33)Subjects and Stores –New Section Kickoff

In this section we will cover store observable pattern and we will be implementing our own custom store from scratch. We will first introduce notion of rxjs subject and we will talk about different types of subjects. We are going to be covering a plane subject, behaviour subject,async subject and replay subjects. We are then going to cover a few more rxjs operators that are commonly used togather with store pattern. First we need to grab the store code. Go to github repo, then go to this branch -2-store. Go to this url-

<https://github.com/angular-university/rxjs-course>

this code is download in D:\Angular University\1)Rxjs In Practice\5)rxjs-course-2-store

you can either download zip file, or you can run this command-

**git checkout –b 2-store origin/2-store**

after we have code. Then run server by running this command-

**npm run server**

then start server by running **npm start**.

Lets now introduce the notion of subject. in next lectures we are going to cover what is rxjs subject, whne should we use it and why and what are different types of subjects that are we have avalaible and we will then use one of these subjects to implement our own centralized store following the centralized store pattern. But we won’t be using a 3rd party library. Instead we are going to be writing our own custom store for learning purposes.

34)What are Rxjs Subjects? A simple Explanation

Here we will introduce the notion of rxjs subject. We have kept this notion to the end of the course because notion of subject might easily be misused. It’s better to, as much as possible create our observable using ,for example observable.create (like we have done with our http observable) or with one of many methods avalaible in rxjs such as **fromPromise**  and other methods like **of**, that allow us to create observables directly from a source. However if some of those methods are not convenient or if we run into a source of data that is not easily transformable to observable or if we are doing multicasting of one value to multiple separate observable consumers then we might want to look into notion of subjects.

In our http observable we have very clear separation between the observable, that is getting returned by create method and observer which is parameter here that allows us to either emit a new value using next or complete or error out the observable.

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 => {

if (response.ok) {

return response.json();

}

else {

observer.error('Request failed with status code: ' + response.status);

}

})

.then(body => {

observer.next(body);

observer.complete();

})

.catch(err => {

observer.error(err);

});

return () => controller.abort()

});

}

in this way of creating observable there is very clear separation between the observable and the observer. But there are many situations where this is not very convenient way of creating a observable. and in those situations we can resort to use of subjects as subject is at same time an observer and observable.

Lets create a subject in about.component. now we if explore the api of subject(create a subject and press ., ide will give suggestions) we will see that it has same kinds of methods that observer had. We have next, error and complete method. But we have more than these three observer specific methods. We also have pipe method. so we can pipe the subject togather with any rxjs operator. so a subject, it looks like it is simultaneously an observable and observer. So we can directly emit value with it but we can also combine it with other observables.

About.ts-

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

import {concat, fromEvent, interval, noop, observable, Observable, of, timer, merge, Subject} from 'rxjs';

import {delayWhen, filter, map, take, timeout} from 'rxjs/operators';

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

@Component({

selector: 'about',

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

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

})

export class AboutComponent implements OnInit {

ngOnInit() {

const subject = new Subject();

}

}

Althrough we could use a subject here as a public member variable for example of this component and share it directly with other components of the application , that is usually not a good idea. the subject is meant to be private to part of application that is emitting a given set of data. The same way that here, in our http observable we would’nt want other parts of application to get access to observer.Only this part of program can emit errors, complete the http observable or emit backend response. It would not be a very good idea to share this observer outside of this method.

Now going back to about.component.ts, we can quickly derive the observable from the subject by using below method – **asObservable.** This will give us back an observable that we assign here to a variable that we are going to call series$. So this observable(series$ is emitting the values of subject). This means if we do subject.next(1) and we issue here couple of values, like we issue couple of values like 1,2,3. These value that are getting emitted via subject are also being emitted here in it’s derived observable. code-

ngOnInit() {

const subject = new Subject();

const series$ = subject.asObservable();

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

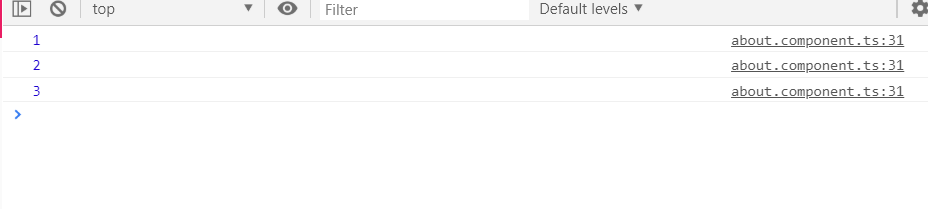
subject.next(1);

subject.next(2);

subject.next(3);

}

Output-



It’s ok to share series$ observable with other parts of application because unlike subject, we do not have here next, error or complete methods that we have in case of subject. So other parts of application will only be able to subscribe to value emitted by this observable, but they wnt be able to emit values on behalf of this observable itself. In order to see that series$ observable behaves like any other observable, code-

ngOnInit() {

const subject = new Subject();

const series$ = subject.asObservable();

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

subject.next(1);

subject.next(2);

subject.next(3);

subject.complete();

subject.next(4);

}

We see values upto 3. Then observable completes and 4th value is not seen.

