We know promises from javascript for using the data stream

In angular it is Observable

These are used to handle the multiple async operations

Where does this data come from?

It can come from a variety of sources, for example, our users interacting with the page through mouse events and key events. They can also come in the form of responses from third-party API HTTP calls. We can also emit data structures into observable streams ourselves. These data structures can be any kind of data, for example, arrays or objects. Sometimes, the data source is referred to as “the Producer”, because it supplies the data stream to the Observable.

This "data over time" concept is great when working with asynchronous, event-based applications.

Here's a real-world metaphor that we can use to get our head around the concept of observables.

Let's say you want to watch a TV show from any of the available online show-streaming services.

What you'll do to watch a TV show online can be summed up in a few steps:

1. Visit the streaming service and locate the show to watch;
2. Click the play button;
3. The show starts to arrive over time, episode by episode;
4. You want to keep binge-watching but you must go to work — so you click the stop button.

**Note**: in this example, you can keep watching if there are episodes available. When there are no episodes available, you will keep listening for new episodes whenever they appear — unless the streaming service has marked the series as over, as we’ll explain below.

In the above scenario:

* the streaming show is the Observable stream;
* the person watching the show is the Observer;
* the act of pressing the play button is known as subscribing;
* the act of pressing the stop button is called unsubscribing;

You are the Observer and the show that's streaming is the Observable. Being the Observer, you can get three different notifications: next, error, and complete.

Whenever the next episode comes through the data stream, you get the next notification.

If there was an error, you'll get an error notification.

Once the entire show stream is complete, you get the complete notification — so you know there are no more episodes to watch and the player stops on its own.

**What is RxJS?**

RxJS is a library that makes it possible for us to work with these streams of data in JavaScript.

In RxJS, the Observer is an Object with 3 built-in methods: next(), error(), and complete(). In RxJS parlance, we say that each episode is emitted from the stream. Thus, whenever a new episode is emitted, the next() method gets called.

If we need to do something each time an episode is emitted, we pass that as a parameter to the next() method.

For example:

next( eatAPieceOfPizza() )

Great, whenever a new episode is emitted, besides watching the stream, we're also having a piece of pizza. This is known as handling the next condition.

If an error occurs, such as our internet connection breaking, we can handle the error condition like this:

error( takeAWalk() )

When the stream is complete, we can read a book, like this:

complete( readABook() )

An Observer is like a person that watches the data stream and — based on what emits from the stream (i.e based on notifications from the stream) — does some predefined action accordingly.

Thus, the Observable delivers (emits) some values, and the Observer knows what to do with them.

What kind of data can be emitted from an Observable stream?

An Observable stream can emit virtually any kind of data:

* primitive values such as strings or numbers;
* click events (or any other event for that matter!);
* responses from HTTP requests, etc.

**Example :**

**Component.ts**

import { UppercasePipe } from '@angular/common';

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

import { nextTick } from 'process';

import {Observable , from,of} from  'rxjs' ;

import {map} from 'rxjs/operators';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  title = 'Observable';

private names=["geetha","lalitha","soaps","pasts",'milk'];

  constructor(){}

  users=new Observable(function(x){

    x.next("i am from observable");

    x.next("i am also from observable");

    setTimeout(function(){

   console.log("i am late ");

    },10000)

    setTimeout(function(){

      console.log("please wait.. ");

       },3000)

  }

  )

  details():any{

    from(this.names).pipe(map(x=>x.toString().toUpperCase())).subscribe(function(res)

    {

   console.log(res);

    })

  }

  buttonclick(){

    this.users.pipe(map(x=>x.toString().toUpperCase())).subscribe({

      next(x)

      {

        console.log(x);

      }

    })

  }

}

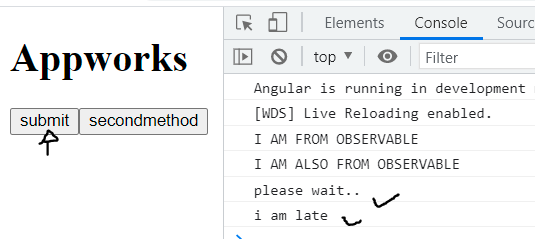
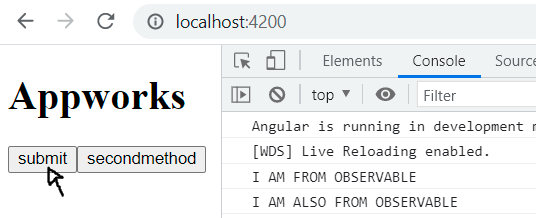
Component.html

