9)What are RxJs Operators? Learn the Map Operator

We will call same api with our observable then we are going to transform data(that we get back) into list of courses. We are going to take http$ observable that we have defined, from this observable we are going to define other observable called courses. This courses observable is going to emit arrays of courses instead of exact payload that came from server. We will apply map operator on observable. It takes function as argument. Output of map operator is, it is going to give us another observervable. So an operator is way of deriving one observable from another. We know how map works.

There are 2 docs for Rxjs, here are links-

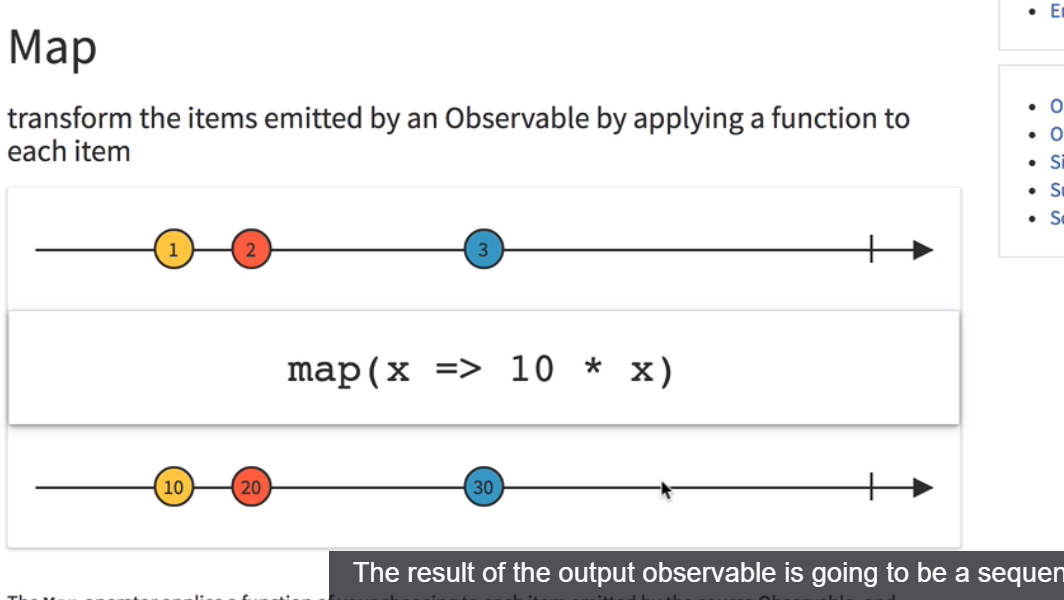
<http://reactivex.io/documentation/operators/map.html> (this is used in videos)

<https://rxjs-dev.firebaseapp.com/api/operators/map>

docs refrence used in this course is –

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

here function that we pass to map will take values of original o servable and transform them into values of second observable. so value of new observable will be what we return in map function. so all values of output observable are result of applying this function to the values of the input observable.



Lets talk about code now. We have outsource the function which returns us mani observable, in util.js. util.js is in common folder.

util.js-

import { Observable } from 'rxjs';

export function createHttpObservable(url: string) {

return Observable.create(observer => {

fetch('/api/courses')

.then(response => {

return response.json();

})

.then(body => {

observer.next(body);

observer.complete();

})

.catch(err => {

observer.error(err);

});

});

}

Now in about component we define new observable based on above our http observable. Whenever we want to derive new observables from existing observables , we need to use one of rxjs operators for that. For that we are going to be needing pipe function. the pipe function is what allows us to chain multiple operators in order to produce new observables. About.component.ts-

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

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

import { map } from 'rxjs/operators';

@Component({

selector: 'about',

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

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

})

export class AboutComponent implements OnInit {

constructor() { }

ngOnInit() {

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

const courses$ = http$

.pipe(

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

);

courses$.subscribe(

courses => console.log(courses),

() => { },

() => console.log('Completed')

);

}

}

Here we used Object.values function to get array from individual properties of Object.

As we can see that representing http request as an observable is a great way to have all these multiple rxjs operators avalaible to combine multiple http request. Now we will show all these courses thatw e fetched from backend in courses component.

10)Building components with RxJs – Imperative design

Here we will use data that we got from api in courses component. The goal here is to understand how we can use RxJs observables to build our components using reactive design. In courses tab we have 2 types of courses beginner and advance. But in http observer we are getting both types of courses. First we move our http$ Observable code to home component, this is the component that contains courses screen. One way of doing it is-

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

ngOnInit() {

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

const courses$ = http$.pipe(map(res => Object.values(res['payload']))

);

courses$.subscribe(

courses => {

this.beginnerCourses = courses

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

this.advancedCourses = courses

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

},

noop,

() => console.log('Completed')

);

}

So this is one valid way of building our application.Here we take data from success callback and pass it to our template. The problem with having a lot of logic inside subscribe callback is ,that will not scale very well in complexity. We will quickly end up in same condition where we have nested subscribe blocks inside each other. that is what we are trying to avoid in first place by using rxjs. One of main things that we are trying to avoid by using rxjs is precisely to avoid callback hell. It means if we want to use rxjs as it was meant to be used, we should avoid adding lot of logic inside our subscribe calls and we should definitely not nest subscribe calls togather. That is rxjs anti pattern.

Using RXJS as we have seen here, by calling subscribe and taking data inside of it, is considered as imperative approach of implementing our components. Now we are going to provide different way of implementing these components using reactive design.

11)Building Components with Rxjs Rective design

We used http observable to implement our component but we did it using imperative design , that in a more complex component would lead to nested subscribe calls. Here we will refactor our component to use reactive design.

Here we will define 2 sources of data. One is going to be beginner courses observable and others is going to be , advanced courses observable. We are going to derive these 2 using Rxjs operators and we are going to pass these observables directly to templeate. We will see it’s advantages later. Our subscribe method will be empty.

So we change beginnerCourse and advancedCorse to be of observable type. When we define observables at level of our components , this also means that data itself is not avalaible here for direct mutation in this component. We only have here definations of streams of data. This provides other advantages as we will see in a moment.

First we drive new 2 observables from courses$ observable.

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

ngOnInit() {

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

const courses$ = http$

.pipe(

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

);

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'))

);

}

now lest see how we can use these observables directly in template. In html we can see that we need here array of courses to pass here to courses-card-list. We have 2 such card list one for each type of course.

<mat-tab label="Beginners">

<courses-card-list

[courses]="beginnerCourses">

</courses-card-list>

</mat-tab>

<mat-tab label="Advanced">

<courses-card-list

[courses]="advancedCourses"

></courses-card-list>

</mat-tab>

So lets see how we can extract the values of our observables here in templates.

We could manually subscribe to it and assign it in html but as we are using angular, we used async pipe. Async pipe is going to subscribe to these observables and it’s going to retrieve that data and it is going to assign that data to courses property of course-cad-list. This is very convinient way of passing observable data to our template. **The async pipe also has advantage that it unsubscribe from the observable once the component gets destroyed. So here is reactive version of our component-**

[Home.compoennt.html-](http://Home.compoennt.html-)

<mat-tab-group>

<mat-tab label="Beginners">

<courses-card-list

[courses]="beginnerCourses$ | async">

</courses-card-list>

</mat-tab>

<mat-tab label="Advanced">

<courses-card-list

[courses]="advancedCourses$ | async"

></courses-card-list>

</mat-tab>

</mat-tab-group>

Now we do not need subscribe method in component. now subscriptions are going to be handled at level of template. So we will never run into nested subscription scanerio in our component. so this is reactive version of our component. we do not use subscribe method instead we simply define stream of values using observables and rxjs and that’s all that we at level of our component.

This works as last lecture , but we don’t use here calls to subscribe in our component. We are not just using rxjs to make call to backend and then extracting the data in subscribe and making fully imperative program. Instead we are basing our whole program around the notions of observable streams of data. This approach looks promising as it is more maintainable. We dnt run into the case of nested subscribes and other problems that we would face while trying to use rxjs imperatively but we have accidentally stumbled upon an issue which is we are doing 2 http requests to backend instead of one and we are fetching data twice fro backend. Lets see it and see which rxjs operaot can fix it.

12)Sharing http responses with the shareReplay Operator

This operator solves very common problem that we will face while building reactive applications. To understand this problem, lets go back to our code.

Open network tab and refresh the page, you will see that we are making 2 calls to backend. These calla gives us exactly same data. So what is going on here? In our code we have 2 observables and these observables are each subscribe to using async pipe. So what we have here is 2 different subscriptions to 2 different observables, that both are derived from same http observable. The problem is that each subscription we are doing one to beginner’s course and other to advance courses is going to trigger separate http request because each subscription is taking over blueprint of the stream that we have here(courses$ observable), and its using it to instantiate concrete stream of values. That is why we have 2 http request. If we do third subscription, we will see 3 http calls in networks tabs.