Idea here is to demonstrate that we have just used the subject to produce a custom observable. as we can see that a subject is very convenient way of creating a custom observable. what we have here(in about.component.ts) is very much easier to understand than using Observable.create(see util.js). there are couple of differences though. We do not have any way of providing unsubscribe logic to our observable that gets derived here from the subject(in http observable we execute a function on unsubscription). And we also run the risk of sharing accidentally the subject with other parts of application which means that those other parts of application could potentially takeover the behaviour of the observable by directly calling next,error or complete on the subject, which is not intended. By this reason we should try to use subject as little as possible. Instead we should try to derive our observables directly from the source as much as possible using methods such as **fromPromise, fromEvent.** So these are preferred ways for creating our own observables by using these RxJs utility methods. However if by some reason that is not practical or even not possible, then using a subject is a great way of creating a custom observable. notice that another very common use case for rxjs subjects is multicasting.

In case of multicasting we want to take one value from one observable stream and remit that into multiple separate outputs streams. As we will see the notion of subjects is going to be essential for us to implement our own custom store solution. Lest know more about subjects. Lets go over what are multiple different types of subjects.

35)Behaviour Subject in Detail- when to use it and why?

We saw how subject is mix between observer and observable. as discussed we should only use a subject to create our own observable(by using asObservable method), if some of observable creation methods are convenient solution for creating stream. However in rare occasions where we will be needing subject, we will probabaly not be using here the default plain subject. We will be most likely using behaviourSubject, which is very similar to plain subject but it also supports late subscriptions. Lets give an example of a late subscription.

Here we are doing early subscription-

ngOnInit() {

const subject = new Subject();

const series$ = subject.asObservable();

series$.subscribe(val => console.log('Early Subscription:' + val));

subject.next(1);

subject.next(2);

subject.next(3);

setTimeout(() => {

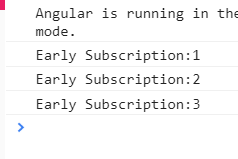
series$.subscribe(val =>

console.log('Late Subscription:' + val);

}, 3000);

}

Output-



Here we are defining subject, then from subject we derive a observable. we are doing here a initial early subscription. The subscription comes along before any value gets emitted.

Now after emitting some value we make second subscription which is late subscription. We have not getting any value in late subscription. So the default bahviour of this plane subject is following-

We are going to subscribe to the subject and we will receive new values that are made after subscription. But we will not get access to previously emitted values. This subject here by default has no memory. After second subscription if we emit some value, then that value is going to be received by both subscriptions. This is because that value was emitted after we have made both subscriptons. The problem is that when we are writing a asynchronous programs we very often want our late subscribers to receive something from observable. typically we want our late subscribers to receive the latest value emitted by the observable.

Let’s say for example that observable corresponds to a http request and we have here a late subscriber to that http request. Even though subscription happened after the request has been completed and we got response from backend, we would still want to receive that value.

We want to be able to write our program in a way that our logic still works independently of timming of each subscription. So in order to that we have different type of subject called **behaviour subject**. The goal of behaviour subject is to always provide something to subscribers .even if the subscription happens late we still want to get latest value emitted by observable before subscription. So because the goal is to always a value to subscribers , we also need to pass a initial value , while creating behaviour subject.

This is the value that is going to be given to this subscriber here(in first subscription) because the subscriber performed he subscription before the first value gets emitted. After that value emitted by subject will be passed to all subscriptions. Another difference is that late subscriptions will get the last value emitted by subject. Code-

ngOnInit() {

const subject = new BehaviorSubject(0);

const series$ = subject.asObservable();

series$.subscribe(val => console.log('Early Subscription:' + val));

subject.next(1);

subject.next(2);

setTimeout(() => {

series$.subscribe(val =>

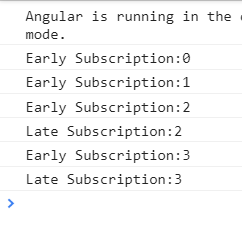
console.log('Late Subscription:' + val);

subject.next(3);

}, 3000);

}

Output-



Now lets learn how behaviour subject handles completion. So if completion happens before second subscription takes place then late subscribers will not get any value.code-

ngOnInit() {

const subject = new BehaviorSubject(0);

const series$ = subject.asObservable();

series$.subscribe(val => console.log('Early Subscription:' + val));

subject.next(1);

subject.next(2);

subject.complete();

setTimeout(() => {

series$.subscribe(val =>

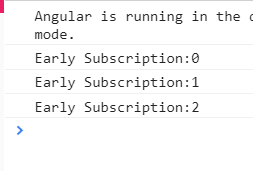
console.log('Late Subscription:' + val);

subject.next(3);

}, 3000);

}

Output-



Completion means tht late subscribers will not receive the last emitted value.

So this logic that the behaviour subject has of remembering the last emitted value will only be effective as long as observable is running and it will not work after completion. Behaviour subject is probably the most commonly used type of subject and it is one that we would be using to implement our store. But before that lets cover some other varieties of subjects that you may come across.

