p

Not defining type of variable causes it to have type any

**let** a;

a = 1;

a = true;

a = 'a'

Define enum in typescript :

**enum** Color {

    Red,

    Green,

    Blue

}

**let** backColor = Color.Blue;

tsc main.ts | node main.js

Intellisense doesn’t work in this situation :

we should use type assertion to have intellisense :

(note the type is not changed after assertion in runtime and still it is any type)

**let** message;

message = 'abc';

**let** endsWithC = (<string>message).endsWith('c')

**let** endsWithCAlt = (message as string).endsWith('c')

a way of defining function in typescript :

**let** drawpoint = (x,y) **=>** {

}

drawpoint2({

    x:'1', y:'2'

})

It’s how to define custom type in typrscript:

**interface** Point {

    x: number,

    y: number

}

**let** drawPoint3 = (point: Point) **=>** {

}

In interface we can have functions and because they can access the properties we should not specify the paramaters

**interface** Point2 {

    x: number,

    y: number,

    draw : () **=>** void

}

The cohesion concept in OOP tells us that we should have Point type and function in the same place call Class: (without this they are hanging in the air somehow)- Class groups properties and functions that are highly related

**class** Point {

    x: number;

    y: number;

    draw() {

*// ...*

    }

    getDistance(another: Point) {

*// ...*

    }

}

Object creation in typescript :

**let** pointObj: Point = new Point();

**let** pointObj2 = new Point();   *// looks better*

pointObj.x = 2;

pointObj.y = 3;

pointObj.draw();

in typescript we can not have multiple constructors so here is the solution for creating object without default constructor :

**class** Point {

    x: number;

    y: number;

**constructor**(x?: number, y?: number) {

        this.x = x || 0;

        this.y = y || 0;

    }

    draw() {

        console.log('X: ' + this.x + ', Y:' + this.y)

    }

    getDistance(another: Point) {

*// ...*

    }

}

In typescript we have private – public and protected access modifiers :

(by default properties are public)

**class** Point3 {

**private** x: number;

**private** y: number;

**constructor**(x?: number, y?: number) {

        this.x = x || 0;

        this.y = y || 0;

    }

    draw() {

        console.log('X: ' + this.x + ', Y:' + this.y)

    }

    getDistance(another: Point) {

*// ...*

    }

}

*// in js*

**let** log = **function**(message) {

    console.log(message)

}

*// in ts*

**let** doLog = (message) **=>** {

    console.log(message)

}

**let** doLog2 = (message) **=>** console.log(message)

**let** doLog3 = () **=>** console.log()

**class** Point4 {

**constructor**(**private** x: number, **private** y: number) {

    }

    draw() {

        console.log('X: ' + this.x + ', Y:' + this.y)

    }

    getDistance(another: Point4) {

*// ...*

    }

}

Properties in typeScript :

**class** Point4 {

**constructor**(**private** \_x: number, **private** \_y: number) {

    }

    draw() {

        console.log('X: ' + this.x + ', Y:' + this.\_y)

    }

    getDistance(another: Point4) {

*// ...*

    }

**get** x() {

        return this.\_x;

    }

**set** x(value) {

        if (value < 0) {

            throw new Error('value cannot be less than 0.')

        }

        this.\_x = value;

    }

}

new Point4(2,4).x = 4;

when we use export in class definition we can have a module :

export **class** Point5 {

**constructor**(**private** \_x: number, **private** \_y: number) {

    }

    draw() {

        console.log('X: ' + this.x + ', Y:' + this.\_y)

    }

    getDistance(another: Point4) {

*// ...*

    }

**get** x() {

        return this.\_x;

    }

**set** x(value) {

        if (value < 0) {

            throw new Error('value cannot be less than 0.')

        }

        this.\_x = value;

    }

}

import {Point5} from './point'

export **class** LikeComponent {

**constructor**(**private** \_likesCount: number, **private** \_isSelected: boolean) {

   }

   onClick() {

    this.\_likesCount += (this.\_isSelected) ? -1 : 1;

    this.\_isSelected = !this.\_isSelected

   }

**get** likesCount() {

      return this.\_likesCount;

   }

**get** isSelected(){

      return this.\_isSelected;

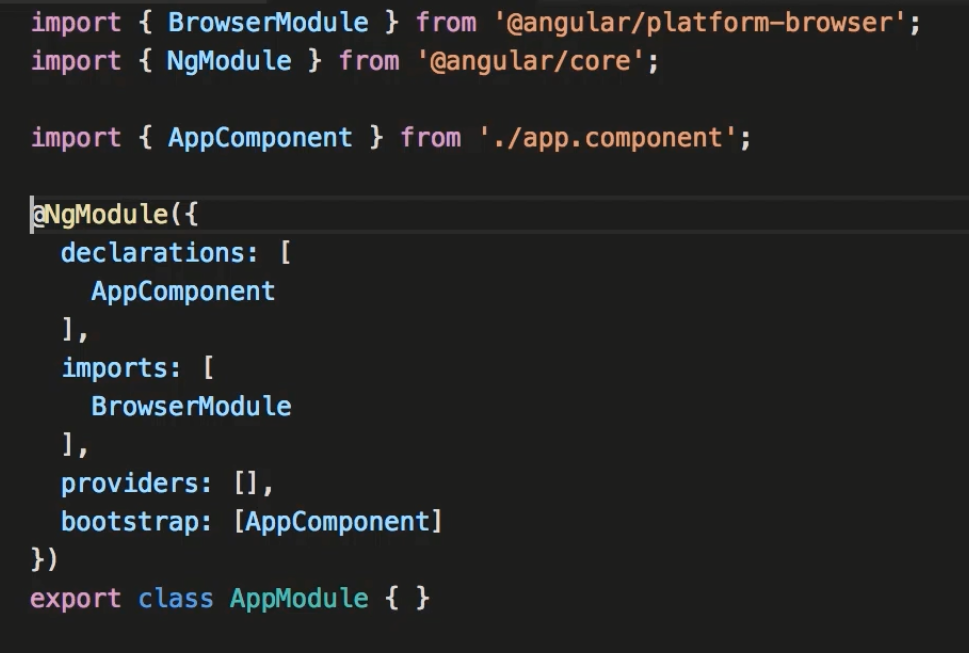
   }

}

We must register a component in a module to use it

Components are decorated with decorator functions @Component({}) and the templates and selector are assigned. In this way we can convert a plain typescript class to component

With decorator function @NgModule({}) we convert a plain typescript class to a angular module. In declaration property w specify components that are part of this module



Install auto import (steoates)

By using angular cli a component is created and added to module

ng g c course

this expression will be evaluated at runtime and the value of title field wii be inserted in the DOM

if the value of title field changes in the future Angular will automatically update the DOM

bind the view to a field in a component – this is called **string Interpolation**

<h2>{{ title }}</h2>

<h2>{{ "Title : " + title }}</h2> // simple javascript expression

<h2>{{ "Title : " + getTitle() }}</h2>

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

@Component({

  selector: 'app-course',

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

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

})

export **class** CourseComponent {

    title = 'List of courses'

    getTitle() {

      return this.title;

    }

  }

We use directives to manipulate a DOM element like removing or inserting or change a class of DOM element or it’s style (if we use directive that modifies DOM we should prefix it with asterisk)

<ul>

    <li \*ngFor="" ></li>

</ul>

<ul>

    <li \*ngFor="**let** course of courses" >

        {{course}}

    </li>

</ul>

We use a specific class called Service for implementing services in angular

In app module we should register any dependency that components in module are dependent upon

Angular will create a single object for entire module and will be injected (Singleton pattern)

ng g s courses

@NgModule({

  declarations: [

    AppComponent,

    CourseComponent

  ],

  imports: [

    BrowserModule

  ],

  providers: [

    CoursesService

  ],

  bootstrap: [AppComponent]

})

export **class** AppModule { }

here is how to consume the service via injection :

@Component({

  selector: 'app-course',

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

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

})

export **class** CourseComponent {

    title = 'List of courses'

    courses;

**constructor**(**private** service: CoursesService) {

      this.courses = service.getCourses();

    }

    getTitle() {

      return this.title;

    }

  }

This class in decorated with @Injectable function when it has dependency inside it’s constructors

@Injectable({

  providedIn: 'root'

})

export **class** MailService {

**constructor**(**private** service: LogService) {

    }

In case of component class it has injection inside component decorator function

ul>li will do the job

with Property binding we bind a property or attribute with a property or field in out component

 <img [src]="imageUrl" >

export **class** AuthorsComponent {

  authors;

  imageUrl : string = '/assets/angular.png'

**constructor**(**private** service: AuthorsService) {

    this.authors = service.getAuthors();

  }

**Property binding** is on way binding meaning if user changes DOM the underline fields or properties inside ts file wont get updated. But string interpolation is two way binding