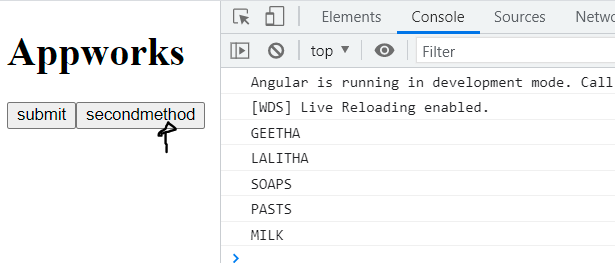
<h1>Appworks</h1>

<button (click)="buttonclick()">submit</button>

<button (click)="details()">secondmethod</button>

o/p





**Unscribes the subscriptions**

https://blog.bitsrc.io/6-ways-to-unsubscribe-from-observables-in-angular-ab912819a78f

@Component({...})  
export class AppComponent implements OnInit {  
subscription: Subscription  
ngOnInit () {  
var observable = Rx.Observable.interval(1000);  
this.subscription = observable.subscribe(x => console.log(x));  
}  
}

What happened if don’t write the unsubscribe code for the subscriptions?

Looking at the above implementation, we called the interval to emit values at the interval of 1sec. We subscribe to it to receive the emitted value, our function callback will log the emitted value on the browser console.Now, if this AppComponent is destroyed, maybe via navigating away from the component or using the destroy(...)method, we will still be seeing the console log on the browser. This is because the AppComponent has been destroyed but the subscription still lives on, it hasn't been canceled.If a subscription is not closed the function callback attached to it will be continuously called, this poses a huge memory leak and performance issue.If the function callback in our AppCompoennt subscription had been an expensive function, we will see that the function will still be called despite its parent being destroyed this will eat up resources and slow down the overall app performance.

So, whenever we use Observables in a component in Angular, we should set up the ngOnDestroy method, and call the unsubscribe method on all of them.

@Component({...})  
export class AppComponent implements OnInit, OnDestroy {  
 subscription: Subscription   
 ngOnInit () {  
 var observable = Rx.Observable.interval(1000);  
 this.subscription = observable.subscribe(x => console.log(x));  
 } ngOnDestroy() {  
 this.subscription.unsubscribe()  
 }  
}

We added ngOnDestroy to our AppCompoennt and called unsubscribe method on the this.subscription Observable. When the AppComponent is destroyed (via route navigation, destroy(...), etc) there will be no hanging subscription, the interval will be canceled and there will be no console logs in the browser anymore

**Multiple subscriptions unsubscribe**

@Component({...})  
export class AppComponent implements OnInit, OnDestroy {  
 subscription1$: Subscription  
 subscription2$: Subscription ngOnInit () {  
 var observable1$ = Rx.Observable.interval(1000);  
 var observable2$ = Rx.Observable.interval(400);  
 this.subscription1$ = observable.subscribe(x => console.log("From interval 1000" x));  
 this.subscription2$ = observable.subscribe(x => console.log("From interval 400" x));  
 } ngOnDestroy() {  
 this.subscription1$.unsubscribe()  
 this.subscription2$.unsubscribe()  
 }  
}

**There are lot of subscriptions then how to destroy in component:**

We can gather them subscriptions in an array and unsubscribe from them in the **ngOnDestroy**:

@Component({...})  
export class AppComponent implements OnInit, OnDestroy {  
 subscription1$: Subscription   
 subscription2$: Subscription   
 subscriptions: Subscription[] = [] ngOnInit () {  
 var observable1$ = Rx.Observable.interval(1000);  
 var observable2$ = Rx.Observable.interval(400);  
 this.subscription1$ = observable.subscribe(x => console.log("From interval 1000" x));  
 this.subscription2$ = observable.subscribe(x => console.log("From interval 400" x));  
 this.subscriptions.push(this.subscription1$)  
 this.subscriptions.push(this.subscription2$)  
 } ngOnDestroy() {  
 this.subscriptions.forEach((subscription) => subscription.unsubscribe())  
 }  
}