36)AsyncSubject and Replay Subject – Learn the differences

Async subject is ideal to for using with long running calculations.

ngOnInit() {

const subject = new BehaviorSubject(0);

const series$ = subject.asObservable();

series$.subscribe(val => console.log('Early Subscription:' + val));

subject.next(1);

subject.next(2);

subject.complete();

}

Where we have here observable(subject variable in code above) that is emitting here a lot of intermediate calculation values. Lets say that calculaton is ongoing and we are progressively reporting here(subject.next(2)) the latest, most update value of calculation but the calculation is not finished yet. When the calculation is finished, then the last value of the subject is going to be emitted and then subjects is going to be completed(subject.complete()). In this scenario we really don’t want to receive the intermediate values of calculation. Instead what we would like to receive is the last value that got emitted just before the completion. So that’s the final value of calculation. In order to implement that use case we can use AsyncSubject. AsyncSubject will wait for observable completion before emitting any of values to the multiple subscribers. The value emitted is going to be the last value. code-

ngOnInit() {

const subject = new AsyncSubject();

const series$ = subject.asObservable();

series$.subscribe(val => console.log('Early Subscription:' + val));

subject.next(1);

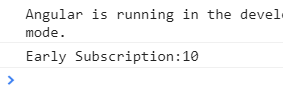
subject.next(2);

subject.next(10);

subject.complete();

}

Output-



So subscriber does not receive intermediate value. it only receive last value. if we never call complete on subject i.e if subject never completes, then subscriber will not receive any value.

Now lets see what happens if we add a second late subscriber. Code-

ngOnInit() {

const subject = new AsyncSubject();

const series$ = subject.asObservable();

series$.subscribe(val => console.log('Early Subscription:' + val));

subject.next(1);

subject.next(2);

subject.next(10);

subject.complete();

series$.subscribe(val => console.log('Thirds subscriber:' + val));

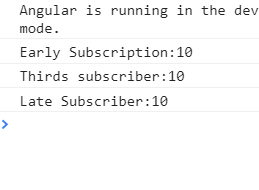
setTimeout(() => {

series$.subscribe(val => console.log('Late Subscriber:' + val));

}, 2000);

}

Output-



So its no matter when we subscriber to this observable, we will always receive last value.

So we see that asynce subject is useful for long running calculations where we only want to receive the last value. there are other situations however where late subscribers want to receive the complete stream from beginning. Lets say subscriber in setTimeout might want all value emitted by observable from starting. So to have this behaviour we can use ReplaySubject. Code-

ngOnInit() {

const subject = new ReplaySubject();

const series$ = subject.asObservable();

series$.subscribe(val => console.log('Early Subscription:' + val));

subject.next(1);

subject.next(2);

subject.next(10);

subject.complete();

series$.subscribe(val => console.log('Thirds subscriber:' + val));

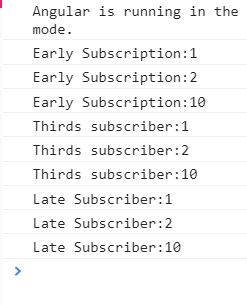
setTimeout(() => {

series$.subscribe(val => console.log('Late Subscriber:' + val));

}, 2000);

}

Code-



ReplaySubject as the name suggest is going to replay the complete observable to all late subscribers and notice that this logic is not related to observable completion. So we do not have to wait for observable to complete, for late subscribers to have all values. Even if we remove subject.complete() line from above observable, our output will not change. If subject emits new value after late subscription, then new value will be broadcasted to all subscribers like normal subject does.

Now let’s start with our custom store .lets start by deciding what subject do we need to use.

37)Store Service Design – what subject to use  
In the last few lessons we have introduced several types of our ex-GI subjects.

We are now going to put subjects to practical use by implementing a very common design pattern in our application.

We are going to be implementing a centralized store. In order to understand what are the benefits of this approach.

Have a look here at our home component that is displayed here.

code-

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

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

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

import {catchError, delay, delayWhen, finalize, 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[]>;

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(),

retryWhen(errors =>

errors.pipe(

delayWhen(() => timer(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'))

);

}

}

As you can see with the current design of the home component every time that navigate into it by using here the navigation menu we are going to trigger here in New HTP requests so we will be fetching our course data from the back end again and again let's now confirm that that is indeed the case. Reload the page and see network tab. so this data did not change.

We are just fetching it back because we have not kept it in memory on the client so whenever we navigate between two routes and we discarded and recreated our home component we lost this data that we had here.

We would like to avoid that we have to make these requests constantly to the server.

Instead we would like to be able to store the data here on the client side independently of the home component so whenever the home component gets discarded our data should not get discarded with it. We need a central place in memory on the client to store our data whenever our home component needs the data.

It simply needs to subscribe to it and it's going to receive the latest version of the data.

So we have here an indication of what our design will be.

We are going to design a centralized service that is going to contain our data and that service is going to expose a couple of observables. that service is going to be responsible for fetching the data from the back and at the appropriate moment.

And also it's going to be responsible for storing that data in memory providing it to the rest of the

application in the form of an observable.

Let's then see what the shared observable Service will look like. We're going to start creating it by going here to the common directory we're going to create here a

file We're going to call it store.service.ts. this is going to contain our centralized observable store.

We are going to write this under the form of a class.

So we are going to call the class store and we are going to make it an angular service ,Since this application is angular application. There will be minimal angular related code in this class.

We're only going to need here the injectable decorated.

code-

Store.service-

import { Injectable } from '@angular/core';

import { Subject, Observable, BehaviorSubject } from 'rxjs';

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

@Injectable({

providedIn: 'root'

})

export class Store {

private subject = new BehaviorSubject<Course[]>([]);

courses$: Observable<Course[]> = this.subject.asObservable();

}

we are going to make the service injectable in the rest of the application by specifying here the

providedIn property and we're going to assign it the value of route. this configuration here simply means that there is only one store for the whole application. Then we inject this service in home component. This service is going to contain our data and we will be able to consume the data in the form of observable. now lets define those observable in store class. Lets go to store class, here we are going to define public api of our store.