To solve this problem we will use another rxjs operator **shareReplay**.What we want to do is we want to take courses$ observable , which is http observable and we want to somehow share the execution of this particular stream across multiple subscribers. so we somehow want to avoid the default observable behaviour which is to create a complete new stream by subscription. Instead we want this stream(courses$) to be executed only once and then result of that stream we want it to be replayed to each new subscriber. To obtain this behaviour we pipe new operator, **shareReplay**. Like name implies this operator is going to make sure that our http response is going to be passed on to each new subscription instead of executing again , the same http request.

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

.pipe(

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

shareReplay()

);

To confirm that this is indeed working as expected, lets add some logging to http execution. We will log that http requested was executed.in order to do that we introduce new operator called **tap**. Tap is operator that is meant to be used to produce side effects in our observable chain. So whenever we want to update something outside of observable chain such as for ex updating here a variable at level of the component or in this case to issue a logging statement. Code-

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

);

this.beginnerCourses$ = courses$

.pipe(

map(courses => courses

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

this.advancedCourses$ = courses$

.pipe(

map(courses => courses

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

}

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

<mat-tab-group>

<mat-tab label="Beginners">

<courses-card-list

[courses]="beginnerCourses$ | async">

</courses-card-list>

</mat-tab>

<mat-tab label="Advanced">

<courses-card-list

[courses]="advancedCourses$ | async"

></courses-card-list>

</mat-tab>

</mat-tab-group>

Now we can see in console as well as in network tab that our request is only sent once.

So we can indeed see in this case , the execution of courses$ observable is shared across multiple subscriber.

Now having introduced shareReplay operator we have here good starting point for covering more complex scenarios such as combining multiple requests, performing them in parallel and more operators, all of them with their own rxjs operators.

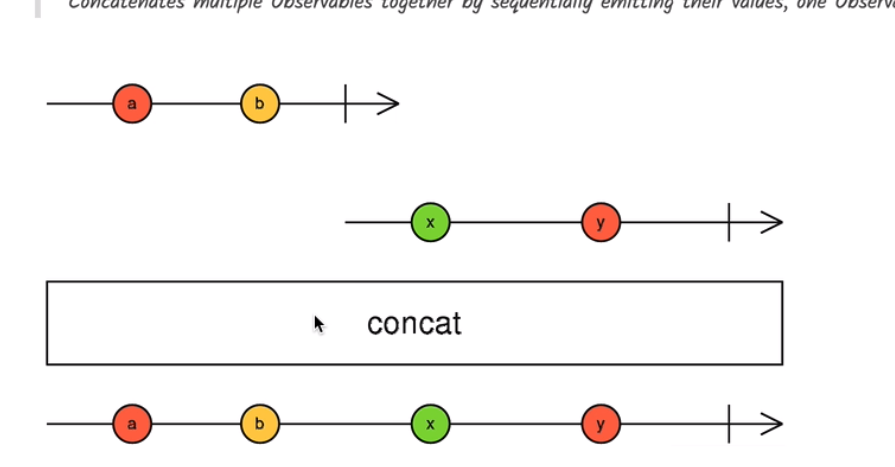
13) RxJs Higher-Order Mapping Operators PDF

Here we downloaded the pdf

14)Observable concatenation – In depth explanation

In next lessons we will be introducing new ways of combining observables togather. Here we will be covering observable concatenation. we will see quick demo here and then in next lecture we are going to give practical use case of concatenation in the scenario of http requests by using **concapMap** operator .

so lets introduce the notion of observable concatenation. lets move to about.compoennt.This is what we want to do here-



Link-

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

here we want to combine these 2 observable togather, to first emit source 1 values and then only after this observable completes only then we would like to show the values of second observable. when all observables complete(here we have 2), then our resultant concatenation observable will also complete.

So this a sequential concatenation of all values of each stream. Remember we emit values of next observable only after first observable completes. We can concatenate as many observables as we want. Code-

About.component.ts-

ngOnInit() {

const source1$ = of(1, 2, 3);

const source2$ = of(4, 5, 6);

const source3$ = of(7, 8, 9);

const result$ = concat(source1$, source2$, source3$);

result$.subscribe(value => console.log(value));

}

Output-

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

Here we created observables using of method . our observable will emit these values that we pass it of and after emitting these it will complete.

So the key notion when it comes to concatenation is the notion of completion. If source1$ observable never completes, then values of other observables will never appear on screen. Code-

ngOnInit() {

const source1$ = interval(2000);

const source2$ = of(4, 5, 6);

const source3$ = of(7, 8, 9);

const result$ = concat(source1$, source2$, source3$);

result$.subscribe(value => console.log(value));

}

Here we will never see values from source2$ and source3$ because source1$ never completes. It is like this, first we subscribe to source1$, when it completes then we subscribe to source2$ and son on.

You might be wondering what is use of this, we will see practical use of these in next lecture using **concatMap** operator.

Question-

1)what is concat

<https://www.udemy.com/rxjs-course/learn/v4/questions/5949944>

15)Form Draft Pre-Save Example and Rxjs Filter Operator

Here we are going to implement an example of an operation that would benefit from observable concatenation. we will also cover another commonly used rxjs operator called filter. Lets atrt by defining 2 streams that we would like to concatenate.

In courses page we can edit course. We can change name, description etc. what we want to do is , we want to save course details as user edits it. We want to implement autosave functionality.Here we used value changes observable of form, this observable emits values of form(value property of form) whenever we change some value. we can combine this observable with other values. We use  **filter** operator to filter form values that are valid. Filter takes a function as a argument, only that values are emitted that return true for this function. This means if we change some value and overall form is not valid, then that form value will not be emitted.

ngOnInit() {

this.form.valueChanges

.pipe(

filter(() => this.form.valid)

)

.subscribe(console.log);

}

This is our first stream of values that we want to combine.

Now the second stream of values is going to be when http request( that we make to backend to save current form value).

First, lets see how we will implement it without observables.we use fetch api to make http request-

ngOnInit() {

this.form.valueChanges

.pipe(

filter(() => this.form.valid)

)

.subscribe((changes) => {

fetch(`/api/courses/${this.course.id}`,{

method: 'PUT',

body: JSON.stringify(changes),

headers: {

'content-type': 'application/josn'

}

})

.then(response => console.log(response));

});

}

Here fetch returns promise, so we can chain then call to fetch.

But like we did before ,we would like to handle these http requests as observables. So what we would like to do is to convert this promise into an observable. we can do that by calling **fromPromise** , this is an rxjs method that takes a promise and creates observable derived from it. Then we assign this observable to saveCourse$ variable. In order to trigger http request we have to subscribe to this observable. code-

Course-dialog.component.ts-

ngOnInit() {

this.form.valueChanges

.pipe(

filter(() => this.form.valid)

)

.subscribe(changes => {

const saveCourse$ = fromPromise(fetch(`/api/courses/${this.course.id}`, {

method: 'PUT',

body: JSON.stringify(changes),

headers: {

'content-type': 'application/json'

}

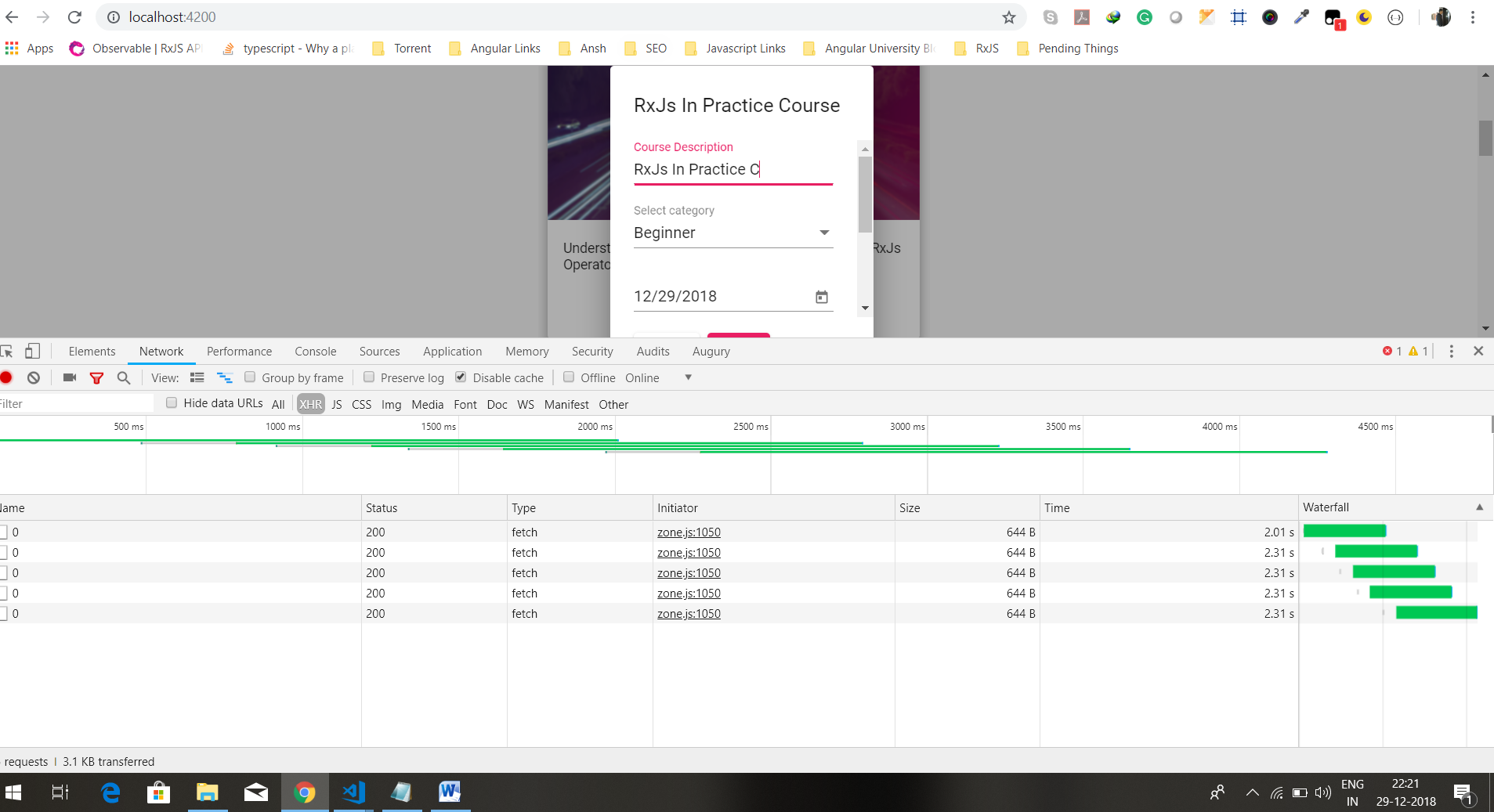
}));