<img  src= {{imageUrl}}  />

**Attribute binding**

There are some properties that html elements have but they don’t exist in DOM element inside memory and vice versa for example colSpan of td element is not in the DOM but in the html . Property binding in angular works for the properties that are in the DOM. We should prefix the propert with attr. In order it to work and not get error. This is called attribute binding

<table>

    <tr>

        <td colspan="attr.colSpan"></td>

    </tr>

</table>

export **class** CourseComponent {

    title = 'List of courses'

    courses;

    colSpan = 2;

in order to install bootstrap we use

npm install bootstrap –save

to install it in node\_modules folder and save it as dependency in package.json file

by listing the dependency in package.json when someone check out the project the dependency can be installed

in styles.css we import the reference of bootstrap folder in node\_module

@import "~bootstrap/dist/css/bootstrap.css";

body {

  padding:20px;

}

<button class="btn btn-primary">Save</button>

**Class binding** in angular : if value is true the class will be added if not the class will be removed

For this purpose we use a variation of property binding

<button class="btn btn-primary" [class.active]="isActive">Save</button>

export **class** CourseComponent {

    title = 'List of courses'

    courses;

    colSpan = 2;

    isActive = true;

**Style binding**

<button [style.backgroundColor]="isActive ? 'blue' : 'white'" >Save</button>

List of style binding can be found on w3school

Variation of property binding : class , attribute, style binding

**Event binding**

For handling the events raised by the DOM and instead of brackets we use parathesis

<button (click)="onSave()">Save</button>

Sometimes we want to get access to the event object that was raised in the event handler for example in the event object of a mouse movement we can access the x and y coordinates

We do it by sending $event parameter

<button (click)="onSave($event)">Save</button>

export **class** CoursesComponent {

  onSave($event) {

    console.log("Button was clicked", $event)

  }

}

**Event bubbling**

If a DOM event is fired is will bubble up all the way up and passess all the parrent elements

In this case after clicking the button the vent of the parent div will b fired.

<div (click)="onDivClicked()">

    <button (click)="onSave($event)">Save</button>

</div>

export **class** CoursesComponent {

  onDivClicked() {

    console.log("div clicked");

  }

  onSave($event) {

    console.log("Button was clicked", $event)

  }

}

We can stop propogation and bubbling up :

onSave($event) {

    $event.stopPropogation();

    console.log("Button was clicked", $event)

  }

Event filter

<input (keyup)="onKeyUp($event)" />

 onKeyUp($event) {

    if($event.keyCode === 13) console.log("Enter was pressed")

  }

There is a better way for doing this : (by eventfilter) :

<input (keyup.enter)="onKeyUp($event)" />

 onKeyUp($event) {

     console.log("Enter was pressed")

  }

**Template variable**

We can refere to the input field by target property of $event

console.log($event.target.value)

declare a variable in the template that refer to the inputfield

<input #email (keyup.enter)="onKeyUpNew(email.value)" />

  onKeyUpNew(email) {

    console.log(email)

 }

In property binding the direction of binding is from component in the view hence it is one way binding and changing it from the view doesn’t change the value in the log

<input [value]="email" (keyup.enter)="onKeyVer2()" />

export **class** CoursesComponent {

  email = "me@email.com";

 onkeyUpVer2() {

    console.log(this.email)

  }

}

**Two way binding with banana in the box**

<input [value]="email" (keyup.enter)="email = $event.target.value; onkeyUpVer2()" />

**Instead of above method :**

We can use a built-in angular directive called ngModule inside [()] for implementing two way binding

We should add FormsModule :

@NgModule({

  declarations: [

    AppComponent,

    CourseComponent,

    AuthorsComponent,

    CoursesComponent

  ],

  imports: [

    BrowserModule,

    FormsModule

  ],

<input [(ngModel)]="email" (keyup.enter)="onkeyUpVer2()"  />

Actually we bind to ngModule

**Pipes**

We use pipes to format data

Angular has some built-in types like uppercase and number(decimal)

{{course.title | uppercase}}

{{course.students | number }} <br/>

Pipes can be chained

{{course.title | uppercase  | lowercase }}

Number pipe can have parameter for number of integer and min and max number of precision

{{course.rating | number: '1.2-2'}}

{{course.rating | number: '2.2-3'}}

{{course.price | currency : 'AUD'}} <br/>

{{course.price | currency }} <br/>

{{course.price | currency : 'AUD': true:'3.2-2'}}

{{course.releaseDate | date:'shortDate'}}

When we Import the browser module it also import commonmodule

**Custom pipes**

Just like components we should register pipes in module

{{ text | summary }}

import {Pipe, PipeTransform} from '@angular/core'

@Pipe({

    name: 'summary'

})

export **class** SummaryPipe **implements** PipeTransform {

    transform(value: string, args?: any) {

        if(!value) {

            return null;

        }

        return value.substring(0, 50) + '...';

    }

}

@NgModule({

  declarations: [

    AppComponent,

    CourseComponent,

    AuthorsComponent,

    CoursesComponent,

    SummaryPipe

  ],

We can supply parameter in custom pipe :

{{ text | summary : 10}}

In angular we can change the parameter in method signature of an implemented interface

    transform(value: string, args?: any) {

transform(value: string, limit?: number) {

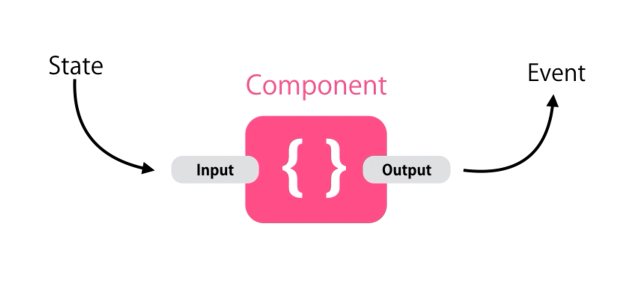
even we can have another parameters

transform(value: string, limit?: number, another?: number) {

Added to Github

In angular if you want to use a field as a property you should mark it as an input field

And if we want to have event binding we need to define speacial property named output property



The combination of input and output of a component is what we call public api of a component



We annotate the field with @Input() decorator now this field is exposed to outside and we can bind it to a property somewhere else

 @Input() isFavorite: boolean;

<favorite  [isFavorite]="post.isFavorite" ></favorite>

With alias we can set another name for input properties

  @Input('is-favorite') isFavorite: boolean;

<favorite  [is-favorite]="post.isFavorite" ></favorite>

It’s good because if we change the name of variable all the places have to be renamed

Output properties

 @Output() change = new EventEmitter();

<favorite  [isFavorite]="post.isFavorite" (change)="onFavoriteChanged()"></favorite>

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

@Component({

  selector: 'favorite',

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

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

})

export **class** FavoriteComponent **implements** OnInit {

  @Input('isFavorite') isFavorite: boolean;

  @Output() change = new EventEmitter();

**constructor**() {}

  ngOnInit(): void {}

  onClick() {

    this.isFavorite = !this.isFavorite;

    this.change.emit();

  }

}

export **class** AppComponent {

  title: string = 'Angular demo';

  post = {

    title: "Title",

    isFavorite: true

  }

  onFavoriteChanged() {

    console.log("Favorite changed")

  }

}

Passing event data :

App componenet here is the subscriber of change event :

<favorite  [isFavorite]="post.isFavorite" (change)="onFavoriteChanged()"></favorite>

  onClick() {

    this.isSelected = !this.isSelected;

    this.change.emit(this.isSelected);

  }

onFavoriteChanged(isFavorite) {

    console.log("Favorite changed", isFavorite)

  }

The value will be emitted to all subscribers of event (by rasing event we see when the state of component changes)

<favorite  [isFavorite]="post.isFavorite" (change)="onFavoriteChanged($event)"></favorite>

$event is a built-in object in angular previously $event when clicking buttons whas syandard dom event object but here in the custom component is anything that we pass when raising an event

In case we want to supply object in emit method

    this.change.emit({ newValue: this.isSelected});

now $event object represent actual javascript object that has property called newValue

<favorite  [isFavorite]="post.isFavorite" (change)="onFavoriteChanged($event)"></favorite>

  onFavoriteChanged(eventArgs) {

    console.log("Favorite changed", eventArgs)

  }

We can use interface for input parametr of method :

**interface** FavoriteChangedEventArgs {

  newValue: boolean

}

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export **class** AppComponent {

  title: string = 'Angular demo';

  post = {

    title: "Title",

    isFavorite: true

  }

  onFavoriteChanged(eventArgs: FavoriteChangedEventArgs) {

    console.log("Favorite changed", eventArgs.newValue)

  }

We can export this interface in Favorite component

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

 @Component({

  selector: 'favorite',

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

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

})

export **class** FavoriteComponent **implements** OnInit {

  @Input('isFavorite') isSelected: boolean;

  @Output() change = new EventEmitter();

**constructor**() {}

  ngOnInit(): void {}

  onClick() {

    this.isSelected = !this.isSelected;

    this.change.emit(this.isSelected);

    this.change.emit({ newValue: this.isSelected});

