# Rxjs operators

Starting rxjs 5.5 all the operators are provided as pure functions. So you need to put the new pure functions inside the pipe operator of Observable.

**Debounce**

In rxjs 5 onwards debounce is migrated to debounceTime. Debounce expects a function producing an observable. The sampling will be done at output of this observable.

debounceTime only expects a number at which the sampling is done. Same with throttle and throttleTime

How does debounceTime(1000) work?

Suppose source observable is emitting a value every 100 seconds or say at changing interval such as what happens during typing. The debounceTime will take up all values till 1000ms. Then it will output the last value within the time limit of 1000 ms. So if user has typed “hello” in 1000 ms then “h”, “he”, “hel”, “hell”, “hello” are received from source but only “hello” is outputted by debounceTime(1000). Note that if user has finished typing “hello” in say 500ms even then the value “hello” will be outputted after 1000ms. DebounceTime introduces a delay time interval and it outputs the last value received within that interval after that interval.

**Throttle**

Prior to rxjs 5.5 if you would like to use throttle operator you would do as:

import 'rxjs/add/operator/throttle';

import { interval } from 'rxjs/observable/interval';

const obs: **Observable**<number> = **interval**(100).**throttle**(() => **interval**(1000));

const obs1 = obs.**subscribe**(d => {

**console**.**log**(d);

(d == 100) && obs1.**unsubscribe**();

});

In above the import statement imports only the throttle operator. This operator can work on an instance of observable. There are many other operators like map, filter etc. They needed to be imported separately. interval(100) creates an observable like1,2,3,4… Each digit is generated at 100ms. Throttle above works on the observable created by interval(100). Throttle intercepts at every 1000 ms. So the above code generates 10, 20, 30… 100.

Throttle function intercepts an observable every 1000 ms and in between values are lost.

In new rxjs 5.5 pipe operator is available. Above code can be rewritten as below:

import { throttle } from 'rxjs/operators';

import { interval } from 'rxjs/observable/interval';

const obs: **Observable**<number> = **interval**(100).**pipe**(**throttle**(()=>**interval**(1000)));

const obs1 = obs.**subscribe**(d => {

**console**.**log**(d);

(d == 100) && obs1.**unsubscribe**();

});

See the pipe operator on the observable created by interval(100). Inside pipe operator the throttle operator is used as pure function. Pure function means it has no dependency on anything outside the function code. Previously the throttle operator would work on an observable but now it is not, it is used now as independent pure function.

Debounce also works in similar fashion but there is a difference. Debounce also works on two observables. Say first observable is continuously creating a series of values. Second observable creates a sampling rate observable. Now if debounce is used then when a sampling rate data is available from second observable then last value from first observable is sent to output observable.

import { throttle, debounce } from 'rxjs/operators';

const obs: **Observable**<number> = **interval**(120).**pipe**(**throttle**(() => **interval**(1000)));

const obs1 = obs.**subscribe**(d => {

**console**.**log**(d);

(d == 100) && obs1.**unsubscribe**();

});

# Reactive forms nth level control nesting

There are situations when deep nesting of angular components is required in forms. This is specially required if you have created Angular components and want to use those components in several places. This becomes cumbersome if Angular components are big and you do not want to put entire long code in the parent TS file. In such situations it is always better to break the stuff in several Angular components and nest those components. I was not successful in using recursive use of angular component where the same component is used in that component.

Keep in mind three things:

1. Html layout
2. TS controls
3. Binding between the two

Whatever you write in html file will be displayed in the browser. Actual controls are to be created in TS file and there need to be one-to-one mapping between html layout and TS controls. The attributes **formControlName, formGroupName, formArrayName and formGroup** are used for binding between TS file controls and HTML layout. These attributes are responsible for two-way data binding between html layout and TS controls. There may be following situations:

1. Html layout present but no TS control and no binding

The control will appear in the browser, but it will not be part of the form. At the time of submit the corresponding data will not be included in form JSON.

