1)Helicopter View

We are going to learn observable patterns and we are going to learn numeras rxjs operators. This practical guide for rxjs library. Rxjs has steep learning curve. What we need to do with rxjs is, w eneed to first introduce a couple of baseline concepts. That is whatw e will do in this course. We will cover notions of streams and observables. In order to make sure that w eundrstand rxjs from begibubg we are going to start by implementing our own custom http observable. this observable is going to be used throughout this course to build our application and it isgoing to support fetaures like error handling and cancellation.

Then we will dive into observable operator. we will cover more advanced observable combination strtaegies. We will cover merge, switch, exhaust and concat strategies. We will cover corresponding map operators- switchmap, mergeMap, exhaustMap and conactMap.

Then we will cover Rxjs error handling in detail. We will cover several rxjs errorHandling strategies such as catchandRecover, catchAndThrow and retry.

After that will expand rxjs catalog by covering commonly needed operators such as ForkJoin and very commonly used operations like throttling and debouncing.

After that w e will be creating our own custom rxjs operator from scratch. We will create debig operator that is going to allow us to troubleshoot our rxjs operators. It will outpt messages on console w=which will help us to understand rxjs programs. These messages will have different logging levels. This could be info, error, debug or trace.

Then we will dedicate whole section to rxjs subjects. W will cover more specialized subjects like behaviour subject, async subject and replaced subjects. We are going to take one of these specialized subjects and we are going to use it to implement a very common recative design pattern, which is centralized store pattern. We are going to implement store from scratch and we are going to refactor our existing application to use centralized store pattern. Thenw e are going to learn about some of its advantages. Then we will again introduce new rxjs operators that will come very handy for dealing with long running observables such as for example this store observable.

In the end of course you will know how to build applications in reactive styles. you will be familiar with core concepts surrounding rxjs this includes observers, observables, subjects and stream

3. Environment Setup - Get the Lessons Code Up and Running

Open this-

<https://github.com/angular-university?utf8=%E2%9C%93&q=rxjs&type=&language>=

here search rxjs. Open rxjs-course. This is repository for all code in this lecture. It has multiple branches, each branch corresponds to a particular sections. The code which you will need at beginning of section. you can either clone a branch using git or you can download the zip file. after downloading run npm install. We are going to be running here locally a small development server that contains a very simple rest api, that is going to serve us some data concerning courses and lessons. Data is coming from db-data file. to strat small rest api server run-

**npm run server**

now we need to run our front end server, so run this command in separate api-

**npm start**

after it finishes, pen localhost:4200 in browser.

4)Understanding Rxjs –what are streams?

Here we will saw what are stream of values.

Here we saw stream of values from mouseclick, set Interval,setTimeout. Here we saw that some streams can complete while some will not complete.

ngOnInit() {

document.addEventListener('click', evt => console.log(evt));

let count = 0;

setInterval(() => {

console.log(count);

count++;

});

setTimeout(() => console.log('Fiished....'), 3000);

}

5)what is Rxjs and what problem does it solve

Here we will start off where we left in last lecture. We saw 3 examples of 3 separate streams-

ngOnInit() {

document.addEventListener('click', evt => {

console.log(evt);

});

let counter = 0;

setInterval(() => {

console.log(counter);

counter++;

}, 1000);

setTimeout(() => {

console.log('Finished....');

}, 3000);

}

Well one typical thing that we want to do is, that we may want to combine these streams. We may want to do this after user clicks on certain part of screen we wait for some time, only after this wait period is over we will start emitting interval. Now in order to do that we have to nest these callback togather , creating what is known as callback hell. Code-

ngOnInit() {

let count = 0;

document.addEventListener('click', evt => {

setTimeout(() => {

setInterval(() => {

console.log(count);

count++;

});

}, 3000);

});

}

So as you can see our typical js front end program is really a program that intends to combine multiple types of streams, streams caused by user interaction, by backend request, by timeout intervals, all those asynchronous events. We want to combine them togather in order to produce final result. So all these streams have natively defined at the level of browser , these callback api, as we can see that is not very convenient, if we are combining multiple streams of values togather. So we are going to see a better way of combining streams other than using native callbacks. We are going to introduce RXjs Libbrary. Rxjs stands for **reactive extensions for javascript** and it is library that makes it very simple to combine streams of values togather in a maintainable way. it’s an extension to standard js because the plain javascript version that we have in our browsers only has these callback interface that really does not scale well in complexity , we start to have lot of nested code.