  }

}

export **interface** FavoriteChangedEventArgs {

  newValue: boolean

}

Output alias :

  @Output('change') click = new EventEmitter();

There are no separate request for server to download templates all of theme are bundled with javascript

Styles in angular : if we have small component we can declare the style in styles array

 @Component({

  selector: 'favorite',

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

  styleUrls: ['./favorite.component.css'],

  styles:[

    `

    `

  ]

})

export **class** FavoriteComponent {

  @Input('isFavorite') isSelected: boolean;

  @Output('change') click = new EventEmitter();

Also we can specify style in html file :

<style>

</style>

<app-course></app-course>

<app-authors></app-authors>

<app-courses></app-courses>

<favorite  [isFavorite]="post.isFavorite" (change)="onFavoriteChanged($event)"></favorite>

<title-case></title-case>

Based on the priority of definition in decorator function styles priority will be specified and previous stylee will be completely ignored

 @Component({

  selector: 'favorite',

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

  styleUrls: ['./favorite.component.css'],

  styles:[

    `

      .glyphicon {

    color: green;

    }

    `

  ]

})

Shadow DOM

Allows us to apply scoped styles to elements without bleeding out to the outer world

@Component({

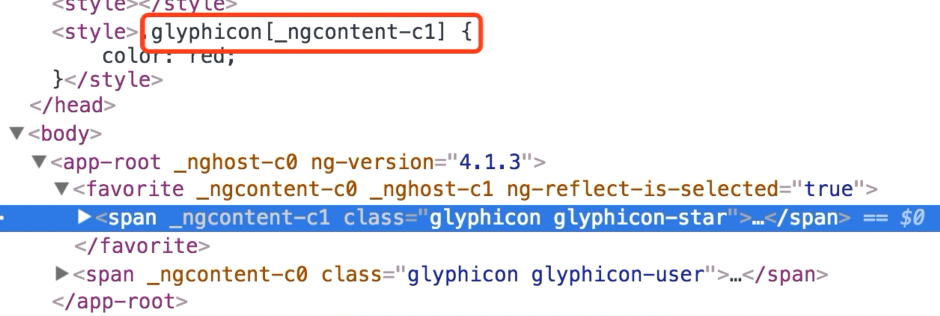
  selector: 'app-root',

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

  styleUrls: ['./app.component.css'],

  encapsulation: ViewEncapsulation.Emulated

})



In this way angular add atrribute to the class and use theme in styles to have unique styles for each component

encapsulation: ViewEncapsulation.ShadowDom

if we set to ShadowDOM it uses the default shadowdom in browsres thar most browsers don’t support

**ngContent**

<div class="panel panel-default">

    <div class="panel-heading">

        <ng-content select=".heading"></ng-content>

    </div>

    <div class="panel-body">

        <ng-content select=".body"></ng-content>

    </div>

</div>

<bootstrap-panel>

    <div class="heading">Heading</div>

    <div class="body">

        <h2>Body</h2>

        <p>Some content here...</p>

    </div>

</bootstrap-panel>

**We don’t need selector if we use just one ng-content**

**ng-container**

**this is an angular custom built-in element and at runtime angular is going to take the content of this ng container (not div or any element in the DOM but only the content)**

<bootstrap-panel>

    <ng-container class="heading">Heading</ng-container>

    <div class="body">

        <h2>Body</h2>

        <p>Some content here...</p>

    </div>

</bootstrap-panel>

<div class="panel panel-default">

    <div class="panel-heading">

        <ng-content select=".heading"></ng-content>

    </div>

    <div class="panel-body">

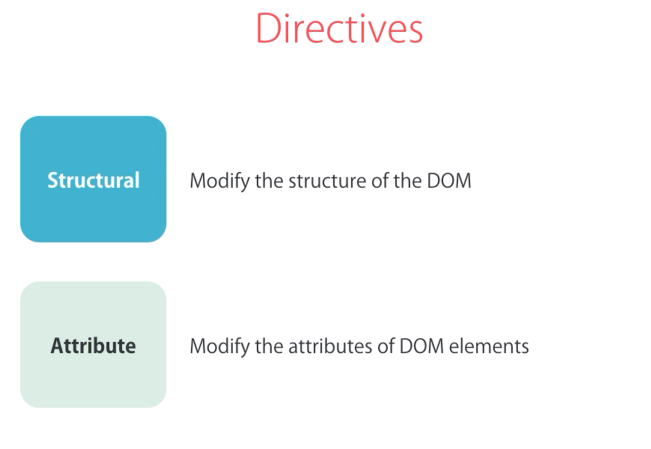
        <ng-content select=".body"></ng-content>

    </div>

</div>

**Directives**

**We use the directive to modify the DOM**

****

**Structural directives are prefix with asterisk:**

**ngIf :**

<div \*ngIf="courses.length > 0">

    List of courses

</div>

<div \*ngIf="courses.length == 0">

    No courses yet

</div>

**Other div will not be added in the DOM**

**We can implement it with template variable (use with hashtag) and ng-template :**

<div \*ngIf="courses.length > 0; else noCourses">

    List of courses

</div>

<ng-template #noCourses>

    No courses yet

</ng-template>

**Even we can implement in another way :**

<div \*ngIf="courses.length > 0; then coursesList else noCourse"></div>

<ng-template #coursesList>

    List of courses

</ng-template>

<ng-template #noCourse>

    List of courses

</ng-template>

**We can use hidden proeprty of html element**

<div hidden>

    List of courses

</div>

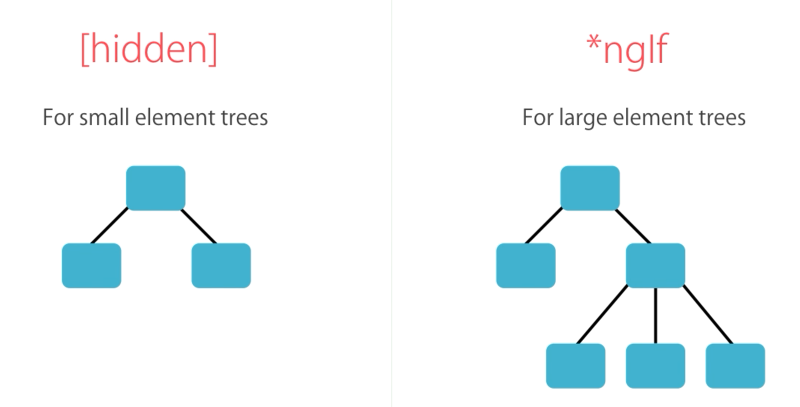
**And even we can bind it by property binding :**

<div [hidden]="courses.length == 0">

    List of courses

</div>

**Here elemenets are in the DOM but hidden not like ngIf**

****

**ngSwitchCase :**

<div [ngSwitch]="viewMode">

    <div \*ngSwitchCase="'map'">Map View Content </div>

    <div \*ngSwitchCase="'list'">List View Content </div>

    <div \*ngSwitchDefault>Otherwise</div>

</div>

  viewMode = 'map';

 changeViewMode(value: string) {

    this.viewMode = value;

  }

**ngFor :**

<ul>

    <li \*ngFor="let course of courses">{{course.name}}</li>

</ul>

  courses = [

    { id : 1, name:'course1' },

    { id : 2, name:'course2' },

    { id : 3, name:'course3' }

  ]

**Exported values in for :**

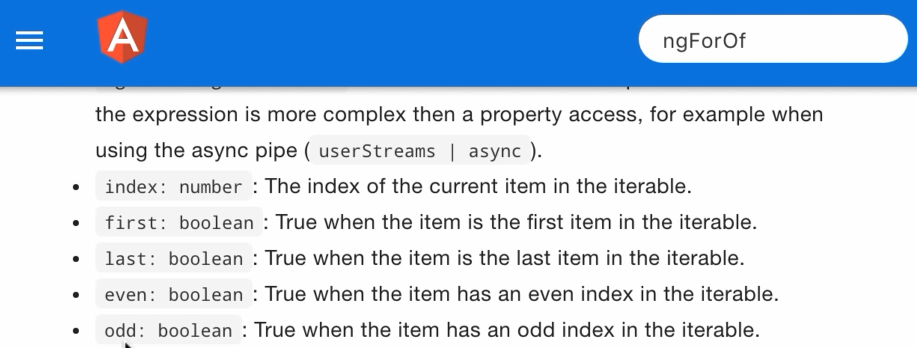
<ul>

    <li \*ngFor="let course of courses; index as i">

        {{ i }} - {{course.name}}

    </li>

</ul>

****

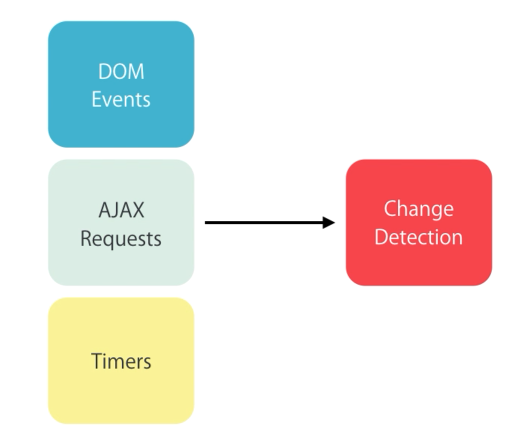
<ul>

    <li \*ngFor="let course of courses; even as isEven">

        {{course.name}} <span \*ngIf="isEven">EVEN</span>

    </li>

</ul>

****

<ul>

    <li \*ngFor="let course of courses; even as isEven">

        {{course.name}} <span \*ngIf="isEven">EVEN</span>

        <button (click)="onRemove(course)">remove</button>

    </li>

</ul>

  onRemove(course) {

**let** index = this.courses.indexOf(course);

    this.courses.splice(index, 1);