We are going to define here an observable called courses$ which is going to be of type of observable of course array. So this is where we are going to store list of courses that we see here on the screen(home component) both the beginner courses and the advanced courses they will all be available here to the remainder of the application by subscribing to these courses observable.

The question now is how are you going to define this observable.

This is one of those situations where it's really not convenient to use Observable. create or one of

the existing methods in the RxJs Library.

So this is a good situation for using a subject to create this observable.

Let's then define here a private member variable which is going to be our subject.

We're going to make this private to this class so that only this class has the ability for emitting new values for this observable.

We want to keep that power constrained here inside this class.

We wouldn't want parts of the applications such as for example the home component to be able to emit a list of courses on behalf of the store itself. only the store has the power of emitting new value for this observable(courses$ in store). The subject is just a private implementation detail that we are using to essentially create this courses observable.

So let's start by defining here a plane are RxJs subject and we are going to build this observable(courses$) here by deriving it from the subject just like we did before using that **asObservable** method. With this subject we are going to be able to emit values here for the courses observable which can then be consumed in other parts of the application such as for example the home component.

Let's now discuss what type of subject do we want to use for this store implementation.

It's important for our application that the late subscribers to this observable also get the latest emitted value.

So whenever we navigate throughout the application going to the About screen for example and back to the courses screen we will have each time new instances of the home component created. Each time the component gets destroyed and recreated as we navigate back to the Course's route.

So we wanted the later instances of this component to also get the Course's data.

This means that the subject that we are looking for is the behavior of subject.

This is the subject implementation that is going to ensure that the late subscribers always get the

latest version of the Course array.

So we are going to provide here an initial value for that array which is going to be the empty.

So initially there are no courses loaded in the store. Notice also that we have specified here the parametric type course array in our behavior subject. it is going to help us to write our code in a type safe way. This essentially means that we can only pass to the subject via the next method, only instances of coursearrays .so if we try to in for example an array of numbers we will get a compilation error.

Now that we have designed our store service let's then start implementing it.

We are going to load some data from the back end and we're going to emit it in these observable.

38) The Store Pattern- Loading Initial Data, Selector Methods, Consuming Data

Hello everyone and welcome back in this lesson we're going to continue the implementation of our store.We're going to start by filling our store with some data.

This is going to be at application startup time. when we start the application, We are going to call an initialization method in our store that is going to then do a request to the backend(similar to what we are doing in home component to load course), fetch the data from the back and then emit it to the rest of the application using here using the courses observable.

So in order to trigger the initialization logic of the store we are going to be using our application root component. in the root component ,we are going to inject here our store and then we are going to call an initialization operation on the store.

Let's then implement here the onInit interface and by implementing the ngOnInit method , we are going to be able to call the store.

Let's create here a new value in the store we are going to call it the init method. this method is only going to be called once at application startup time.

App.component.ts-

constructor(private store: Store) {

}

ngOnInit() {

this.store.init();

}

And the goal is that this Method then calls the backend and fetches the list of courses. so we are going to take HTTP observable that we're have defined at the level of the Home component and we're going to remove it from the home component ,the home component will no longer contact the backend directly using HTP observables.

Instead the home component is going to get all the data that it needs from the store.

On the other hand it's the store that needs to call the back and so let's paste in here our HTTP observable.

we are calling here the /api/courses end point and we are receiving here an observable.

store.service.ts-

import { Injectable } from '@angular/core';

import { Subject, Observable, BehaviorSubject, timer } from 'rxjs';

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

import { createHttpObservable } from './util';

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

@Injectable({

providedIn: 'root'

})

export class Store {

private subject = new BehaviorSubject<Course[]>([]);

courses$: Observable<Course[]> = this.subject.asObservable();

init() {

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

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

.pipe(

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

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

// shareReplay(),

// retryWhen(errors =>

// errors.pipe(

// delayWhen(() => timer(2000)

// )

// ))

)

.subscribe(

courses => this.subject.next(courses)

);

}

}

We can for the purposes of demonstrating the store implementation remove these retry logic and we also want do not need shareReplay because this observable(http$) that we are defining here(in init method of store) will not be shared outside of the init method.

Let's then subscribe to this observable and take the output which contains the payload that we are

receiving from the server and emit it here in our courses observable.

Let's start by subscribing to this observable.

Otherwise nothing is going to happen. inside the subscription, we are going to receive the array of courses from the back end and we are going to emit the array here using the Courses observable.

We going to do that using the subject .we are going to call subject.next and we're going to pass in

the least of courses.the home component can then consume the content of the store by subscribing to courses array.

So we could define here a courses observable local variable that corresponds to the observable

provided by the store. [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(),

// retryWhen(errors =>

// errors.pipe(

// delayWhen(() => timer(2000)

// )

// ) )

// );

const courses$ = this.store.courses$;

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 would then apply here a filtering operation and fetch the beginning courses and the advanced courses directly from the store.

This logic that we have here(in home.component.ts) that selects the beginning and the advanced courses is something that

maybe other parts of the application other than the home component might also want to do.

So let's take this and we want to refactor these into the store by creating a selector method.