1. TS control created but no html layout present and no binding

You can create a control by using FormBuilder and attach that control to the form in TS code. If there is no corresponding html layout and therefore no binding defined, then the TS control’s value will be part of form JSON at time of rendering but there will be no display of control in the browser. There will be no error thrown either.

1. Html layout present and TS control created with incorrect binding

You have provided html layout and have also created TS controls and have attached those controls with form in TS file. But your binding is incorrect. Incorrect binding will throw browser error like control not found showing the path of control in browser.

There are three artifacts which are used in TS file to create form controls. They are FormControl, FormGroup and FormArray. Angular top-level form is actually FormGroup. The top-level form is binded with formGroup attribute. A simple implementation is as below:

## Html layout and binding

<form [*formGroup*]="myForm" *novalidate*>

<input *type*="text" [*formControlName*]="'child0'">

<button *type*="submit" *class*="btn btn-primary" *click*)="submit()">Submit</button>

</form>

Above is a simple form with single input text control. It expects a control with the name ‘child0’ in the TS file. The ‘child0’ control must be duly attached to the form myForm. See the [formGroup] attribute. It expects a form with the name ‘myForm’ to be created in TS file. The html form and TS form controls are bound together with the [formGroup] attribute.

## TS controls

**ngOnInit**() {

let formControls = {};

formControls["child0"] = ["", [validators], [asyncValidators]];

this.myForm = this.fb.**group**(formControls);

}

formControls object is created. Many controls can be attached to the formControls object. Finally, the formControls object is attached to the myForm. Hence myForm is actually a group containing many form controls. See the hierarchy of **child0** both in html layout and TS file. In html file child0 is placed directly in **myForm** and in TS file also the **child0** is attached to **myForm** using fb.group statement.

Treatment of formArray artifact is somewhat special. **formArray** is used as array of **groups.** Having said that in TS file you need to:

* At first create a group of controls in TS file by using fb.group().
* Create an array by using fb.array([]) and attach the group to the array
* Finally attach the array to the form

In html layout you need to create a container for array and associate it through [formArrayName] to the array created in TS file. Provide a placeholder for formGroupName as the index of array. This formGroupName as index is automatically created by Angular in the TS file. Then finally create the individual layout for controls of group in the array in html.

<form [*formGroup*]="myForm" *novalidate*>

<ng-container [*formArrayName*] = "'myArray'">

<ng-container \**ngFor* = "let group of myForm.get('myArray').controls; let j= index">

<ng-container [*formGroupName*] = "j">

<input *type*="text" [*formControlName*] = "'text1'">

<input *type*="text" [*formControlName*] = "'text2'">

</ng-container>

</ng-container>

</ng-container>

<button *type*="submit" *class*="btn btn-primary" (*click*)="submit()">Submit</button>

</form>

**ngOnInit**() {

let group1 = this.fb.**group**({

text1: [""],

text2: [""]

});

this.myForm = this.fb.**group**({});

this.myForm.**setControl**("myArray", this.fb.**array**([group1]));

}

In TS file see how the group1 is created consisting of text1 and text2 controls. Then the form myForm is instantiated and array of group is attached with the myForm.

## Nesting of form controls

Situation is main form consists of a text box and a component child1. The child1 has a text box and a component child2. The child2 has a textbox and a component child3. The child3 has a textbox and a component child4. The child4 has an array consisting of two textboxes.

Initially I tried it and failed. I was creating all the TS controls in the main form. I was able to nest till one level but thereafter it failed. Finally, I realized that each parent component should pass reference of self to the child as say parentControl. The child will draw itself in the parent control, both in html and at TS file level. If there are further children same concept is repeated.