  }

**After removing the element angular will execute it’s change detection mechanism and realize one of the object in courses array is no longer there so it removes the corresponding ListItem in the DOM**

    <li \*ngFor="let course of courses; even as isEven">

        {{course.name}} <span \*ngIf="isEven">EVEN</span>

        <button (click)="onRemove(course)">remove</button>

        <button (click)="onChange(course)">Change</button>

    </li>

  onChange(course) {

    course.name = 'UPDATED';

  }

**After updating the value also angular will run it’s change detection mechanism**

At a high level, Angular walks your components from top to bottom, looking for changes. Angular runs its change detection mechanism periodically so that changes to the data model are reflected in an application’s view. Change detection can be triggered either manually or through an asynchronous event (for example, a user interaction or an XMLHttpRequest completion).

**F**

<button (click)="loadCourses()">Load Courses</button>

 loadCourses() {

    this.coursesForBtn = [

      { id : 1, name:'course1' },

      { id : 2, name:'course2' },

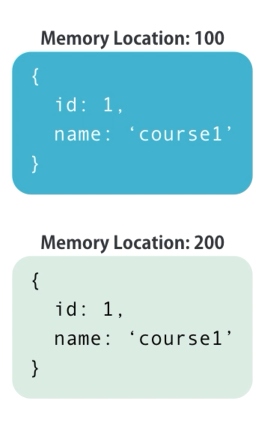
      { id : 3, name:'course3' }

    ]

  }

**When we click the load button several times tha angular will get the objects with same content from the server and populate the ul with same content but objects are different from the previous one in the memory so angular will reconstruct the DOM tree.**

**In ngFor we have this ability to specify how angular will track the objects by default it itracks them based on object identity (reference) in the memory**

****

**So we use different mechannism other than object identity in the memory and use their id’s for example :**

<ul>

    <li \*ngFor="let course of coursesForBtn; trackBy: trackCourse">

        {{course.name}}

    </li>

</ul>

  trackCourse(index, course) {

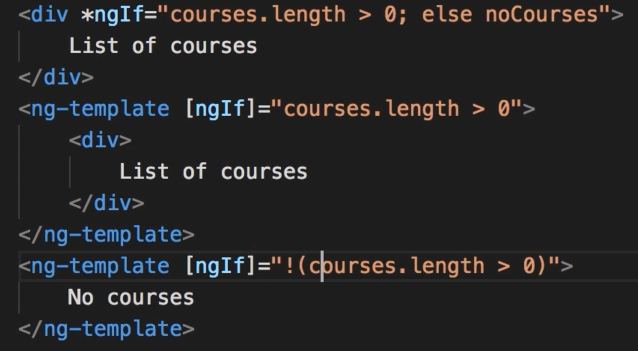
    return course ? course.id : undefined;

*// if we have a course object we want to return it's id otherwise we want to return undefined*

  }

**If we deal with large list using trackBy will have benefit but not with small list**

When we use asterisk angular will create ng-template and use property binding for showing html mark up just like this :

****

**ngClass**

instead of class binding :

 [class.glyphicon-star]="isSelected"

 [class.glyphicon-star-empty]="!isSelected"

We can use ngClass like this : (ngClass is an example of attribute directive we use it to modify attribute of existing DOM element)

<span class="glyphicon"

 [ngClass]="{

    'glyphicon-star': isSelected,

    'glyphicon-star-empty': !isSelected

 }"

 [style.fontSize]= "'30px'"

 (click)="onClick()"

 ></span>

**ngStyle**

when we use multiple style bindings it’s better to use ngStyle directive

so instead of this :

<button

    [style.backgroundColor]="canSave ? 'blue' : 'gray'"

    [style.color]="canSave ? 'white': 'black'"

    [style.fontWeight]="canSave ? 'bold': 'normal'"

>

    Save

</button>

We can have this :

<button

    [ngStyle]="{

        'backgroundColor': canSave? 'blue': 'gray',

        'color': canSave ? 'white': 'black',

        'fontWeight': canSave ? 'bold': 'normal'

    }"

>

    Save

</button>

**Safe traversal operator**

If an object is null and we access one of it’s property we get error consider this :

export class AppComponent {

  task={

    title:'Review applications',

    assignee: {

      name: 'John Smith'

    }

  }

<div>

    <span>{{task.assignee.name}}</span>

</div>

One way is to use \*ngIf (if the value is truthy it will be shown ) :

<span \*ngIf="task.assignee">{{task.assignee.name}}</span>

There is another way for doing that :

    <span>{{task.assignee?.name}}</span>

**Custom Directive**

We can use custom directive to have more control over behaviour of DOM elements

ng g d input-format

this command will create a directive and add it to the app.module

import { InputFormatDirective } from './input-format.directive';

@NgModule({

  declarations: [

    InputFormatDirective

  ],

The type script class is decorated with @Directive decorator function

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

@Directive({

  selector: '[appInputFormat]'

})

export class InputFormatDirective {

  constructor() { }

}

The selector has square brackets which means any element that has this attribute and angular is gonna apply this directive to the element. (prefix it with app is a good idea)

We use HostListener decorator and give the name of event as a parameter to bind the method with the event : (we use HostListener decorator to subscribe to the event raised from the host DOM object)

import { Directive, HostListener } from '@angular/core';

@Directive({

  selector: '[appInputFormat]'

})

export class InputFormatDirective {

  @HostListener('focus') onFocus() {

    console.log("on focus ");

  }

  @HostListener('blur') onBlur() {

    console.log("blur");

  }

  constructor() { }

}

<input type="text" appInputFormat />

By using ElementRef we can access the object that includes the event that has been raised.

For using it we must inject it in the constrcutor

import { Directive, ElementRef, HostListener } from '@angular/core';

@Directive({

  selector: '[appInputFormat]'

})

export class InputFormatDirective {

  constructor(private el: ElementRef) {

    //elementRef is a service inside angular for accessing DOM object

  }

  @HostListener('blur') onBlur() {

    let value: string = this.el.nativeElement.value;

// this give us access to actual DOM object

    this.el.nativeElement.value = value.toLowerCase();

  }

}

We can use property binding and add input to the directive :

<input type="text" appInputFormat [format] = "'format'" />

import { Directive, ElementRef, HostListener, Input } from '@angular/core';

@Directive({

  selector: '[appInputFormat]'

})

export class InputFormatDirective {

  @Input('format') format;

  constructor(private el: ElementRef) {

    //elementRef is a service inside angular for accessing DOM object

  }

  @HostListener('blur') onBlur() {

    let value: string = this.el.nativeElement.value;  // this give us access to actual DOM object

    if (this.format == 'lowercase')

      this.el.nativeElement.value = value.toLowerCase();

    else

      this.el.nativeElement.value = value.toUpperCase();

  }

}

If we have only one input we can use the name of directive as input like this :

<input type="text"  [appInputFormat] = "uppercase" />

 @import { Directive, ElementRef, HostListener, Input } from '@angular/core';

@Directive({

  selector: '[appInputFormat]'

})

export class InputFormatDirective {

  @Input('appInputFormat') format;

  constructor(private el: ElementRef) {

    //elementRef is a service inside angular for accessing DOM object

  }

  @HostListener('blur') onBlur() {

    let value: string = this.el.nativeElement.value;  // this give us access to actual DOM object

    if (this.format == 'lowercase')

      this.el.nativeElement.value = value.toLowerCase();