We are going to create here in the store and new Method that we are going to call selectBeginner courses.

This is going to return as an observable that gives us only to beginner courses and filters out any

other course category. Then we create this method in store. Store.service-

export class Store {

private subject = new BehaviorSubject<Course[]>([]);

courses$: Observable<Course[]> = this.subject.asObservable();

init() {

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

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

.pipe(

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

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

// shareReplay(),

// retryWhen(errors =>

// errors.pipe(

// delayWhen(() => timer(2000)

// )

// ))

)

.subscribe(

courses => this.subject.next(courses)

);

}

selectBeginnerCourses() {

return this.filterByCategory('BEGINNER');

}

selectAdvancedCourses() {

return this.filterByCategory('BEGINNER');

}

filterByCategory(category: string) {

return this.courses$

.pipe(

map(courses => courses.filter(course => course.category == category))

);

}

}

}

The logic that we have here is filtering course category.So this is a very common operation that again other parts of the application might be interested in performing. So we are going to create here an extra method in store service. We're going to call it filter by category.

We're going to give to these Method an input parameter which is going to be the category that we are filtering on and in body we filter out based on category here.

We are going to then use this method to implement the selectBeginner courses.

So this is our first selector method that is implemented using this method.

We can very easily now define an extra selector called selectAdvanced courses.

We can now use these methods here in home component. Code-

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

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

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

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

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

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

import { Store } from '../common/store.service';

@Component({

selector: 'home',

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

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

})

export class HomeComponent implements OnInit {

beginnerCourses$: Observable<Course[]>;

advancedCourses$: Observable<Course[]>;

constructor(private store: Store) {

}

ngOnInit() {

const courses$ = this.store.courses$;

this.beginnerCourses$ = this.store.selectBeginnerCourses();

this.advancedCourses$ = this.store.selectAdvancedCourses();

}

}

Now in network tab you can see that we are sending only one request. We use subject to emit value emitted by http onservable. Then when ever we subscribe to that store observable, we will automatically get last value emited by Behaviour subject.

Let’s look at the current form of the home component.

The component is now much simpler than our initial implementation.

We have here very little logic.

The component is a pure projection of state. We simply define here our observable streams and we consume the data that we need from the store using selector methods.

Let's have a look at these new version of the home component in action and we're going to see that whenever we reload the application we're going to get here in the network tab, a initial courses API request. that got the courses here from the backend.

But now as we navigate through the application and we go back to the home component we see that the component is still getting the courses from the store but we are no longer doing HTP requests to the backend each time.

And this is because the data is now stored in the centralized store. So this means that whenever we recreate the home component we don't have to contact the backend again.

Instead we can simply contact the store and subscribed to the data that we are going to eventually receive , once it arrives from the back end. and with this we have shown what a component looks like that consumes data from the store.

Let's now show what a component looks like, If it needs to modify some of the data already existing in the store.

39)BehaviourSubjectStore – Example of a Data modification Operation

We will complete the implementation of our store. We are going to add a modification operation. In last lecture we added the read operation, there we have home component consume the data from store by using selector methods. Now we will add modification operation which will be used in course-dilaog component. In this component we can edit the course and we would like when we hit the save button for request to be sent to the backend to save the new course data, we would also like that new value of courses observable to be emitted so that the changes that we made to the course in memory are also immediately reflective in an optimistic way in remainder of application while backend save request is still ongoing. Lets then start implementing our save course logic. Lets go to course-dialog component which corresponds to dialog on screen which appears when we click on edit button on course. This is in tis component that we trigger backend save request.

We add a click event handler to save button. On click to save button we call save method. In this save method we are going to call store. We call saveCourse method of store. We will implement this method in store. This method will do following- the first thing that this method is going to do is to save the course in the, in memory store and broadcast the new value of course to all subscribers. So the store is going to be modified optimistically in memory. Then a request is going to be made to backend and course is also going to be saved in our database.

The saveCourse method is going to have 2 arguments. One is id of course which is modified and second argument is value of modified course. This method savecourse is then going to return us an observable, this observable returned will provide us some information about backend save request. So this observable is going to complete successfully , if backend save request went through without a problem. But it is also going to throw an error if something goes wrong. If save happens succssfuly, we want to close this dialog. We want to handle error locally at the level of course-dialog component. In this case we are going to print a logging message. Code-

Course.dialog.compoennt-

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

import {MAT\_DIALOG\_DATA, MatDialogRef} from "@angular/material";

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

import {FormBuilder, Validators, FormGroup} from "@angular/forms";

import \* as moment from 'moment';

import {fromEvent} from 'rxjs';

import {concatMap, distinctUntilChanged, exhaustMap, filter, mergeMap, tap} from 'rxjs/operators';

import {fromPromise} from 'rxjs/internal-compatibility';

import { Store } from '../common/store.service';

@Component({

selector: 'course-dialog',

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

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

})

export class CourseDialogComponent implements AfterViewInit {

form: FormGroup;

course:Course;

@ViewChild('saveButton') saveButton: ElementRef;

@ViewChild('searchInput') searchInput : ElementRef;

constructor(

private fb: FormBuilder,

private dialogRef: MatDialogRef<CourseDialogComponent>,

@Inject(MAT\_DIALOG\_DATA) course:Course,

private store: Store ) {

this.course = course;

this.form = fb.group({

description: [course.description, Validators.required],

category: [course.category, Validators.required],

releasedAt: [moment(), Validators.required],

longDescription: [course.longDescription,Validators.required]