6)What is an Rxjs Observable? A simple explanation

In previous lecture we defined steam of values which was an interval over the time that was emitting values, 0,1,2,3. Lets now create here using RXjs , definition of that stream. Notice that it’s not instance of the stream, it’s a definition of that stream, declaration of that stream. We can do that using interval method of Rxjs.

$ at the end means that this variable is Rxjs observable.

const interval$ = interval(1000);

now you can see on console, no value is printed. The first thing is we are not specifying what to do with values, we do not have consol.log in this program but also this interval variable, interval$, this is not actually a stream of values, this is definition for a stream of values. It’s like blueprint for how the stream will behave if we instantiate it. So lets here create couple of different interval streams. An observable will only become the stream if we subscribe to it and we do that by using subscribe method.

const interval$ = interval(1000);

interval$.subscribe(val => console.log(`Stream 1 ${val}`));

So once we subscribe to observable , we have create stream of values. We receive these values via subscribe method.

A defination of stream is observable. Here we created this definition(observable) using rxjs.

Now lets create second stream, so that we can see that indeed this interval$ variable which is observable of number is indeed a simply template blueprint for creating new streams.

const interval$ = interval(1000);

interval$.subscribe(val => console.log(`Stream 1 ${val}`));

interval$.subscribe(val => console.log(`Stream 2 ${val}`));

now we have 2 streams each emitting values in its own sequence. Rxjs provides many other functions such as interval for defining other types of streams. Lets see timer method-

const interval$ = timer(3000, 2000);

interval$.subscribe(val => console.log(`Stream 1 ${val}`));

interval$.subscribe(val => console.log(`Stream 2 ${val}`));

this observable will wait for 3 seconds, then it will start emitting values at interval of 2 seconds.

Now les define new stream that we have define before using native api’s. we are going to define here click stream. We can define stream of clikcs using **fromEvent** method of rxjs.

const clicks = fromEvent(document, 'click');

first argument is source of click and second argument is event that we are subscribing to .it returns us only defination of stream, this is not an instance of stream. We are not specifiying what we to do with stream values, we have not yet instantiated a stream. When we subscribe to it , then we are going to get values emitted by these observables.

const click$ = fromEvent(document, 'click');

click$.subscribe(evt => console.log(evt));

here type of click$ is Observable<Event> which means observable of type browser event.

So now we understand difference between observable and streams. Lets introduce core rxjs concepts, lets go deeper into subscribe and we are going to talk about stream errors and streams completion.

7)3 core concepts – Errors, Completion, and Subscirptions

We learnt that observable is blueprint for stream. we can derive concrete istances of streams from observable by calling

Lets see what we can do with subscribe method. We can have couple of more arguments that can be passed to it. First argument is callback which is executed when we get a value, value is passed as argument to this callbcak. Second argument is error handler. Some streams, might go wrong. Let’s say request to http call, it may have error. and in that case we might want to do some error handling logic in order to try for example recover from error.we will do that here at level of this callback, which is error callback for this subscription. Third argument is also a callback, it is executed when stream has terminated itself, it will no longer emit further values, but we want to report that back to user of stream that this would be last value. note that if streams error out or if it completes, we can no longer emit values using this observable. this gurantee given by observable contract. Cancellaton and completion are 2 exclusive events.

It means if observable errors out no further values will be there. If observable completes, no further values. Some observable may continue to emit values without error or completion, this is fine. but once there is error or completion, no values will be emitted. Code-

const click$ = fromEvent(document, 'click');

click$.subscribe(

evt => console.log(evt),

err => console.log(err),

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

);

Another important notion in what concerns observable is the notion of subscription and cancellation. Like in interval like observable, we might want to unsubscribe from it after some time and no longer handle the values that it might be emitting. For that we need way of unsubscribing from stream of values ,which is given us here by return value of subscribe call. interval$.subscribe gives us a subscription object that we can use to unsubscribe. Lets store result of subscription in a variable. Code-

const interval$ = timer(3000, 2000);

const sub = interval$.subscribe(val => console.log(`Stream 1 ${val}`));

setTimeout(() => {

sub.unsubscribe();

}, 5000);

Here after 5 seconds we will not see any values on console.