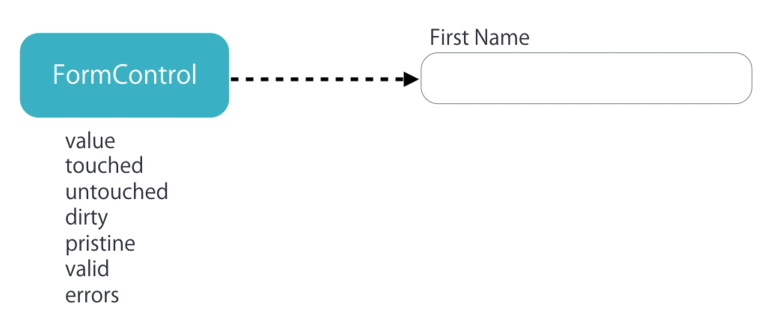
    else

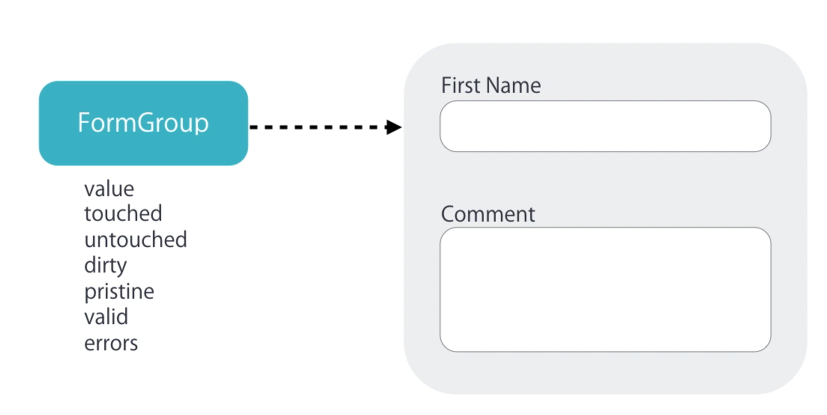
      this.el.nativeElement.value = value.toUpperCase();

  }

**Form-Validation**

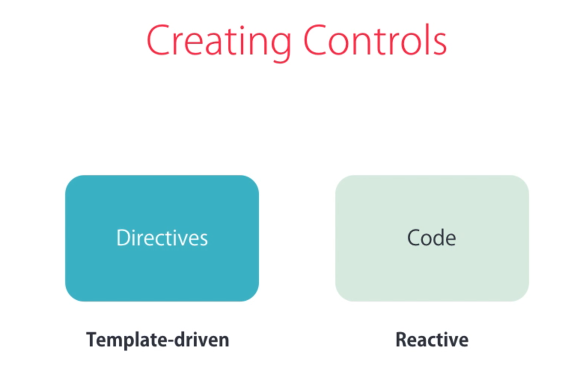
For each input of the form we need a control Object

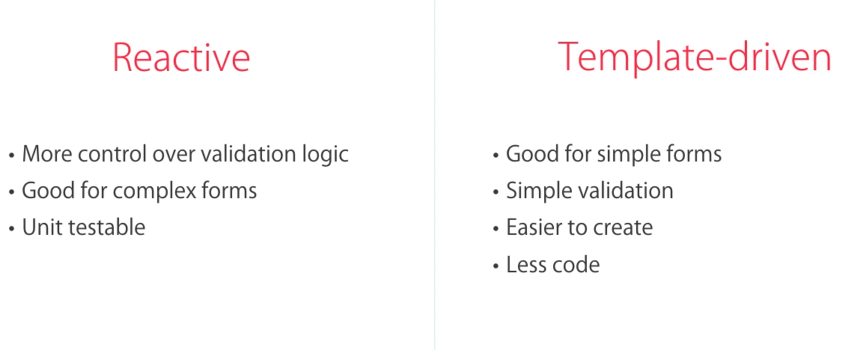




To add validation for the form we need to create formGroup object for the form and formControl object for each input field so that we can keep track of each control and also the entire group.

We have 2 ways to create this control objects :





By using ngModel as an attribute angular will create the control objects and associate it with the input field under the hood.

 <input ngModel name="firstName" id="firstName" type="text" class="form-control"

All we need to validate by directive is to use ngModel with name attribute

We can send the value of ngModel by defining templateVariable nad pass it to the eventHandler

   <input ngModel name="firstName" id="firstName" type="text" class="form-control" change="log(firstName)" #firstName="ngModel">

**Validation**

Adding required in html5 will validate the field

  <div class="form-group">

        <label for="firstName">FirstName</label>

        <input required ngModel name="firstName" id="firstName" type="text" class="form-control" (change)="log(firstName)" #firstName="ngModel">

        <div class="alert alert-danger" \*ngIf="!firstName.valid">First Name is required</div>

    </div>

It’s better to do this in this way :

 <div class="form-group">

        <label for="firstName">FirstName</label>

        <input required ngModel name="firstName" id="firstName" type="text" class="form-control" (change)="log(firstName)" #firstName="ngModel">

        <div class="alert alert-danger" \*ngIf="firstName.touched && !firstName.valid">First Name is required</div>

    </div>

**Other html5 built-in validation attribute**

<form>

    <div class="form-group">

        <label for="firstName">FirstName</label>

        <input required ngModel minlength="3" maxlength="10" pattern="banana" name="firstName" id="firstName" type="text" class="form-control" (change)="log(firstName)" #firstName="ngModel">

        <div class="alert alert-danger" \*ngIf="firstName.touched && !firstName.valid">

            <div \*ngIf="firstName.errors?.required">FirstName is required</div>

            <div \*ngIf="firstName.errors?.minlength">First name should be minimum 3 characters.</div>

            <div \*ngIf="firstName.errors?.maxlength">First name should be minimum 3 characters.</div>

            <div \*ngIf="firstName.errors?.pattern">First name doesn't match the pattern.</div>

        </div>

    </div>

    <div class="form-group">

        <label for="comment">Comment</label>

        <textarea ngModel id="comment"  class="form-control" cols="30" rows="10"></textarea>

    </div>

    <button class="btn btn-primary">Submit</button>

</form>

**Better implementation :**

<div \*ngIf="firstName.errors?.minlength">First name should be minimum

{{ firstName.errors?.minlength.requiredLength}} characters.</div>

**Use angular created class for bordering in styles.css :**

.form-control.ng-touched.ng-invalid {

  border: 2px solid red

}

**Angular will automatically add ngForm directive to form selector**

<form #f = "ngForm" (ngSubmit)="submit(f)">

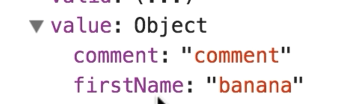
  submit(f) {

    console.log(f)

  }

**When ngForm directive is added to an element it will create formGroup object**

**In the value object of ngForm we can access the values of form by the name attribute as key in key/value pair (json represeatation of the form)**

****

 submit(f) {

    // console.log(f)

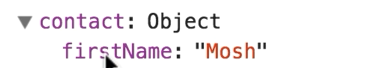
    console.log(f.value)

  }

**ngModelGroup**

we can use this directive to group form-control

    <div ngModelGroup="contract">



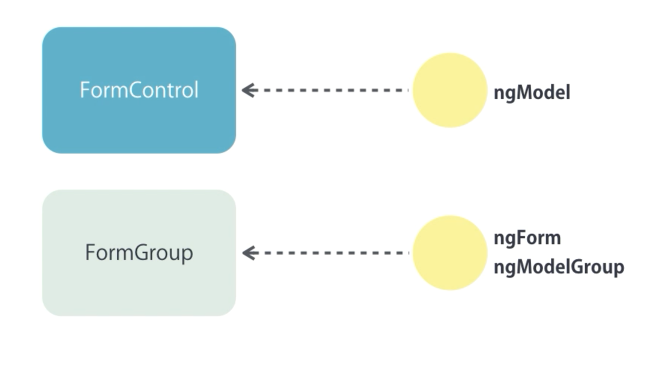
When we want to use validation for this group and show it on the top

    <div ngModelGroup="contact" #contact="ngModelGroup">

        <div \*ngIf="!contact.valid">

            ...