});

}

ngAfterViewInit() {

}

save() {

this.store.saveCourse(this.course.id, this.form.value)

.subscribe(

() => this.close(),

err => console.log('Error while saving the course', err)

);

}

close() {

this.dialogRef.close();

}

}

Lets now implement saveCourse method in our store. First we start by modifying the course in the memory. We are going to change the course with changes(argument to fucntion) and we are going to broadcast the changes to rest of application using our subject. We get refrence to complete array of courses by using subject.getValue. then get the index of course which is modified. We know where that course is in array and we can directly modify it. but we should avoid to do that ,instead of modifying the in memory value directly, we should create here new value of courses array and emit that value using the subject. Idea is that consumers of the courses data will get notified that a new value is avalible. If we mutate the data diretly in memory the comoponents would not know that data has been modified.

Then we create new Courses array that we will be broadcasting as new courses value. we start by taking existing courses array and we are going to create a copy of the array using the slice operator. in this copy here we are going to look for courses that we have modified and we will assign it to new.

The idea here is to avoid mutating any data that was already passed to the componnts. Instead we crate a new courses object. the we broadcast the newCourses using subject. Then make request to backend. Generate observable from fetch api that returns the promise and return that promise.

storeService-

import { Injectable } from '@angular/core';

import { Subject, Observable, BehaviorSubject, timer } from 'rxjs';

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

import { createHttpObservable } from './util';

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

import { fromPromise } from 'rxjs/internal-compatibility';

@Injectable({

providedIn: 'root'

})

export class Store {

private subject = new BehaviorSubject<Course[]>([]);

courses$: Observable<Course[]> = this.subject.asObservable();

init() {

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

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

.pipe(

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

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

// shareReplay(),

// retryWhen(errors =>

// errors.pipe(

// delayWhen(() => timer(2000)

// )

// ))

)

.subscribe(

courses => this.subject.next(courses)

);

}

selectBeginnerCourses() {

return this.filterByCategory('BEGINNER');

}

selectAdvancedCourses() {

return this.filterByCategory('BEGINNER');

}

filterByCategory(category: string) {

return this.courses$

.pipe(

map(courses => courses.filter(course => course.category == category))

);

}

saveCourse(courseId: number, changes) {

const courses = this.subject.getValue();

const courseIndex = courses.findIndex(course => course.id === courseId);

const newCourses = courses.slice(0);

newCourses[courseIndex] = {...courses[courseIndex], ...changes};

this.subject.next(newCourses);

return fromPromise(fetch(`/api/courses/${courseId}`,

{

method: 'PUT',

body: JSON.stringify(changes),

headers: {'content-type': 'application/josn'}

}

)

);

}

}

With this we have completed the implementation of home component using centralize store design. Lets now do a similar refactoring to course component that displays a single course.

40)Refactoring the course component for using the store

Here we will refactor course.component and we will present couple of operators that are useful in context of store solution.

Lets first go with refactoring. Instead of making http request in course component to get course data, we get course data from our store by using selector method. Code-

Course.component.ts-

ngOnInit() {

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

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

this.course$ = this.store.selectCourseById(this.courseId);

}

Course.component.html-

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

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

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

</ng-container>

Store.service.ts-

init() {

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

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

.pipe(

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

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

// shareReplay(),

// retryWhen(errors =>

// errors.pipe(

// delayWhen(() => timer(2000)

// )

// ))

)

.subscribe(

courses => this.subject.next(courses)

);

}

selectCourseById(courseId: number) {

return this.courses$

.pipe(

map(courses => courses.find(course => course.id == courseId))

);

}

Now we are gping to present couple of extra rxjs operators that are useful in handling observables that are derived from store. So what is particular about observable course$(in course.component.ts, now we rare getting it derived from store, instead of making http call in course$ component) which is derived from store? It’s that this observable unlike all other http observables that we have been using so far, this course$ observable does not complete. This is because our courses$ observable in store, from where all data is getting derived by using selector methods, does not complete. This observable here never completes.lets then introduce a couple of new rxjs operators tht are going to help us to handle these particular type of observable.

41)Forcing the completion of long running observables- First and Take Operators

In this lessons we are going to show how to force completion of long running observables, such as for example store courses$ observable. we will see **first** and **take** operators. Lets see a situation where they might be useful. In last lecture we saw that courses$ observable in store never completes. Also all observables derived from it like course$ observable in course component also does not complete.lets see a potential consequence for that. Lets we are using forJoin operator to combine this observable with some other observable. we will never see output of combined observable because course$ observable never completes. This is just a example, there are many other situtaions where we would prefer to have our course observable to complete after emitting first values course.compoent.

ngOnInit() {

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

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

this.course$ = this.store.selectCourseById(this.courseId);