Main form

<form [*formGroup*]="myForm" *novalidate*>

<input *type*="text" *placeholder*="child 0" [*formControlName*]="'child0'">

<child1-nest [*myForm*]="myForm" ></child1-nest>

<button *type*="submit" *class*="btn btn-primary" (*click*)="submit()">Submit</button>

</form>

**ngOnInit**() {

let formControls = {};

formControls["child0"] = [""];

this.myForm = this.fb.**group**(formControls);

}

Child1

<ng-container [*formGroup*]="myForm">

<ng-container [*formGroupName*]="'group1'">

<input *type*="text" [*formControlName*]="'child1'">

</ng-container>

<child2-nest [*parentControl*] = "myForm.get('group1')"></child2-nest>

</ng-container>

**ngOnInit**() {

let group1 = this.fb.**group**(

{

child1: ["12345"]

});

this.myForm.**setControl**("group1", group1);

}

Child2

<ng-container [*formGroup*]="parentControl">

<ng-container [*formGroupName*]="'group2'">

<input *type*="text" [*formControlName*]="'child2'">

<child3-nest [*parentControl*] = "parentControl.get('group2')"></child3-nest>

</ng-container>

</ng-container>

**ngOnInit**() {

let group2 = this.fb.**group**(

{

child2: ["12345"]

});

this.parentControl.**setControl**("group2", group2);

}

Child3

<ng-container [*formGroup*]="parentControl">

<ng-container [*formGroupName*]="'group3'">

<input *type*="text" [*formControlName*]="'child3'">

<input *type* = "radio">

<child4-nest [*parentControl*] = "parentControl.get('group3')"></child4-nest>

</ng-container>

</ng-container>

**ngOnInit**() {

let group3 = this.fb.**group**(

{

child3: ["abcdef"]

});

this.parentControl.**setControl**("group3", group3);

}

Child4

<ng-container [*formGroup*]="parentControl">

<ng-container [*formArrayName*]="'myArray'">

<ng-container \**ngFor* = "let group of parentControl.get('myArray').controls; let j = index">

<div [*formGroupName*]="j">

*<!-- <ng-container \*ngFor = "let control of group.controls"> -->*

<input *type*="checkbox" [*formControlName*] ="'check1'">

<input *type*="checkbox" [*formControlName*] ="'check2'">

</div>

</ng-container>

</ng-container>

</ng-container>

**ngOnInit**() {

let group1 = this.fb.**group**({

check1: [false],

check2: [true]

});

this.parentControl.**setControl**("myArray", this.fb.**array**([group1]));

}

Above code works seamlessly.

# ngClass and ngStyle

## ngStyle

Purpose of using ngStyle is to provide highest priority. This is equivalent to inline style.

[ngStyle]="{

'color': 'red',

'font-weight': 'bold',

'borderBottom': borderStyle

}"

You can use variable as value for ngStyle which you can tackle in code.

## ngClass

It can be passed object, array and string value. The values are appended to existing classes.

<button [ngClass]="['btn', 'btn-primary']">Button</button> //array

<button [ngClass]="'btn btn-primary'" //string

<button [ngClass]="{ btn:true, 'btn-primary':true }"> //object

## ng:deep

Due to shadow DOM, a component’s class is not available to its children. By using ng-deep directive you can make that class available to all children

Example:

::ng-deep fieldset {

border: 2px groove threedface;

padding: 5px;

margin: 10px auto;

}

# Forms

ngModel is two way binding and you need to import it in formsmodule. Template driven forms are enabled by importing the formsmodule. There are two forms approaches. Template driven and model driven / reactive forms. In template driven approach we use directives. These directives in turn create model in background. In model driven approach we create model directly in code.

Template driven forms: Everything including data binding and validations is taken care of by directives.

## Template driven

Prime actor is ngModel two way binding. Validation and data binding are done at template level.

## Custom validator for reactive forms

<https://angular-2-training-book.rangle.io/handout/forms/reactive-forms/reactive-forms_custom_validation.html>

If you need to pass arguments to your custom validator there is a trick. You need to return a function f from custom validator. This f has a parameter of type formControl from which you can have the value of formcontrol. This is how I did:

**myValidate**(s) {

let **func** = (control: **FormControl**) => {

return (control.value.**indexOf**(s) >= 0 ? null : { myValidateError: 'true' });

};

return (func);

}

# Create NPM module using YO and library generator

I used <https://github.com/jvandemo/generator-angular2-library> and found it useful for generating npm modules from angular 2, 4, 5 and so on.