saveCourse$.subscribe();

});

}

In this version of program we have combined the 2 streams by nesting here multiple subscription calls. This is something that we want to avoid on our RxJs pattern. This is anti pattern. Here we have subscribe inside subscribe. Also this code is not working as expected. Open network tab and , and filter only ajax request(click on XHR on upper menu). You will see that if type a sentence, a lot of requests are made. In waterfall execution of our requests, we can see that bunch of requests are executed at same time.



this is not the usual logic that we want for save operation. With save operation, what we want to do is we want to wait for the operation to complete before issuing another save. If we do not wait for previous save to complete then we have no surety that the last value save to database is actually the last valid value of form. What we want here is to wait for first save to complete before triggering a second save. So we need observable concatenation logic. Lets then see how we can solve these and prevent the need to nested subscribes in an elegant way using concatenation and **conCatMap**  operator.

16. The RxJs concatMap Operator - In-Depth Explanation and Practical Example

Here we will practical example of observable concatenation operation. We will see why concatenation is ideally suited for save operations. We want to make sure that our safe operations happen in the same order as the values are emitted. Implementation in last lecture does not provide this logic.

We move our logic to generate put request observable in another function named save Course. This will allow us to understand better , what is going on here.

ngOnInit() {

this.form.valueChanges

.pipe(

filter(() => this.form.valid)

)

.subscribe((changes) => {

const saveCourse$ = this.saveCourse(changes);

saveCourse$.subscribe();

});

}

saveCourse(changes) {

return fromPromise(fetch(`/api/courses/${this.course.id}`, {

method: 'PUT',

body: JSON.stringify(changes),

headers: {

'content-type': 'application/josn'

}

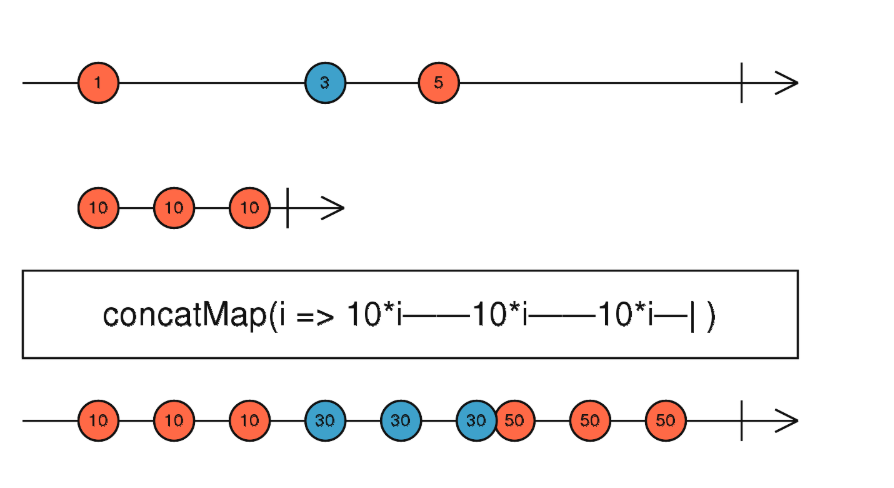
})

);

}

What we really want to do in this situation is, we want to take values of source observable(this.form.valueChanges) and for each value ,create saveCourse$ observable, then we want to concatenate all those derived observables togather in order to make sure that the save operations are done in right order. What we have here is mixture of mapping operation(where we are taking value of one observable(in first subscribe we have changes value) and creating second observable(by calling method))and concatenation logic./ so this mixture of transforming one observable into another and concatenating the result togather is best implemented using rxjs **concatMap**. See docs-

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



We have source observable that is emitting here series of values- 1,3,5. To concatMap we pass mapping function. what mapping function does is it takes input value and then it produces observables. This observable will emit 3 values for each source observable , each value is multiplied by 10 and then it will complete. So it’s a function that transform value into observable. so here is how concatMap works. We are going to take values of first observable(first line in marble diagram) until it completes and for each value for source observable we are going to create a second derived observable. so in our case values of the first observable are going to be valid form values that are getting emitted over time. our derived observable is our save operation. Here derived observable is emitting multiple values but in our case we are going to do http put call to backend and we are going to complete it immediately. But idea is same we take 1 value from source observable and we convert it into an observable. now as long as derived observable is emitting values, those values will be emitted here in output of concatMap. Only when derived observable completes, only in that case we are going to be creating second derived observable from second values of source observable. so once derived observable completes, we take next value from source observable and we are going to create second derived observable. and this process goes on until source observable and last derived observable(which is derived from last value of source observable) completes. Then our resultant observable also completes.

So this is the operation we are looking for.So we want to take formValues turn them into httpRequest and wait for first http request to complete before making second http request.

So lets refactor our code to use concatMap operator.

Code-

Course-dialog.component.ts-

ngOnInit() {

this.form.valueChanges

.pipe(

filter(() => this.form.valid),

concatMap(changes => this.saveCourse(changes))

)

.subscribe();

}

saveCourse(changes) {

return fromPromise(fetch(`/api/courses/${this.course.id}`, {

method: 'PUT',

body: JSON.stringify(changes),

headers: {

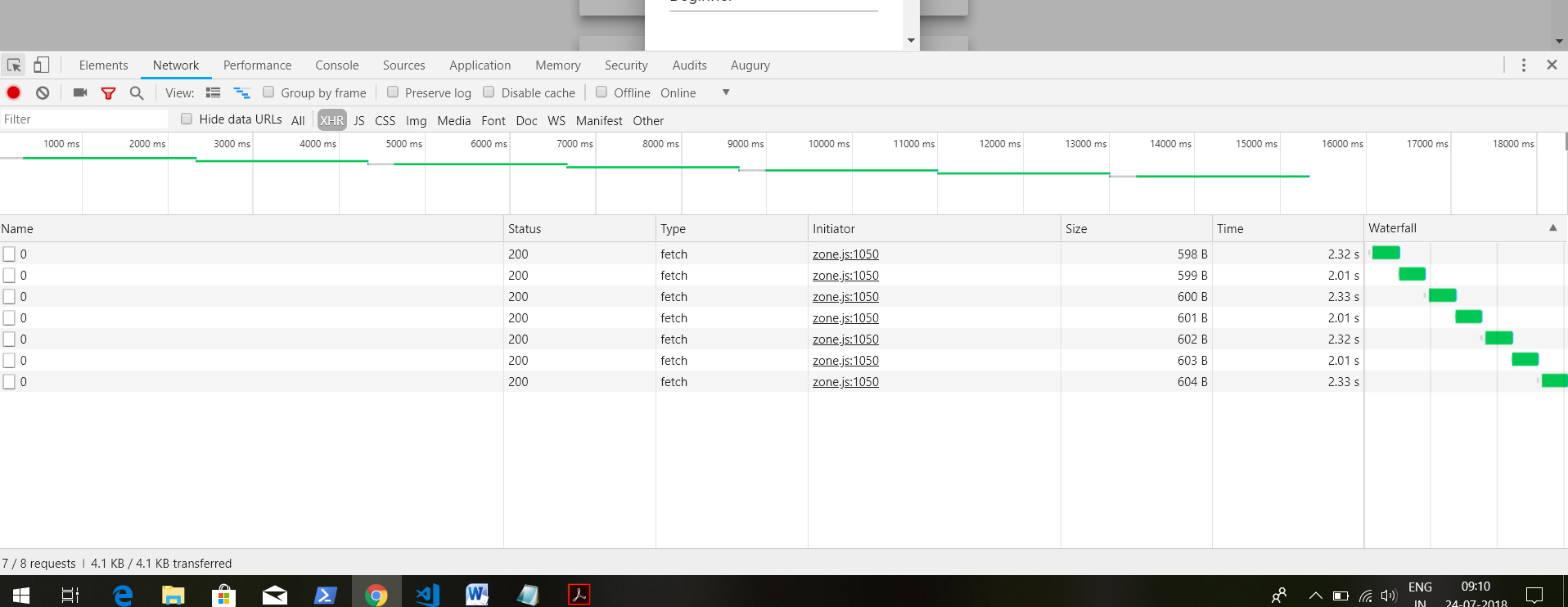
'content-type': 'application/json'