8)Learn how observables work under the hood, Build your own HTTP Observable

Here we are going to create our own observable similar to interval or click observable. We are going to make a call to backend to fetch some data(it will fetchus list of courses thatw e will display in courses tab), it will give us great insight into how observables are built and it is going to help us a lot to introduce all rxjs operators that we are going to be introducing after this lessons. Data is coming from backend service. Go to this url to see data-

<http://localhost:9000/api/courses>

here we will use fetch api to make call to our service. This api returns a promise. This promise is going to be evaluated successfully if the request succeeds and its going to fail if request fails due to some fatal error such as for example, network is down. Promise is different from observable. A promise will get immediately executed, once we define it. It is unlike observable , that when it gets defined it does not trigger any request.it will only trigger request in response to a subscription.

Lets see how we are going to turn this http call to backend into a rxjs stream. Lets see how we are going to create a observable that represents these http request. We are going to be creating custom observable for that. We will use **Observable.create** method. This will allow you to create an observable from scratch and this how all observable that we have seen before , such as interval, timer are created by calling **Observable.create** . we take output of this call and assign it to variable. Output of this call is going to be observable that is going to model here network call to our backend. this function returns a observable, that we store in http$. In order to create observable we pass a function as argument to create method. This function is going to implement the behaviour of our observable. In this case , it’s the function that is going to do network fetch. This function takes one parameter which is known as observer. The observer is what is going to allow us to emit new values , error out the observable or complete the observable. So the observable here(Http$), does not allow us to emit values on its behalf, instead we can only subscribe to it and get values from stream of values. The observer on the other hand should be kept private here, to the implementation of the observable and this is what will allow us to either emit a new value for our stream i.e calling next, , or error it ,by calling error, or even complete it by calling complete. The observer is what we use internally to implement observable. The function that we pass as argument to create will only be called when we subscribe to our observable. Code-

About.ts-

const http$ = Observable.create(observer => {

fetch('/api/courses')

.then(response => {

return response.json();

})

.then(body => {

observer.next(body);

observer.complete();

})

.catch(err => {

observer.error(err);

});

});

Response.json returns the promise containing the response payload. So if we want to retrieve response body containing all the courses from this response payload, we need to call response.payload . this gives us a promise so we return it. Then we use promise chaining. In second then call we get json body of response and not the response object itself. Then we emit that json as value emitted by this observable. we use next method, which is used to emit value in the observable. after emitting this value then we complete the observable. this is because we know we will not emit any further values. We also handles error by using catch. In catch block we emit error from observable.

So with this we are making sure that we are respecting the observable contact. We are either comleting the observable or we call error. When we implement our own observables, make sure that you follow the observable contract. We will break contract, if we emit value after calling complete on observer. This is absolutely essential for other operators that we want to combine with http , to work.

This is initial implementation of our http observable. we are going to be coming back here and add unsubscription logic and error handling.

Lets now subscribe to it. If we do not subscribe to it , then no http call will be made.Code-

About.ts-

ngOnInit() {

const http$ = Observable.create(observer => {

fetch('/api/courses')

.then(response => {

return response.json();

})

.then(body => {

observer.next(body);

observer.complete();

})

.catch(err => {

observer.error(err);

});

});

http$.subscribe(

courses => console.log(courses),

() => {},

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

);

}

Here we are doing anything in error handler, to make it more readable we can use rxjs noop(no operation needed). It is more readable and it same as passing empty function.

http$.subscribe(

courses => console.log(courses),

noop,

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

);

Run this function, you will see output.

now you might say, why we did all this. What was need to create observable , when we already had fetch api that returns promise. So what is advantage here. The advantage is that we can now use all our RXJS operators to easily combine our HTTP steam with other streams such as click handlers, timeouts, other Http Requests. We have whole other world of operators that makes it really simple to combine stream of values.