complete.