## Process

### Step 1: Install and create container to generate the NPM module

$ npm install -g yo

$ npm install -g generator-angular2-library

make a new directory and cd into it:

$ mkdir angular-library-name

$ cd angular-library-name

and generate your new library:

$ yo angular2-library

### Step 2: Create npm module

* Create an angular module in your angular app. Copy and paste your angular module in the src folder as created above.
* Rename the module.ts of the src folder to index.ts
* Remove all samples from src and let stay the package.json and other system files. Following command will generate the npm module in dist folder:

$ npm run build

* Now publish the NPM module by cd dist, npm login, npm publish

### important points

* You must provide exclusive export statements in the module.ts file which is copied into src folder after renaming it as index.ts. For every component and every service there should be independent export \* statement as follows. If you have a nested module in your existing module then there should be one export for the nested module also. In present case there is a nested module broker in the current module.

export \* from './my-graph/my-graph.component';

export \* from './my-graph/info/info.component';

export \* from './neuro.service';

export \* from './neuro1.service';

export \* from './broker/broker.module';

* In nested module’s module.ts file you must exclusively export its components and services. In present case broker.module is a nested module and it has a broker.service and broker.component. So in broker.module.ts following lines are required:

export \* from './broker/broker.component';

export \* from './broker.service';

* If you have a service in parent module then the module.ts or index.ts must treat the service as follows. Parent module name is NeuroGraph1Module and it has got two services NeuroService and Neuro1Service. Conventionally there is one static method created as forRoot() which returns the services as providers. These service need not exist in any other providers[] statement.

export class NeuroGraph1Module {

static forRoot(): ModuleWithProviders {

return {

ngModule: NeuroGraph1Module,

providers: [NeuroService, Neuro1Service]

};

}

}

In app.module.ts file you must add the forRoot() when using the NPM module as follows:

NeuroGraph1Module.forRoot()

* If you have services in a nested module then these have to be treated accordingly. In module.ts file of the nested module provide the forRoot() statement as follows. The nested broker module has one broker service in present case.

export class BrokerModule {

static forRoot(): ModuleWithProviders {

return {

ngModule: BrokerModule,

providers: [BrokerService]

};

}

}

* Since the parent module NeuroGraph1 module has the nested Broker Module so in NeuroGraph1 module consume the Broker module with forRoo()

imports: [

CommonModule

, BrokerModule.forRoot()

],

Sample tested code is provided in present folder

### Point to note

I used npm script “npm run libgen” in the neuro-app’s package.json file to generate the NPM module. I noticed that **npm install npmModuleName** or **npm install npmModuleName@latest** or **npm install npmModuleName@latest –force** do not install the latest versions. Please use **npm update npmModuleName**. This installs the latest version of npm module.



# Lazy Loading

Lazy loading is tricky in Angular 4. Let us consider the app.module as master and lazy.module as lazy module. Lazy module is loaded when corresponding link for lazy module is clicked. Lazy loading only happens when you click a button or link which is part of routing mechanism. In Lazy loading you have to provide routes in app.module as well as in lazy.module. You can at first make a normal route working then a) create a lazy module, b) create routes file for lazy module and then do necessary changes in them

Following are distinct steps:

1. **app.module**
   1. create app.routes.ts and export routes:Routes. Use l**oadChildren** in place of component. There is specific string format for loadChildren value
   2. import routes in app.module.ts with forRoot(routes)
2. lazy.module
   1. create lazy.routes.ts and export routes:Routes. There should be single path with blank ‘’ and component as MyLazyComponent.