}

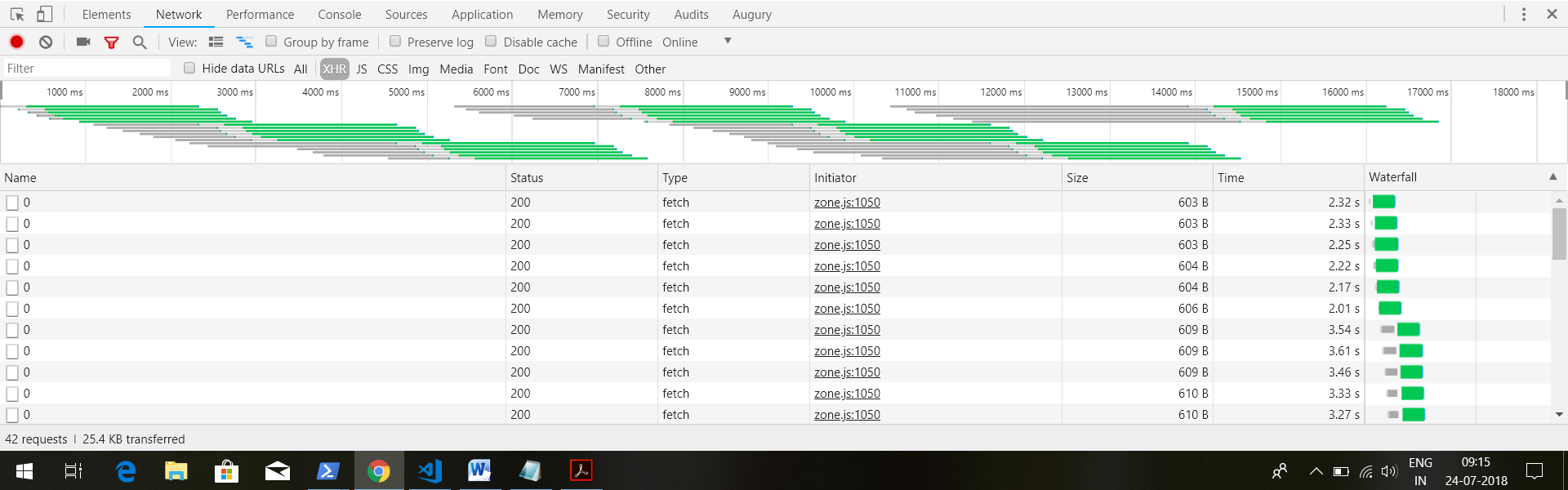
}));

}

In network tab we can see this-



Initially request were like this-



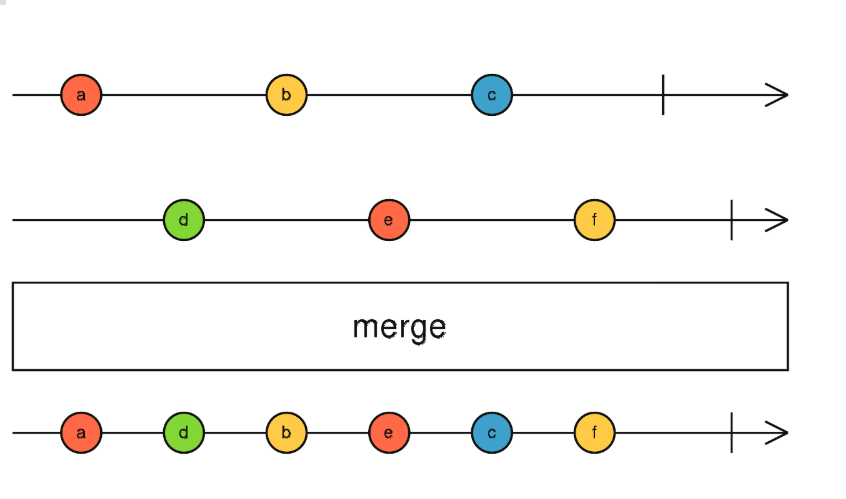
So second save request is made only after first save request is completed.

This means our save operation are happening in sequence as we want.we are making lot of request later we will learn about **debounceTime**  which allows us to reduce number of requests.

Now the question is instead of performing our operations in sequence, what if we want to perform them in parallel as fast as possible. This will introduce new observable combination strategy called merge.

17)Understanding the merge observable combination strategy

In last lectures we introduced concat strategy for combining observables. Concat as we have seen is all about completion, waiting for one observable to complete before subscribing and using the next observable. In this lecture and following one lecture we are going to be covering a new strategy for combing observables. It’s going to be merge strategy. Lets introduce merge and then cover **mergeMap**.



Merge should be used, when we want to take multiple observables, subscribe to all of them and notice here at the same time, and take the values of each of these observable. mergeMap is ideal for performing asynchronous operation in parallel. When one of these observable emits a value we immediately add it to the output. Our combined merge observable will complete when all observables are completed. Aslo if any of these observable throws an error then resultant observable is going to throw an error immediately. Code-

About.component.ts-

import { of,concat,merge } from 'rxjs';

ngOnInit() {

const interval1$ = interval(1000);

const interval2$ = interval1$.pipe(map(val => val \* 10));

const result$ = merge(interval1$, interval2$);

result$.subscribe(val => console.log(val));

}

Output-

**0**

**0**

**1**

**10**

**2**

**20**

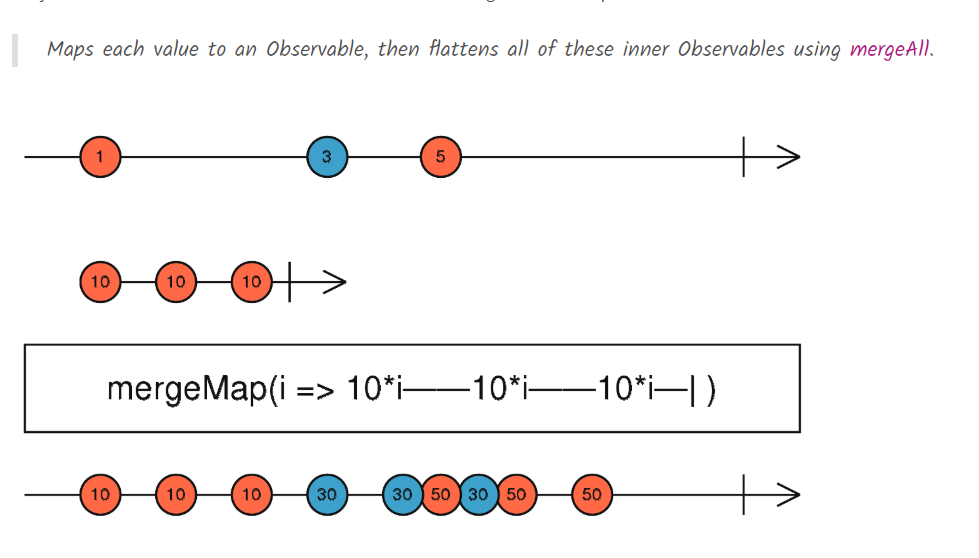
**3**

**30**

Merge straregy is ideal for performing long running operations in parallel and getting the results of each of these operations combined. lets see practical example of when would we like to use merge strategy.

18)The RxJs mergeMap Operator –In Depth Explanation

In last lecture we covered merge strategy for combining observables, here we going to show that merge is ideal for performing http requests in parallel as we have seen **concatMap** operator ensures that previous value is saved before making next request. This is the logic here that we want for save operations, but if you find yourself in situation where you would like to perform multiple calls to your backend in parallel and fetch results of each call as they arrive over time. then in that case you can use mergeMap operator. Lets see diagram-



mergeMap is very similar to concat map that we have already covered. Principle is same , we are going to take values of source observable and we are going to apply a mapping function, that is going to take values and produce a new observable. we took value1 and by running mapping function we produce an observable that emitted 3 times 10. These3 values appear in final result. Now take a look at what happens if several other values are emitted without a large gap between them. Second value 3 emits 2 values 30 and 30 but before it emits third value., 5 is emitted from source observable. so we are going to create a new observable using this mapping function and we are going to subscribe to it and start emitting values without having to wait that the previous observable all have completed. We are deriving from each value(of source observable) a new observable, subscribing to everything in parallel and outputting the values as they come from each source observable into the output. We will only complete the output observable when source observable has completed. In video we replace concatMap with mergeMap. Then in network tab we saw that all save request are being made in parallel.