}

ngAfterViewInit() {

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

.pipe(

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

debounceTime(400),

distinctUntilChanged(),

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

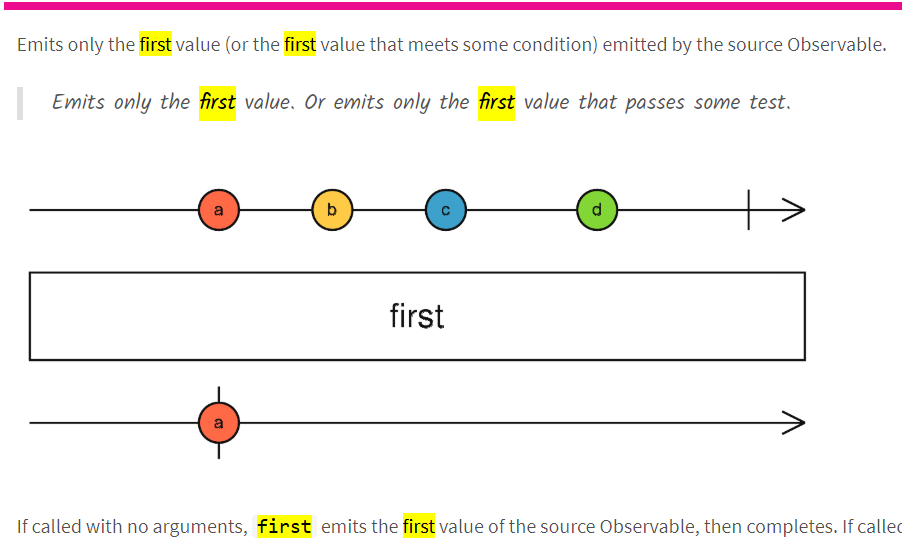
);

const initialLessons$ = this.loadLessons();

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

}

We can use **first**  operator. marble diagram.



This is how we can use it-

ngOnInit() {

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

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

this.course$ = this.store.selectCourseById(this.courseId)

.pipe(

first()

);

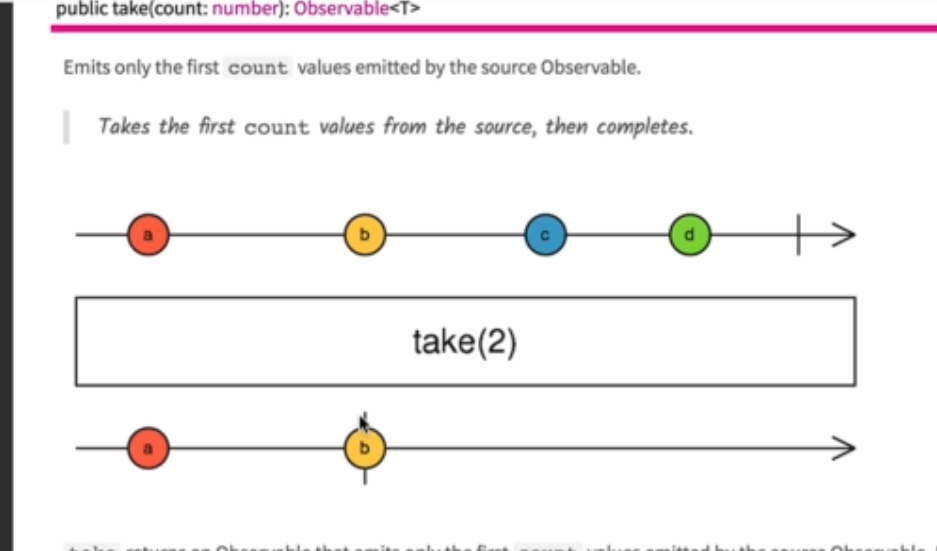
forkJoin(this.course$, this.loadLessons())

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

}

Now we will result output from observable created by forkJoin operator, because w eare completeing course$ observable by using **first** operator.

Another observable that you can use to complete observable is **take** operator.



We also have takeWhile and takeUntil.

Now before moving forward , we want to fix one thing.

Store.service.ts-

import { Injectable } from '@angular/core';

import { Subject, Observable, BehaviorSubject, timer } from 'rxjs';

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

import { createHttpObservable } from './util';

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

import { fromPromise } from 'rxjs/internal-compatibility';

@Injectable({

providedIn: 'root'

})

export class Store {

private subject = new BehaviorSubject<Course[]>([]);

courses$: Observable<Course[]> = this.subject.asObservable();

init() {

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

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

.pipe(

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

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

// shareReplay(),

// retryWhen(errors =>

// errors.pipe(

// delayWhen(() => timer(2000)

// )

// ))

)

.subscribe(

courses => this.subject.next(courses)

);

}

selectBeginnerCourses() {

return this.filterByCategory('BEGINNER');

}

selectAdvancedCourses() {

return this.filterByCategory('BEGINNER');

}

selectCourseById(courseId: number) {

return this.courses$

.pipe(

map(courses => courses.find(course => course.id == courseId)),

filter(course => !!course)

);

}

filterByCategory(category: string) {

return this.courses$

.pipe(

map(courses => courses.filter(course => course.category == category))

);

}

saveCourse(courseId: number, changes) {

const courses = this.subject.getValue();

const courseIndex = courses.findIndex(course => course.id === courseId);

const newCourses = courses.slice(0);

newCourses[courseIndex] = {...courses[courseIndex], ...changes};

this.subject.next(newCourses);

return fromPromise(fetch(`/api/courses/${courseId}`,

{

method: 'PUT',

body: JSON.stringify(changes),

headers: {'content-type': 'application/josn'}

}

)

);