**app.routes.ts**

import {RouterModule, Routes} from '@angular/router';

export const routes : Routes = [

{

path: '',

redirectTo: 'welcome',

pathMatch: 'full'

}, {

path: 'lazy',

loadChildren: './lazy/lazy.module#LazyModule'

}

];

**app.module.ts**

import { BrowserModule } from '@angular/platform-browser';

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

import {RouterModule} from '@angular/router';

import { AppComponent } from './app.component';

import {routes} from './app.routes';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

RouterModule.forRoot(routes)

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

**lazy.routes.ts**

import {RouterModule, Routes} from '@angular/router';

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

import { MyLazyComponent } from '../lazy/my-lazy/my-lazy.component';

export const routes : Routes = [

{

path: '',

component: MyLazyComponent

}

];

**lazy.module.ts**

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

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

import { MyLazyComponent } from './my-lazy/my-lazy.component';

import {RouterModule} from '@angular/router';

import { routes } from './lazy.routes';

@NgModule({

imports: [

CommonModule,

RouterModule.forChild(routes)

],

declarations: [MyLazyComponent],

exports:[MyLazyComponent]

})

export class LazyModule { }

Old learning from Angular 2

## Routing

Angular2 routes the links to components and not controllers.

**Step 1: Prepare the main index.html file**

Provide <base href="/"> in index.html file

**Step 2: app.component.ts**

In template provide following html. Herein see the usage of <router-outlet>. This is where the routing stuff appears.

<div>

<nav>

<a href="#">Navigation:</a>

<ul>

<li><a routerLink='/home' routerLinkActive="active">Home</a></li>

<li><a routerLink='/route1' routerLinkActive="active">Route1</a></li>

<li><a routerLink='/route2' routerLinkActive="active">Route2</a></li>

</ul>

</nav>

**<router-outlet></router-outlet>**

</div>

**Step 3: Create router paths and provide that to rest of application**

Create app.routes.ts file.

import {Routes, RouterModule} from '@angular/router';

import {Route1, Route2, Home} from './app.routes.components';

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

const routes: Routes = [

{

path:'home',

component:Home

},

{

path:'route1',

component:Route1

},

{

path:'route2',

component:Route2

}

,{

path:'',

component:Home

}

];

export const Routing:ModuleWithProviders = RouterModule.forRoot(routes);

**Step 4: Create different components which are to be shown through router**

@Component({

template: '<h1>Route1</h1>'

})

export class Route1 extends MyComponent {

constructor() {

super('Test Data');

super.custom1();

//super.ngOnDestroy();

}

ngOnDestroy() {

console.log('ngOnDestroy called from Route1');

super.ngOnDestroy();

}

};

@Component({

template: '<h1>Route2</h1>'

})

export class Route2 {

constructor(private appService: AppService) { }

subscription: Rx.Subscription;

ngOnInit() {

this.subscription = this.appService.getData('data2').subscribe(

d => {

console.log(d);

});

};

ngOnDestroy() {

console.log('Destroy Route 2');

this.subscription.unsubscribe();

}

}

**Step 5: Prepare the ngModules in app.modules.ts file**

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

import { BrowserModule } from '@angular/platform-browser';

import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';

import {ChildComponent} from './childComponent';

import {AppService} from './app.service';

import {Routing} from './app.routes';

import {Route1, Route2, Home} from './app.routes.components';

@NgModule({

imports: [BrowserModule, HttpModule, Routing],

declarations: [AppComponent, ChildComponent, Route1, Route2, Home],

providers: [AppService],

bootstrap: [AppComponent]

})

export class AppModule { }

# Observables

Observable is immutable, means it cannot be altered but can only be created once. The instance methods on observable are actually operators for the observable. Operators are like map, filter, concat, distinct, do. Operators always output a newly created observable, they do not change the original observable.

**My understanding**

Observables are intelligent objects. Their behavior is very much dependent on subscription. They do not retain data for ever. They maintain copy for each subscription and deliver data to each subscriber. If there is no subscriber then observable does not maintain any data.

Subjects are both observable and observer. Data can be pushed to Subject by next() operation. If somebody X has subscribed to a subject S and then anybody else Y pushes data to S by next() operation then that data is available to X. But if subscription is not done already and data is pushed to subject then that data is lost. The subscriber only receives most current data.

In case of BehaviorSubject even if subscription is done later after somebody pushed data to subject, the subscriber gets the last data. For example the behaviorSubject B is created, then Y pushes many data to B, then subscriber X subscribes to B, now subscriber X gets the last data pushed by Y.