So if order of observable values is important then we should use concatMap. If we want to perform observable in parallel then we should merge map.

Now in next lectures we are going to cover 2 other very commonly used mapping operators **exhaustMap** and **switchMap**.

19)Rxjs exhaustMap Operator – In-Depth Exlanation

Here we will see exhaustMap operator. go to course-dialog.component. here we click on save buttons to save course details. Now we want to prevent user from clicking save button multiple times, and trigger multiple parallel calls to the backend. Notice that clicks that we do here on save button is also stream of values. Lets implement this functionality in following way as an example, lets subscribe here to this stream of click values and whenever we get click value, we are going to map that click value to an observable using a mapping operator. Lets say that for example we were using **conactMap**, so we are sure that save opertaions are sequential here is what it would look like-

Course-dialog.componennt.ts-

@ViewChild('saveButton') saveButton: ElementRef;

saveCourse(changes) {

return fromPromise(fetch(`/api/courses/${this.course.id}`, {

method: 'PUT',

body: JSON.stringify(changes),

headers: {

'content-type': 'application/json'

}

}));

}

ngAfterViewInit() {

fromEvent(this.saveButton.nativeElement, 'click')

.pipe(

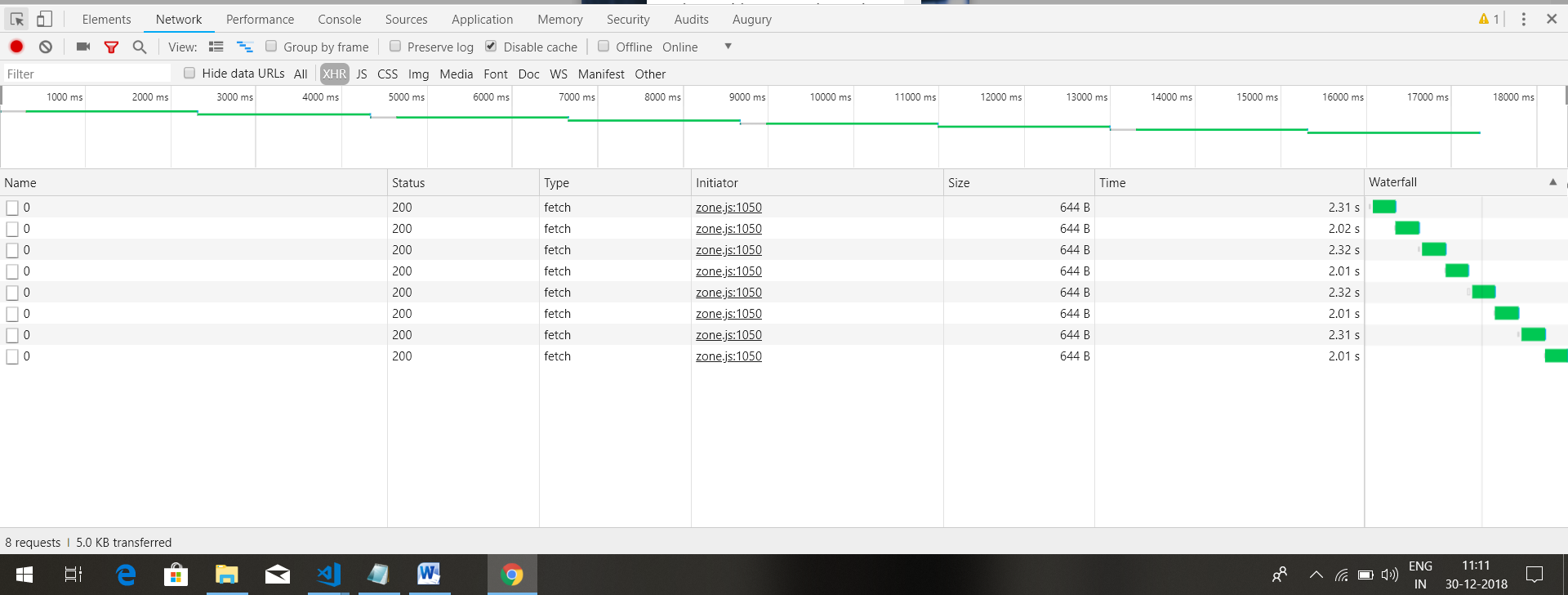
concatMap(() => this.saveCourse(this.form.value))

)

.subscribe();

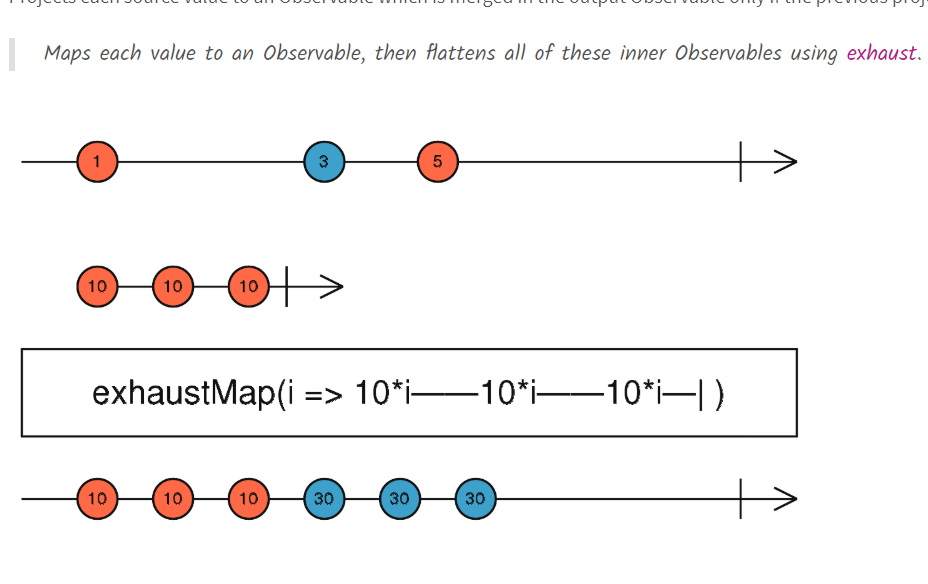
}

So we subscribe to this stream of click and for each clicks we call backend and save the course. And if user clicks multiple times we are going to be emitting multiple save requests. If we click save button multiple times, this is what happens-



So we wait for first request to complete before issuing

Using concatMap in this situation causes problems. Here we want to ignore additional clicks. We can do this by exhaustMap. We can obtain functionality of ignoring all clicks made on save button while a save is already ongoing by using exhaustMap operator instead of concatMap. Lets look at marble diagram-



Source observable emits multiple values 1,3,5. As usual each value is going to be transformed into separate observable by using mapping function that we pass to exhaustMap. It takes a value and returns a observable. this derived observable emits multiple values and then completes. values emitted by derived observable(by first value of source observable) are passed to output of exhaut map. This derived observable will complete. Critical thing is if our derived observable(form first value) is still emitting values(not completed) and our source observable emits another value, then that value will be ignored. That is why we do not see values corresponding to 5 in output observable. Because 5 was emitted from source observable at a time when derived observable from value 3 was not completed. So as a result 5 was ignored. so derived observable was not created from 5.

**So critical notion here is ignoring values while current observable is till ongoing**. This is what we want in our code. We want to ignore clicks if a save request is still ongoing.