}

}

Here initially, before response of http request came back, we will nt aving any course data. so it means at that selectCoursebyId will return undefined. Lets handle that case by using filter operator-

selectCourseById(courseId: number) {

return this.courses$

.pipe(

map(courses => courses.find(course => course.id == courseId)),

filter(course => !!course)

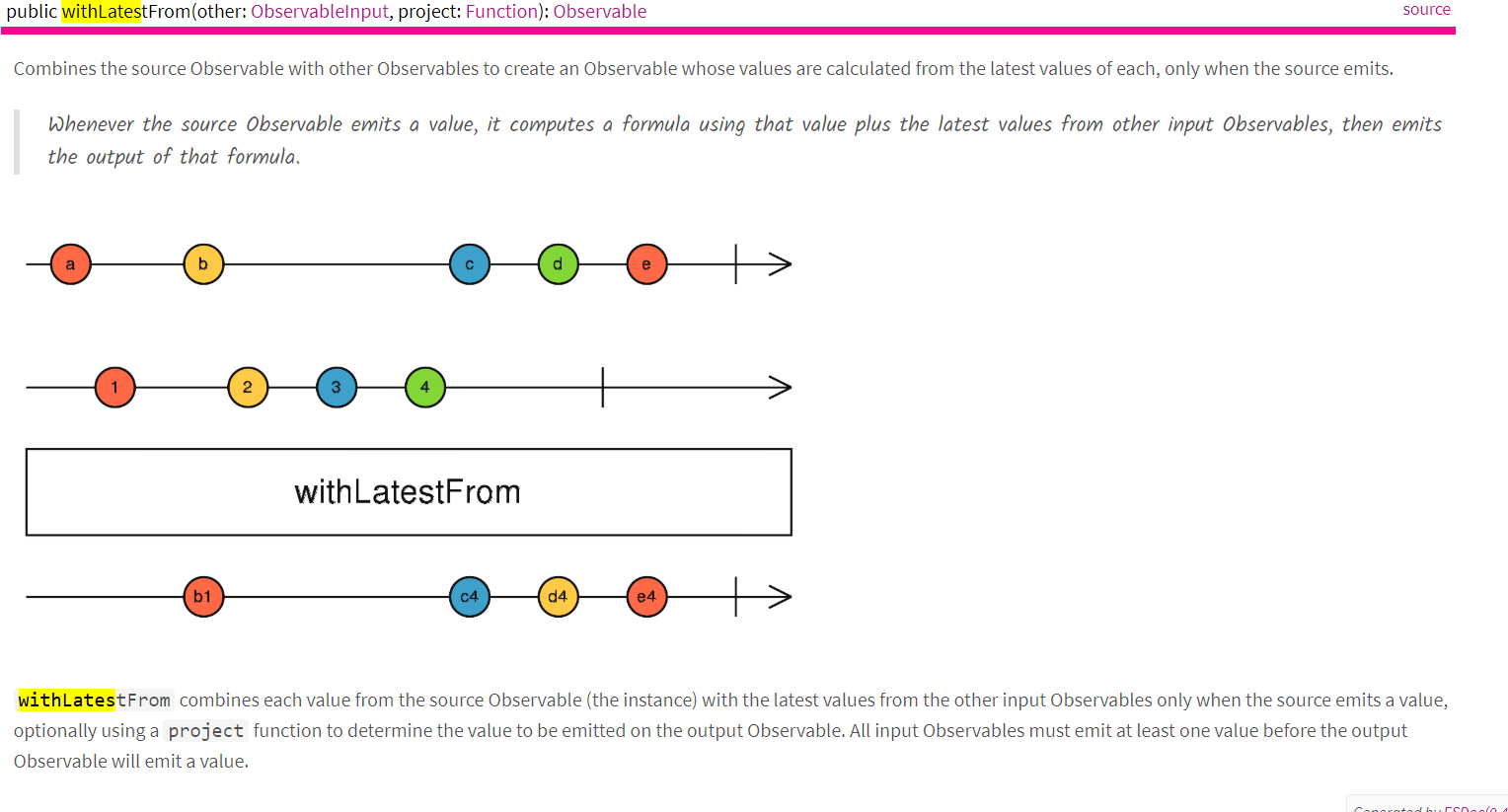
);

}

So we filter out the undefined values.

42)The withLatestFrom RXjs Operator

This operator is useful when dealing with long running observables, like courses$ observable in store.



Lets move to course.component.

Lets see where this operator might come in handy , lets say we are doing backend request and we are calling here loadLessons method. This method returns the observable that emits its first value which is lessons array. lets say we want to take output if this observable and somehow want to combine it with output of course$ observable. so this means in subscribe method we also need data from course$ observable. now course$ observable is not completing. One way to do that can be we can subscribe to course$, store data emitted by it in some variable, then we can use that variable in our subscribe method. But in that way we will declare a extra variable . also we will have some logic in subscribe. We will prefer our components to have reactive design, where we are doing all this annual subscriptions.

In this case we can use with latestFrom operator.

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

we can also provide more than one observable as input to this operator.

lets use it-

this.loadLessons()

.pipe(

withLatestFrom(this.course$)

)

.subscribe(([lessons, course]) => {

console.log(lessons);

console.log(course);

});

In subscribe we will get tupple, which has values from both observables.

With that we covered opertaors that are commonly needed when we are using the store pattern either by implementing it ourselves using behaviour subject or if using a third party library such as ngRx.