If a component is removed because of routing or any other means that component does not automatically unsubscribes the observable / subject. So the observable still retains a copy of data for that component. Later on when the component comes to life, even the earlier copies of data kept in observable / subject is passed to the component. So it is always recommended to unsubscribe a component in the **ngOnDestroy** lifecycle method in the component class. If you unsubscribe a component at proper lifecycle place then that component will properly get the last data as expected when it dies out or comes to life again.

I successfully implemented backbone, lodash, jquery and backbone.radio with Angular 2

# Creating a data service using Subject

## Create a subject

Generally this is done in service at client side or handler.js file at server side.

Subject subject = new rx.Subject

## Subscribe to filtered output of subject

Several sources push data into subject. Data is bagged with an id. Each bag of data is destined for different targets. Filtration on id is done so that each target can receive data for him. The target calls the ajax or async process with an id and expects to receive data with an id.

1. // filter the output of subject

function on(id) {

return (subject.filter(d => d.id === id));

}

1. //subscribe

on('authenticate').subscribe(d => { //do something with d})

## Push output of ajax call / asynchronous process into subject

## subject.next({ id: id, data: some data });

## Initiate ajax / asynchronous call

handler.edgePush('authenticate',data);

## Extra understanding behaviorSubject and ReplaySubject

Please note that for a subject to be used in above manner as data service the subscription comes before than creation / population of subject. If subscription is done after data pushing to a subject then that data is lost forever. Suppose there is a situation that subscription is possible only after data is pushed into a subject and then the last / latest data is required. BehaviorSubject comes to rescue. You can create a BehaviorSubject with a single data into it. Then whenever in future that data is required, just subscribe it and then latest data is available. If you push another data in BehaviorSubject then the older data is overwritten. That means it caches the last single data. Mind it that it stores the last / latest / one data. If you want to use last n number of data then you can use ReplaySubject. With ReplaySubject you can specify the number of last data (n) to retain. Also I implemented multiple BehaviorSubject as properties of an object in Kistler project for similar purpose.

## Async validation

This is third argument. See the testAsync which is asynchronous validation. It expects a promise.

Signature of a promise is:

Promise pr = new Promise((resolve,reject){

//Some code

Resolve({});

});

This code works. See the bind(this). It is required to provide the this to async validator.

ngOnInit() {

this.changePwdForm = this.fb.group({

oldPassword: ['', Validators.required]

, newPassword1: ['', Validators.required]

, newPassword2: ['', Validators.required, this.testAsync.bind(this)]

}, { validator: this.checkFormGroup });

};

testAsync(control) {

let pr = new Promise((resolve, reject) => {

this.appService.filterOn('get:default:credit:card').subscribe(d => {

if (d.data.error) {

console.log('Error in default credit card');

} else {

resolve({ testError: true });

}

});

this.appService.httpGet('get:default:credit:card');

});

return (pr);

}

checkFormGroup(group) {

let ret = null;

if (group.dirty) {

if (group.value.oldPassword == group.value.newPassword1) {

ret = { 'oldAndNewPasswordsSame': true }

} else if (group.value.newPassword1 != group.value.newPassword2) {

ret = { 'confirmPasswordMismatch': true };

}

}

return (ret);

}

Async validation is also possible at group level in following manner. Just add asyncValidator = …

this.userForm = formBuilder.group({

'email': ['', [Validators.required, appService.emailValidator]]

}, { validator: this.groupValidator, asyncValidator: this.testAsync.bind(this) });

testAsync(control) {

let pr = new Promise((resolve, reject) => {

this.appService.filterOn('smartyStreetAutoCom').subscribe(d => {

if (d.data.error) {

console.log('Error in default credit card');

} else {

resolve({ testError: true });

}

});

this.appService.httpGet('smartyStreetAutoCom');

});

return (pr);

}

## How I installed and used spin.js with angular 2

It was typical:

1. Npm install –save spin
2. Npm install –save @types/spin
3. Create new file app.spinner.ts
   1. import Spinner = require("spin.js");
4. Now make use of spin normal code using observables as in Kistler project. For every http call now spinner will be activated.
5. I also used Primeng Ui block to disable UI while spinner is active.