Code-

ngAfterViewInit() {

fromEvent(this.saveButton.nativeElement, 'click')

.pipe(

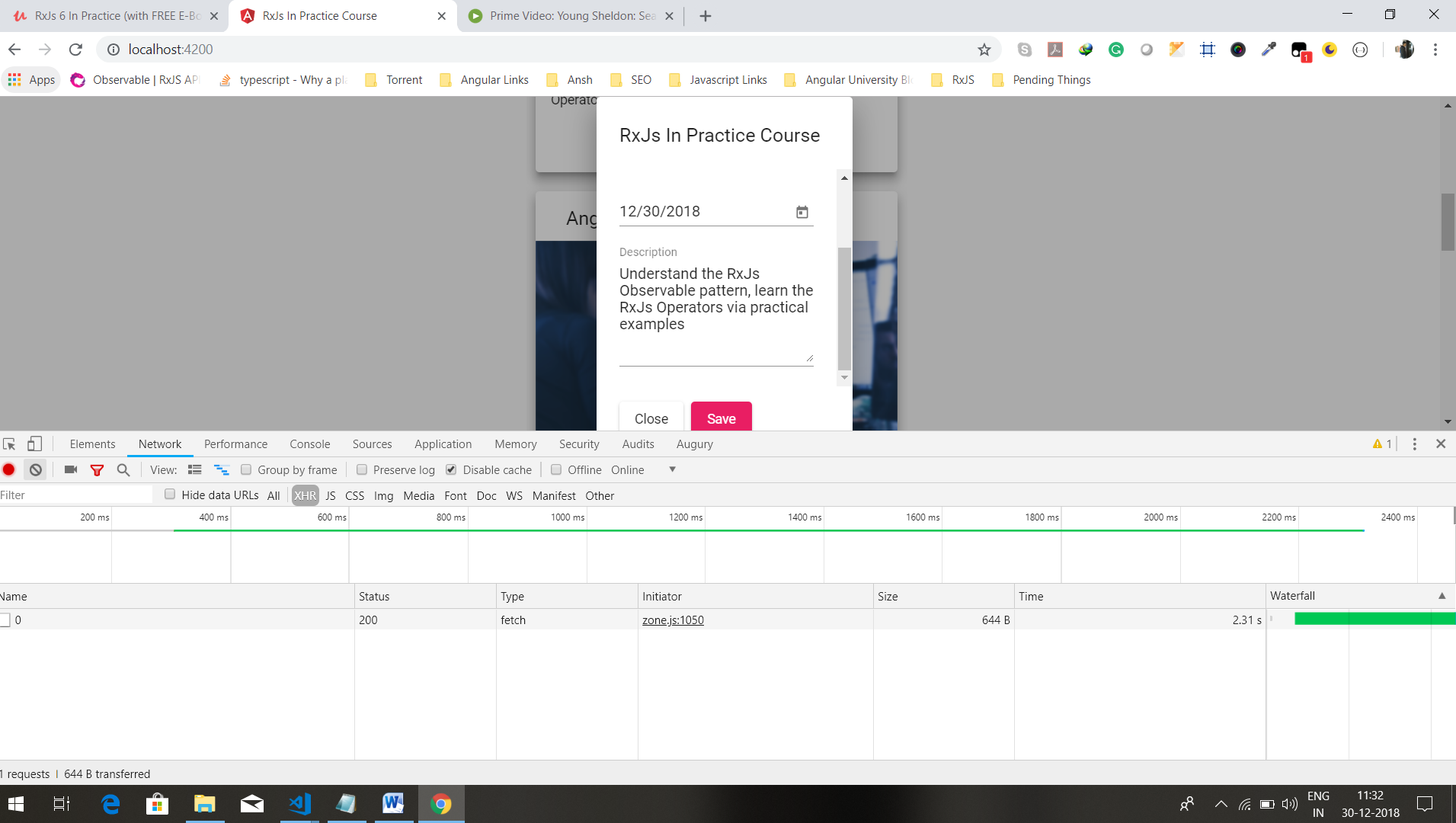
exhaustMap(() => this.saveCourse(this.form.value))

)

.subscribe();

}

Output-

we can see that only one request is issued, even if we clicked on save multiple times.

We covered several observable combination strategies. Concat,merge and exhaust. We are going to cover another strategy which is switch strategy. In order to do that lets talk about observable unsubscription and lets make our http observable unsubscriable.

20)Unsubscription in detail – implementing a cancellable HTTP Observable

Here we will show how to implement unsubscription by making our http observable cancellable. This means users of our http observable will be able to cancel in flight request which is very useful feature for example , in search, if a search request is ongoing and user just types different search, we want to cancel previous ongoing search request .

Lets give a quick example that shows how cancellation works. We define an observable that never ends $interval1. Lets say after some time we are no longer interested in values for this observable. so we will have to unsubscribe, to unsubscribe we need refrence to subscription, it is returned by subscribe method. Code-

import { interval } from 'rxjs';

ngOnInit() {

const interval1$ = interval(1000);

const sub = interval1$.subscribe(console.log);

setTimeout(() => sub.unsubscribe(), 5000);

}

After 4 ,we are not getting values in console. This confirm our understanding of unsubscribe, we no longer get values in subscribe.

Now notice one thing we do always get here subscription object from calling subscribe and we can call unsubscribe on it but we really have no guarantee that, this is going to affect observable, that is going to depend on type of observable. for example our http Observable is clearly not cancellable.

export function createHttpObservable(url: string) {

return Observable.create(observer => {

fetch(url)

.then(response => {

return response.json();

})

.then(body => {

observer.next(body);

observer.complete();

})

.catch(err => {

observer.error(err);

});

});

There is no logic here to cancel ongoing HTTP request. So lets add cancellable logic to it. Fetch api has support for cancellation, so we can create cancellation of this request by using what is called an **abort controller**. Lets create it and see how it is used.

We instantiate new abort controller. This is part of fetch api. Then abort controller provides us what is known as a signal. This is abort signal that if it emits value of true then the fetch request is going to be cancelled by the browser. So we can take the signal and pass it here in a configuration object. which is second argument of fetch call. we can cancel the request by calling controller.abort.

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 => {

return response.json();

})

.then(body => {

observer.next(body);

observer.complete();

})

.catch(err => {

observer.error(err);

});

controller.abort();

});

}

But we do not want to do this here, in body of definition of this observable. what we want to do is , we want to call abort only if we unsubscribe . so how do we make this connection. Note when we pass function to Observable.create, we are not returning any value. we do have option of returning a value out of this function. this value returned should be a function and this is cancellation function. this function is going to be executed by our application via the unsubscribe method. So unsubscribe is going to trigger this function that we return here as a result of creating our observable. so we return this function. code-

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 => {

return response.json();

})

.then(body => {

observer.next(body);

observer.complete();

})

.catch(err => {

observer.error(err);

});

return () => controller.abort();

});

}

Lets test it out-

About.component.ts-

ngOnInit() {

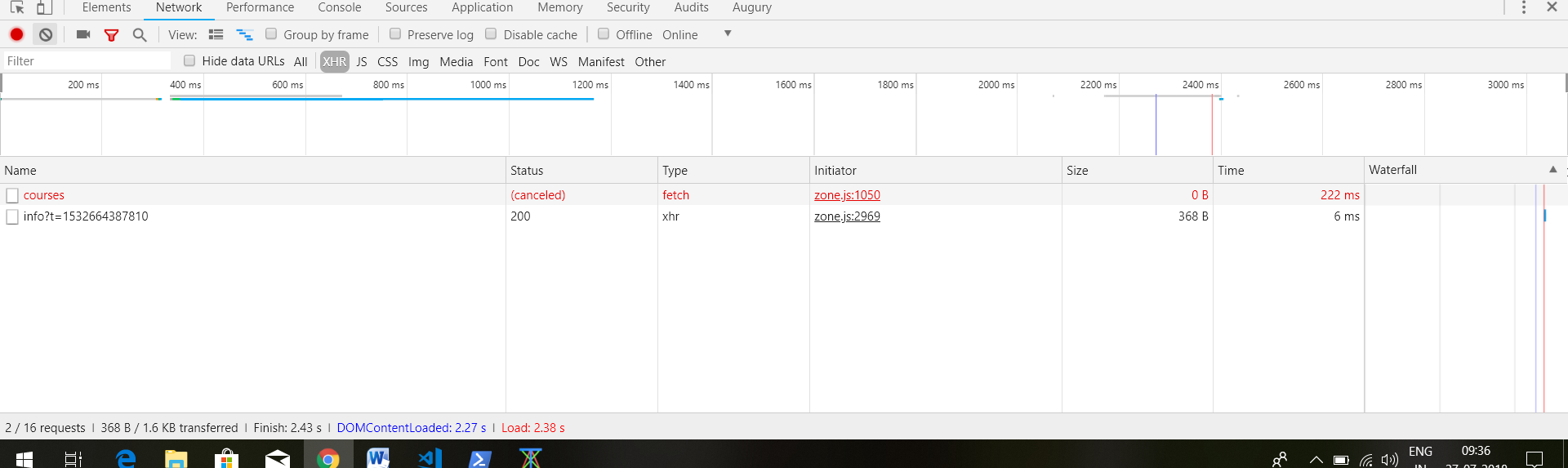
const http$ = createHttpObservable('/api/courses');

const sub = http$.subscribe(console.log);

setTimeout(() => sub.unsubscribe(), 0);

}

So we unsubscribe with delay of 0. So tis way browser will have an opportunity to trigger the ajax request but ajax request is going to be cancelled straight away. We are not going to give time for response to come back from server. The result is that we should see in network tab of the dev tools a cancalled http request.



As we can see that our implementation of http observer now implements cancellation. Now lets give practical example where this functionality will be useful.

Question-

1)about use of signal

<https://www.udemy.com/rxjs-course/learn/v4/questions/5954558>

21)Setting up the course component

Here we will cover **switchMap** operator. This operator is very commonly used for implementing search type aheads which is exactly what we are going to implement in next few lessons. Lets go to courses.component. this component is loaded whenever we click on view course button. In this component we have type ahead which is used to search lessons in a course.