        </div>



Disabling sumit button

<form #f = "ngForm" (ngSubmit)="submit(f)">

    <button class="btn btn-primary" [disabled]="!f.valid" >Submit</button>

**CheckBox**

    <div class="checkbox">

        <label for="">

            <input ngModel name="isSubscribed" type="checkbox"> Subscribe to mailing list

        </label>

    </div>

    <p>

        {{ f.value | json }}

    </p>

**Dropdown list**

    <div class="form-group">

        <label for="contactMethod"></label>

  <select ngModel name="contactMethod" name="" id="contactMethod" class="form-control">

            <option value=""></option>

            <option \*ngFor="let method of contactMethods" [value]="method.id">{{ method.name }}</option>

        </select>

    </div>

**The value of select element has to be string but by using [ngValue] we can bind object to it.**

            <option \*ngFor="let method of contactMethods" [value]="method.id" [ngValue]="method">{{ method.name }}</option>

**Allowing user to support multiple selection :**

 <select multiple name="contactMethod" name="" id="contactMethod" class="form-control">

            <option value=""></option>

            <option \*ngFor="let method of contactMethods" [value]="method.id" >{{ method.name }}</option>

        </select>

 <div class="radio" \*ngFor="let method of contactMethods" >

        <label for="">

            <input ngModel type="radio" name="contactMethod" [value]="method.id">

            {{ method.name }}

        </label>

    </div>

**Reactive Form**

In app.module :

import { FormsModule, ReactiveFormsModule } from '@angular/forms';

 imports: [

    BrowserModule,

    FormsModule,

    ReactiveFormsModule

  ]

We should create form control object for each input like this :

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

import { FormGroup, FormControl } from '@angular/forms';

@Component({

  selector: 'signup-form',

  templateUrl: './signup-form.component.html',

  styleUrls: ['./signup-form.component.css']

})

export class SignupFormComponent {

  form = new FormGroup({

    'username': new FormControl(),

    'password': new FormControl()

  })

}

And use formFroup and formControlName directive like this :

<form [formGroup]="form">

    <div class="form-group">

        <label for="username">Username</label>

        <input

            formControlName="username"

            id="username"

            type="text"

            class="form-control">

    </div>

    <div class="form-group">

        <label for="password">Password</label>

        <input

            formControlName="password"

            id="password"

            type="text"

            class="form-control">

    </div>

    <button class="btn btn-primary" type="submit">Sign Up</button>

</form>

In FormControl constructors we send empty string as initial value and function reference from Validators class.

form = new FormGroup({

    'username': new FormControl('', Validators.required),

    'password': new FormControl('', Validators.required)

  });

Access form property and use get method to get formcontrol object

 <input

            formControlName="username"

            id="username"

            type="text"

            class="form-control">

        <div \*ngIf="form.get('username')?.touched && form.get('username')?.invalid"

            class="alert alert-danger">

            Username is required

        </div>

We can define a getter and not have that much noisy code :

  get username() {

    return this.form.get('username')

  }

 <div \*ngIf="username?.touched && username?.invalid"

            class="alert alert-danger">

            Username is required

        </div>

**Custom Validator function**

****

**We can use interface to define shape of functions too.**

**In another file we create like this :**

import { AbstractControl, ValidationErrors } from "@angular/forms";

export class UsernameValidators {

    static cannotContainSpace(control: AbstractControl) : ValidationErrors | null {

        if((control.value as string).indexOf('-') != -1)

            return { cannotContainSpace: true};

        return null;

    }

}

 form = new FormGroup({

    'username': new FormControl('', [

      Validators.required,

      Validators.minLength(3),

      UsernameValidators.cannotContainSpace

      ]),

    'password': new FormControl('', Validators.required)

  });

         <div  \*ngIf="username?.errors?.cannotContainSpace">username can not contain space</div>

**Async validation**

****

export class SignupFormComponent {

  form = new FormGroup({

    'username': new FormControl('', [

      Validators.required,

      Validators.minLength(3),

      UsernameValidators.cannotContainSpace],[

      UsernameValidators.shouldBeUnique

      ]),

    'password': new FormControl('', Validators.required)

  });

Bayad aval sync ha ro bezarim va badesh async haro

   static shouldBeUnique(control: AbstractControl) : Promise<ValidationErrors> | null {

        return new Promise((resolve, reject) => {

            setTimeout(() => {  // this is asynchronous operation and it does not block operation

                if (control.value === 'mosh')

                    resolve({ shouldBeUnique: true });

                else

                    return resolve(null);

            }, 2000);

        });

    }

**Loader Image**

We can use pending proeprty of FormControl object that atleast one async validator is in progress it returns true

<div \*ngIf="username.pending">Checking for uniqueness...</div>

**Validating when submit form**

****

****

**Nested FormGroup**

export class SignupFormComponent {

  form = new FormGroup({

    account: new FormGroup({

      'username': new FormControl('', [

        Validators.required,

        Validators.minLength(3),

        UsernameValidators.cannotContainSpace],[

        UsernameValidators.shouldBeUnique

        ])

    }),

    'password': new FormControl('', Validators.required)

  });

  get username() {

    return this.form.get('account.username')

  }

 <div formGroupName="account">

        <div class="form-group">

            <label for="username">Username</label>

            <input

                formControlName="username"

                id="username"

                type="text"

                class="form-control">

**use a parrent div and use formGroupName property**

**FormArray**

FormArray is useful when we need to work with array of objects in a form

export class NewCourseUdemyCourseComponent {

  form = new FormGroup({

    topics: new FormArray([])   //FormArray is useful when we need to work with array of objects in a form

  })

}

The first argument of FormArray is AbstractControlArray which we set as empty array : []

In the form we use key event and apply a filter (enter) like this :

(keyup.enter)="addTopic(topic)"

By using a template variable we can have a reference to this input field and send it as a parameter in addTopic method

 <input type="text" class="form-control" (keyup.enter)="addTopic(topic)" #topic>

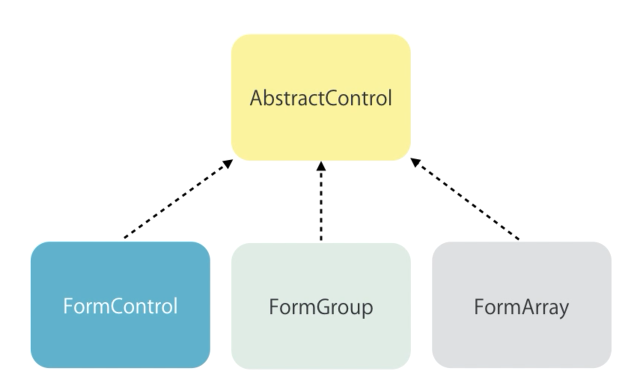
We use type annotation because the topic input can be confused with json input

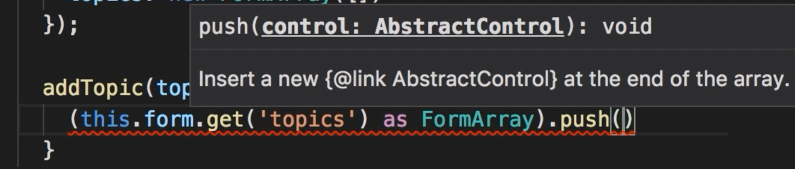
  addTopic(topic: HTMLInputElement) {

  }

We need to get a reference to the topic property of form field which is the FormArray :

By using the get method we do it but it returns an AbstractControl object which doesn’t have the push method so we explicitly cast it to FormArray





Push method gets a parameter of AbstractControl which can be any of child object in above image

 addTopic(topic: HTMLInputElement) {

    (this.form.get('topics') as FormArray).push(new FormControl(topic.value))

  }

By using controls property of FormArray object we can access all the FormControl objects inside this array

  get topics() {

    return (this.form.get('topics') as FormArray).controls;

  }

<form >

    <input type="text" class="form-control" (keyup.enter)="addTopic(topic)" #topic>

    <ul class="list-group">

        <li

            \*ngFor="let topic of topics"

            class="list-group-item">

            {{ topic.value }}

        </li>

    </ul>

</form>

We can clear the value of input like this :

  addTopic(topic: HTMLInputElement) {

    (this.form.get('topics') as FormArray).push(new FormControl(topic.value));

    topic.value = "";

  }

 removeTopic(topic: AbstractControl) {

    let index = this.topics.controls.indexOf(topic);

    this.topics.removeAt(index);

  }

**FormBuilder**

We can create FormCOntrol,FormGroup and FormArrays in the constructor via FormBuilder class

export class NewCourseUdemyCourseComponent {

  form;

  constructor(fb: FormBuilder) {

    this.form = fb.group({

      name: ['', Validators.required],

      contact: fb.group({

        email:[],

        phone:[]

      }),

      topicArray: fb.array([])

    });

  }

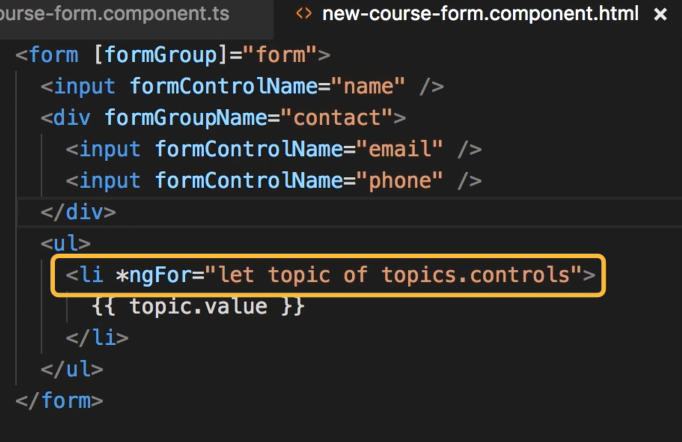
With reactive form we build out form explicitly in the code inside our component then we use directives to associate the controls we create in the component to the elements in the form. The directives we need to remember are :

[formGroup] = “form” which we apply use property binding to bind formElement with the form object in the component

formControlName : which we use like attribute and we use it to associate an input field with a control in our form

formGroupName : if we have subgroup in our form we use formGroupName

formArray : use to store controls in array when we want to render them in the form