Here we make 2 calls to api to get course image and then lessosns of code-

Course.component.ts-

export class CourseComponent implements OnInit, AfterViewInit {

course$: Observable<Course>;

lessons$: Observable<Lesson[]>;

constructor(private route: ActivatedRoute) {

}

ngOnInit() {

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

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

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

.pipe(

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

);

}

}

In html we subscribe to these observables like this-

Course.component.html-

<div class="course">

<ng-container \*ngIf="(course$ | async) as course">

<h2>{{course?.description}}</h2>

<img class="course-thumbnail" [src]="course?.iconUrl">

</ng-container>

<mat-form-field class="search-bar">

<input matInput placeholder="Type your search" #searchInput autocomplete="off">

</mat-form-field>

<table class="lessons-table mat-elevation-z7" \*ngIf="(lessons$ | async) as lessons">

<thead>

<th>#</th>

<th>Description</th>

<th>Duration</th>

</thead>

<tr \*ngFor="let lesson of lessons">

<td class="seqno-cell">{{lesson.seqNo}}</td>

<td class="description-cell">{{lesson.description}}</td>

<td class="duration-cell">{{lesson.duration}}</td>

</tr>

</table>

</div>

Here see how we are subscribing to the observables in html. Here we re using \*ngif with observables.

Now we have everything that we need to build our type ahead logic. The main type of operator for this type o logic will be switchMap but we will also be learning several new operators including debounce time, distinctUntilChanged etc.

22. Building a Search Typeahead - debounceTime and distinctUntilChanged Operators

Here we will see couple of new operators as start to implement our type ahead logic. We are going to be talking about **debounceTime** and **distictUntilChanged** operator. lets start implementation of our type ahead operator.

here we created a stream of keyup events on input box.

First in ngAfterViewInit we defined a observable built from keypress events .Here in order to compile this code we have to define type which will be returned here to mapping function(value that is passed as argument to function used in mapping). here we simple specify any type. if we do not mention type then ts assume observable return value that is empty object. when we try to access target property then we will get error. -

ngAfterViewInit() {

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

.pipe(

map(event => event.target.value)

)

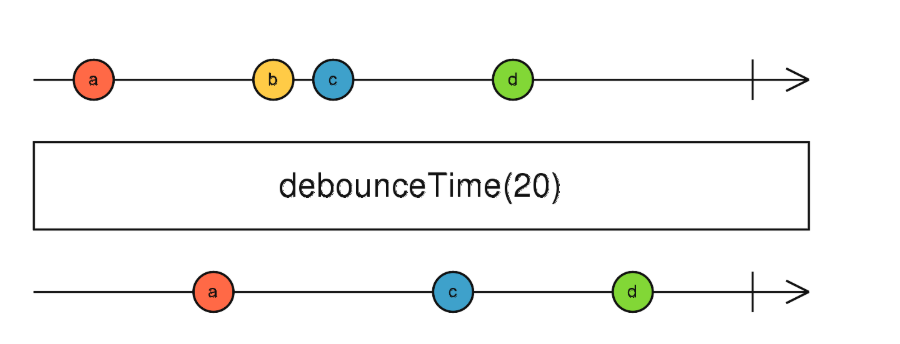
.subscribe(console.log);

}

Now when we type we can see we see each value in console like this-



Here we are getting duplicate values, this is because we presses shift. Lets ay we were taking the stream of values, without any furher processing and we will convert them into http request using for example **concatMap**. We can see that would lead to huge number of request one per each key type and there are also duplicate requests. That we want to avoid. What we want to do is we want user input search to stabilize before emitting any search and we also want to avoid sending duplicate searches. First lets reduce number of request to backend. For this we will sue **debounceTime** operator.



In input stream a is emitted, note this value is not immediately reflected in output. Intead we are going to start here counting 20 ms, during these 20 ms we are going to wait to see of source observable emits some other value, the value a is only going to be passed to output if it remains stable for atleast 20 milliseconds meaning that after a value gets emitted we need to wait for a period of silence of atleast 20 ms before considering a as stable value and adding it to output. Lets see another example.

New value b gets emitted and we started counting another new interval of 20 ms, during those 20 ms we have another value c. because delay between b and c is less than 20 ms. B is not considered as stable value . so it is discarded in output. Notice that we won’t output c directly in the output observable instead we are going to count from emisision of C another new interval of 20 ms. We are going to see if C value is stable. If 20 ms passed and value remains unchanged then c value is considered stable and is added to the output. Now we want to eliminate duplicate values for this we use **distinctUntilChanged** operator.if we have 2 consecutive values that are exactly the same, we will have only one value.that is last value is then ignored. Code-

ngAfterViewInit() {

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

.pipe(

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

debounceTime(500),

distinctUntilChanged()

)

.subscribe(console.log);

Lets now see what would be best operator that we could use in order to turn the search value that we type in input box into a backend request. We are going to see previous operators concapMap, MergeMap and exaustMap are not good choice for this.

23. Finishing the Search Typeahead - The switchMap Operator

Here we are going to take stream of values from input box and we are going to convert in into backend request. We define function loadLessons for it. On load of page we call this function without any value. this function has a default value. so when we dnt pass any argument then this default value is used.now we need operator to convert our keypress events into http requests. Lets try to use known operators first. Lets first try **concatMap** operator. Code-

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

} from 'rxjs/operators';

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

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

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

@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}`);

this.lessons$ = this.loadLessons();

}

ngAfterViewInit()

{

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

.pipe(

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

debounceTime(400),

distinctUntilChanged(),

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

)

.subscribe(console.log);

}

loadLessons(search = ``) {

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

.pipe(

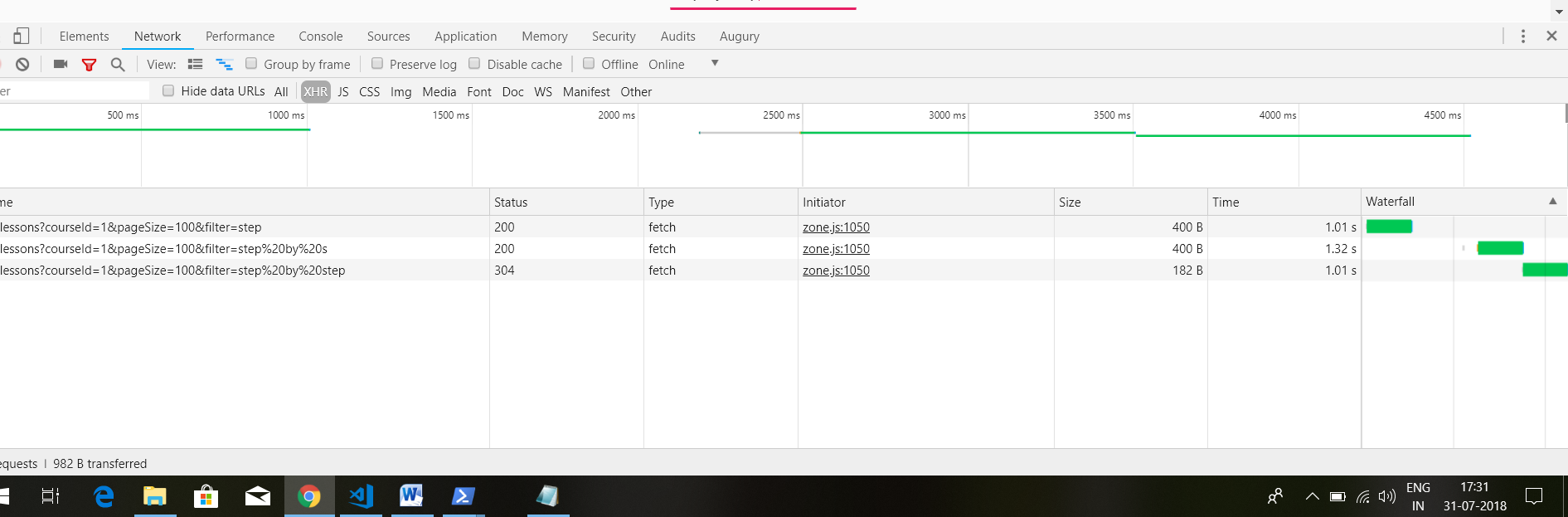
map(res => res['payload'])

);

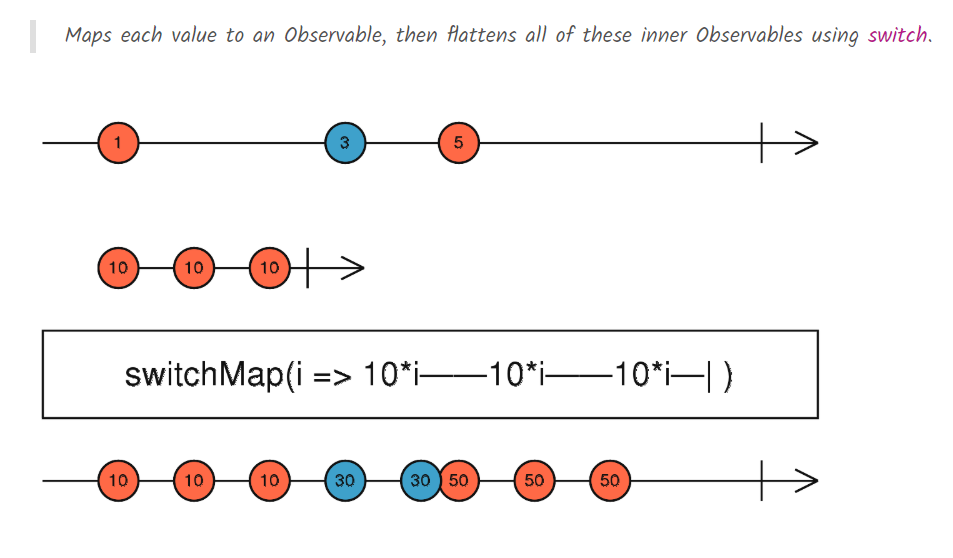
}

}

Type something and see network tab-



As we can see that we emitted one request after another as expected by using concatMap operator. Even if we are using debounceTime and distinctUntilChanged to reduce the number of search request to backend, we would prefer that if user typed in a new search request while current search is still ongoing, we would prefer that current search gets immediately cancelled and then we make new search. The main logic that we are looking for is cancellation**. We want to cancel the ongoing http request immediately and then switch to new http request**. So instead of concatMap , we need switchMap operator. Lets see how it works-



Here we have input observable that is emitting values- 1 ,3 and 5. Thenw e have mapping function passed to switchMap, which takes one value of the stream and creates an observable. so this example above is taking the value multiplying it by 10 and emitting it three times and then completing. Now have a look at what happens in output of switchMap. What is happening here is that we are taking the input value(1 for example) , creates a new observable(which emits 10 3 times) and we subscribe to it and we start emitting values here to output observable(observable creates after applying switchMap operator to source observable) as they arrive over time. but take a look at what happens whenever we complete the initial observable(which emits 10, 3 times). Well if that’s the case and the values have been already all emitted we won’t see any difference here in output. but now take a look in new value here , a new value arrives(3), we create a new derived observable and we start emiiting the value to output. Se have first value in output, then second value but now unlike what happened in case of value 1, before third value gets emiited, what happens is that the source observable emits another value(5). **What switch map is going to do is it unsubscribes from the second observable that was emitting three 30 values.** It is going to unsubscribe from that observable and instead its going to switch to this new observable, that is created here with 3 value’s of 50. So now we start emitting 3 values of 50 until this third observable completes. Whne source input stream completes then the switchMap output stream also completes. So switchMap is all about this unsubscription logic.

As we can see that if we have a made a http request and then we change our search term. Then we want to cancel our ongoing http request and make new http request.we implemented logic to cancel our http request on unsubscription.so we can use switchMap operator to achieve desired result. As we know upon emit of new value switchMap will unsubscribe from previous observable(http observable) and subscribe to new observable. This is what we want in our application. When we unsubscribe from our http observable, we also cancel corrosponding http request(we have implemented logic of it) So lets use it. Code-

Courses.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

} from 'rxjs/operators';

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

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

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

@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}`);

this.lessons$ = this.loadLessons();

}

ngAfterViewInit()

{

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

.pipe(

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

debounceTime(100),

distinctUntilChanged(),

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

)

.subscribe(console.log);

}

loadLessons(search = ``) {

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

.pipe(

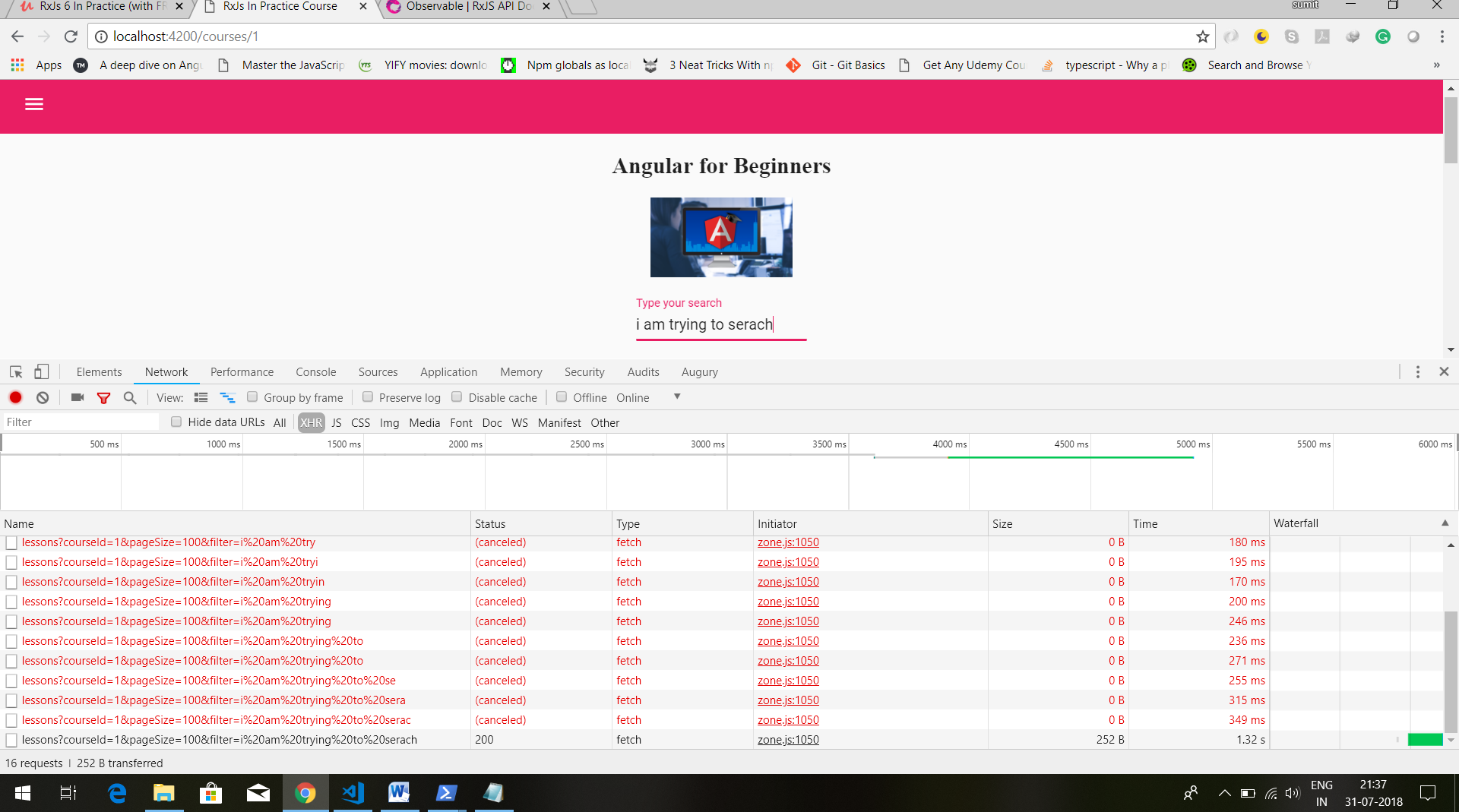
map(res => res['payload'])

);

}

}

In network tab-



Now we can see cancelled http requests. This means that this search was aborted and server no longer needs to handle it. our backend can then abandon the search here, that is no longer needed by client.

Here we can see that our request are cancelled. Note that switchMap observable is just unsubscribing from source observable. we have implemented our httpObserver like this- on subscription, http request is cancelled.

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 => {

return response.json();

})

.then(body => {

observer.next(body);

observer.complete();

})

.catch(err => {

observer.error(err);

});

return () => controller.abort();

});

}

All this is fine, but we are not updating the courses as result of search operation. What we want to do is we want to make lessons$ observable combination of 2 different observables.

We initially want it to have this value-

this.lessons$ = this.loadLessons();

this is initial lessons observable. then when start typing in search text box, we want it to be output of this stream-

ngAfterViewInit()

{

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

.pipe(

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

debounceTime(100),

distinctUntilChanged(),

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

)

.subscribe(console.log);

}

So what we want to do here is to combine 2 different observables and the logic that we want to apply here is concatenation. we want the first initial observable to get loaded and only then should we start responding here to user searches. So first we define these observables in separate variables , then we combine the using concat. 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

} from 'rxjs/operators';

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

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

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

@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()

{

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'])

);

}

}

Herew e will get error that Observable<{}> is not assignable to Observable<Lesson[]>. To remove that error assign return type to loadLessons functions. This is to make sure that concatenated observable is compatiable with type of lessons$ observable which emits values of type lessons array.

In this way concatenation of 2 observable of this type(type which is returned by loadLessons) is going to be compatible with lessons observable.

I have asked questions-

<https://www.udemy.com/rxjs-course/learn/v4/questions/5963164>

Here see the use of concat operation. Now that we have good overview on how the multiple mapping operators work. Lets talk about error handling. What happens if a observable errors out. As we discussed before an observable can either complete or error out. In both cases it will not emit further values. Lets see some of common strategies for doing error handling.