**Consuming Http Service**

We need to import HttpModule In AppModule.ts

import { HttpClientModule } from '@angular/common/http';

 imports: [

    BrowserModule,

    FormsModule,

    ReactiveFormsModule ,

    HttpClientModule

  ]

Dependency injection in angular won’t work unless you register the dependenc in the providers in appModule.tsand as we use the dependency in posts.component.ts :

@Component({

  selector: 'app-posts',

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

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

})

export class PostsComponent {

    constructor(http: HttpClient) {

    }

}

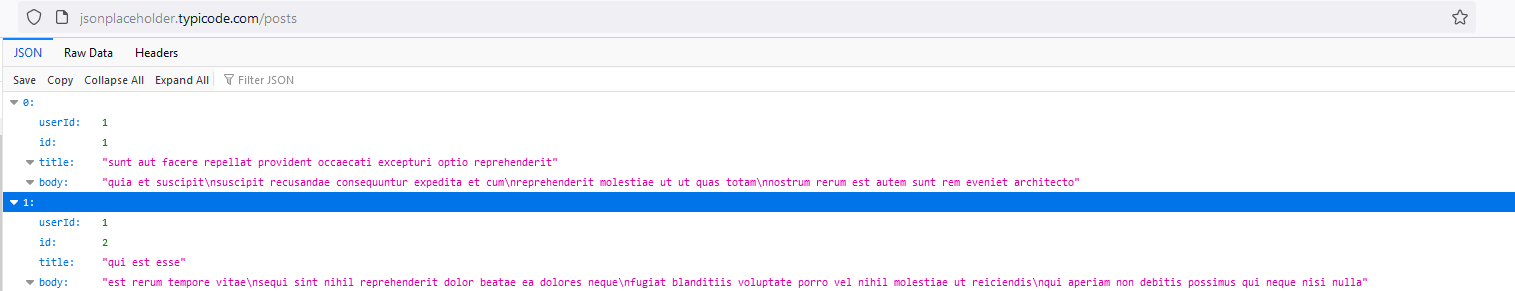
We want to have the htpClient service class but because this HttpClientService has other dependencies we register the HttpClientModule so that all the dependencies inside Http serivice is handled and we don’t need to register all the dependies that http class has inside providers in ngModule.

  constructor(http: HttpClient) {

      http.get('http://jsonplaceholder.typicode.com/posts')

    }

We use observables and promise for async operation (non blocking) and when the result is ready we will be notifieid by subscribe method :



export class PostsComponent {

    constructor(http: HttpClient) {

      http.get('http://jsonplaceholder.typicode.com/posts').subscribe(response => {

        let json = JSON.stringify(response);

        console.log(json);

      });

    }

}

**HttpGet with Subscribe method :**

export class PostsComponent {

  posts: any;

  private url = 'http://jsonplaceholder.typicode.com/posts';

    constructor(private http: HttpClient) {

      http.get(this.url).subscribe(response => {

        this.posts = response;

      });

    }

}

**HttpPost with Subscribe method :**

    createPost(input: HTMLInputElement) {

      let post = { title: input.value }

      this.http.post(this.url, JSON.stringify(post))

        .subscribe(response => {

          post['id'] = response['id'];

          this.posts.push(post);

          console.log(response);

      })

    }

<input

    (keyup.enter)="createPost(title)"

    #title

    type="text"

    class="form-control">

**we can use splice method for putting the object on the first index**

  createPost(input: HTMLInputElement) {

      let post = { title: input.value }

      this.http.post(this.url, JSON.stringify(post))

        .subscribe(response => {

          post['id'] = response['id'];

          this.posts.splice(0, 0, post);

          console.log(response);

      })

Splice(0,0, post)

First arg is the index

Second arg is the number to delete

Third is the object

**HttpPull with Subscribe method :**

We use patch for updating part of the object and pull for updating entire object

  this.http.patch(this.url, JSON.stringify({ isRead : true }));

      this.http.put(this.url, JSON.stringify(post));

  updatePost(post) {

      this.http.patch(this.url + '/' + post.id, JSON.stringify({ isRead : true }))

        .subscribe(response => {

          console.log(response);

        });

    }

**HttpDelete**

  deletePost(post) {

      this.http.delete(this.url + '/' + post.id)

        .subscribe(response => {

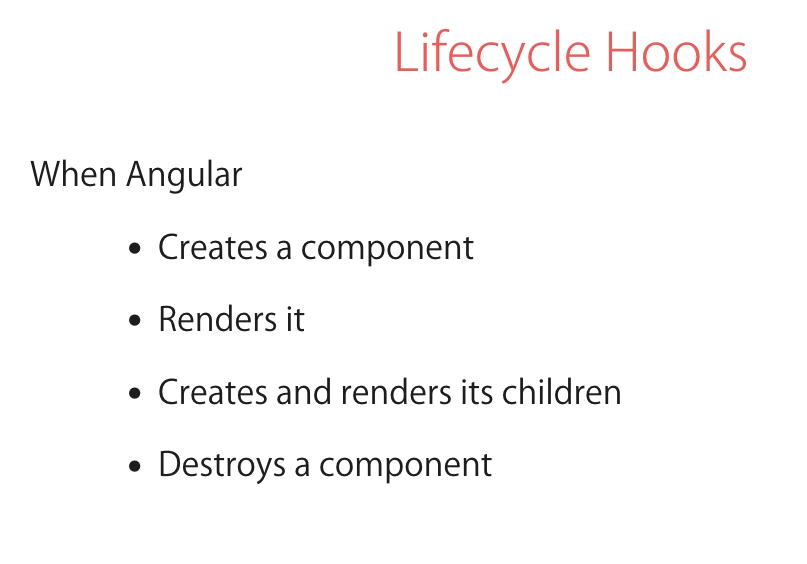
          let index = (this.posts as any[]).indexOf(post);

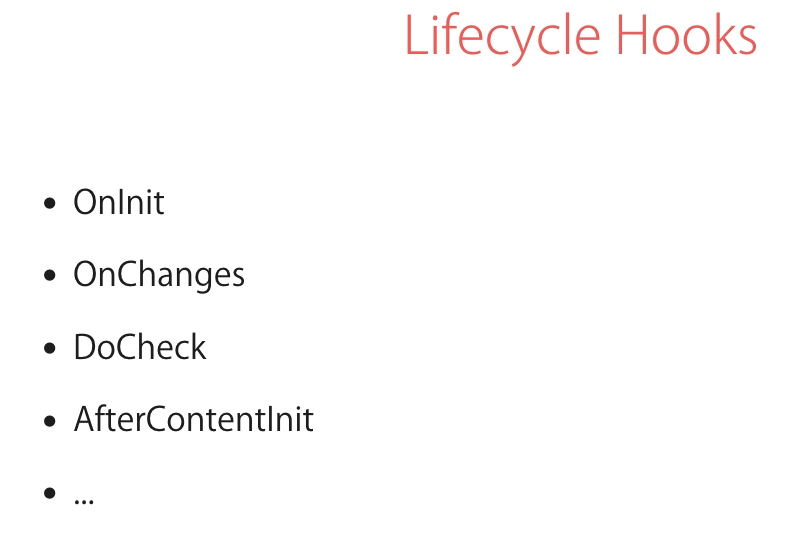
          this.posts.splice(index, 1);

        });

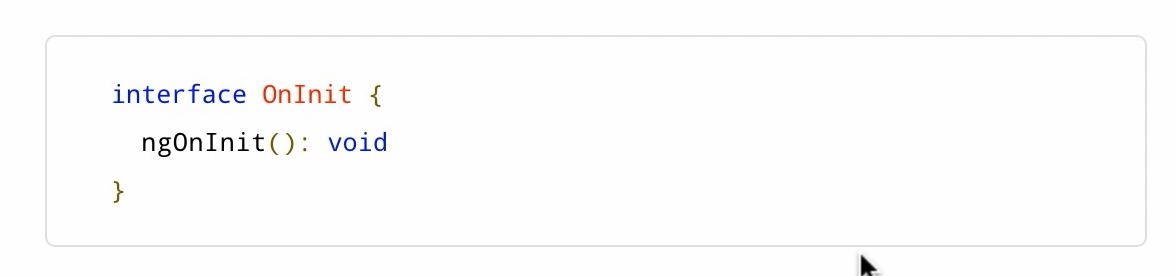
We should not perform expensive opertation in constructors

So we should use life cycle hooks :





Usually all the life cycle hooks start with ng



The interface that angular performs when it initialize the component;

export class PostsComponent implements OnInit {

  posts: any;

  private url = 'http://jsonplaceholder.typicode.com/posts';

    constructor(private http: HttpClient) {

    }

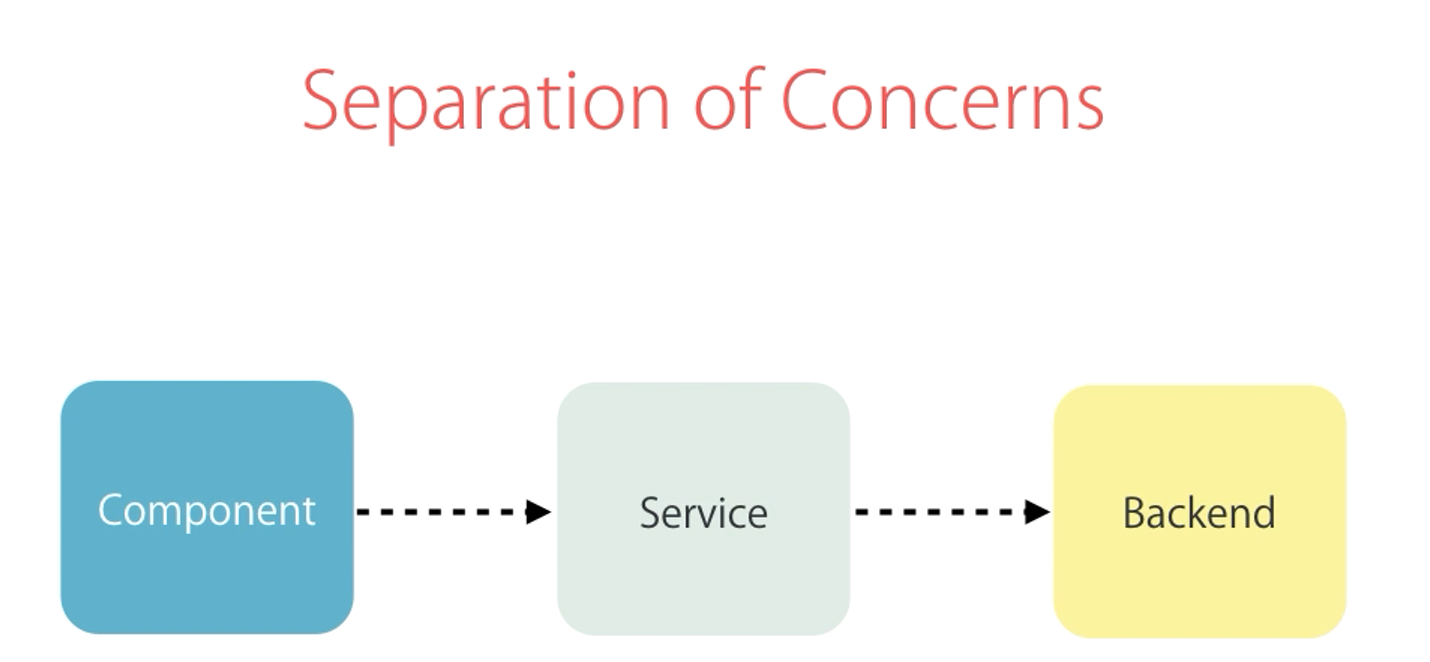
  ngOnInit(): void {

    this.http.get(this.url).subscribe(response => {

      this.posts = response;

    });

  }



Extracting Service

ng g s post

angular will not add service in Providers in ngModule so we should do it manually not to have problem when injecting this service in components

 providers: [

    CoursesService,

    AuthorsService,

    PostService

  ],

getPosts() : Observable<any> {

    return this.http.get(this.url);

   }

   createPost(post): Observable<any> {

    return this.http.post(this.url, post);

   }

   updatePost(post) {

    return this.http.patch(this.url + '/' + post.id, JSON.stringify({ isRead: true }));

   }

   deletePost(id) {

    return this.http.delete(this.url + '/' + id);

   }

  ngOnInit(): void {

    this.service.getPosts().subscribe(response => {

      this.posts = response;

    });

  }

    createPost(input: HTMLInputElement) {

      let post = { title: input.value }

      this.service.createPost(JSON.stringify(post))

        .subscribe(response => {

          post['id'] = response['id'];

          this.posts.splice(0, 0, post);

          console.log(response);

      })

    }

    updatePost(post) {

      this.service.updatePost(post)

        .subscribe(response => {

          console.log(response);

        });

    }

    deletePost(post) {

      this.service.deletePost(post.id)

        .subscribe(response => {

          let index = (this.posts as any[]).indexOf(post);

          this.posts.splice(index, 1);

        });

    }

Handling errors

  ngOnInit(): void {

    this.service.getPosts().subscribe(response => {

      this.posts = response;

    }, error => {

      alert("An unexpected error occured.");

    });

  }

    deletePost(post) {

      this.service.deletePost(post.id)

        .subscribe(

          response => {

          let index = (this.posts as any[]).indexOf(post);

          this.posts.splice(index, 1);

        },

         (error : Response) => {

          if(error.status === 404) {

            alert('This post has already been deleted')

          }

          else{

            alert("An unexpected error occured.");

            console.log(error);

          }

        });

    }

Annotate the error with response type causes it to have intellisense enabled on it.

We can use errors come from the server in reactive form :

   updatePost(post) {

      this.service.updatePost(post)

        .subscribe(response => {

          console.log(response);

        },

        (error : Response) => {

          if(error.status === 400) {

            this.form.setErrors(error.json());

          }

          else{

            alert("An unexpected error occured.");

            console.log(error);

          }

        });

    }

We should have the error handling inside service not component



We need to use catch operator on observable and so import it on the top like this :

import 'rxjs/add/operator/catch';

We can show the rror in the form via reactiveForm object :

  (error : Response) => {

            if(error.status === 400) {

              this.form.setErrors(error);

            }

            alert("An unexpected error occured.");

            console.log(error);

       })

It’s better to work with response object in the service it self and work with the object that is part of application domain in the component it self and not something that is part of httpProtocol

Observable has some useful methods called operators but they are not there by default so wee need to import them . one of them that comes inhandy is called catch. (import syntax is a little bit different)

import 'rxjs/add/operator/catch';

we use another way in newer angular :

deletePost(id) {

    return this.http.delete(this.url + '/' + id)

    .pipe(catchError(err => of(new AppError(err))));

   }

export class AppError {

    constructor(private \_originalError? : any) {

    }

}

Catching Response error code in service and throw domain specific Error object to be used in component

   deletePost(id) {

    return this.http.delete(this.urlAlt + '/' + id)

      .pipe(catchError(

          ( err: Response ) => {

            if(err.status === 404) {

              throw(new NotFoundError(err));

            }

            throw(new AppError(err));

          }));

   }

 deletePost(post) {

      this.service.deletePost(-200).subscribe({

        next: response => {

          let index = (this.posts as any[]).indexOf(post);

          this.posts.splice(index, 1);

        },

        error: (error : AppError) => {

          if (error instanceof NotFoundError) {

            alert("This post has already been deleted");

          }

          else{

            alert("An unexpected error occured.");

            console.log(error);

          }

      }});

    }

Factory methods : create objects

   deletePost(id) {

    return this.http.delete(this.urlAlt + '/' + id)

      .pipe(catchError(

          ( err: Response ) => {

            if(err.status === 404) {

              return throwError(() => new NotFoundError(err))

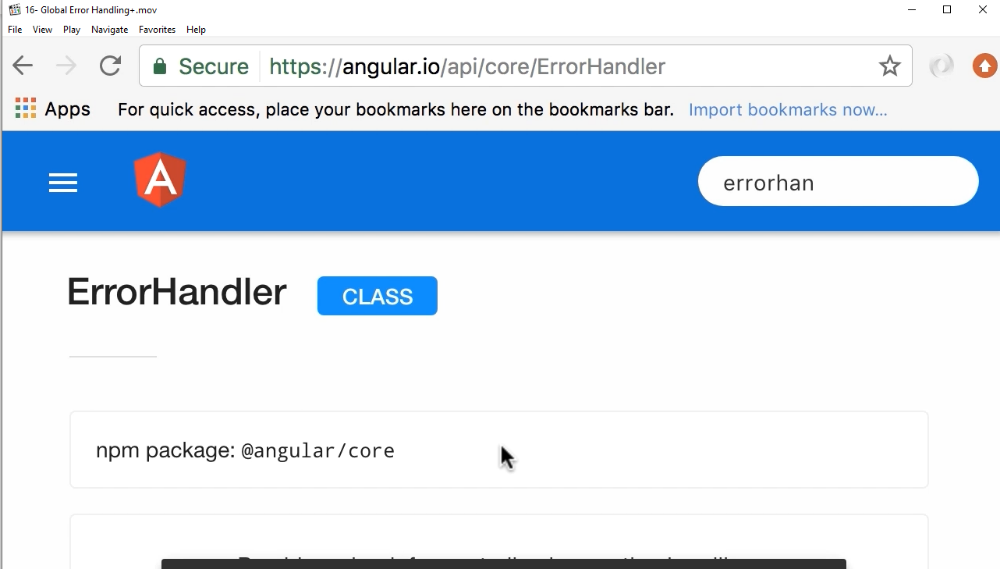
            }

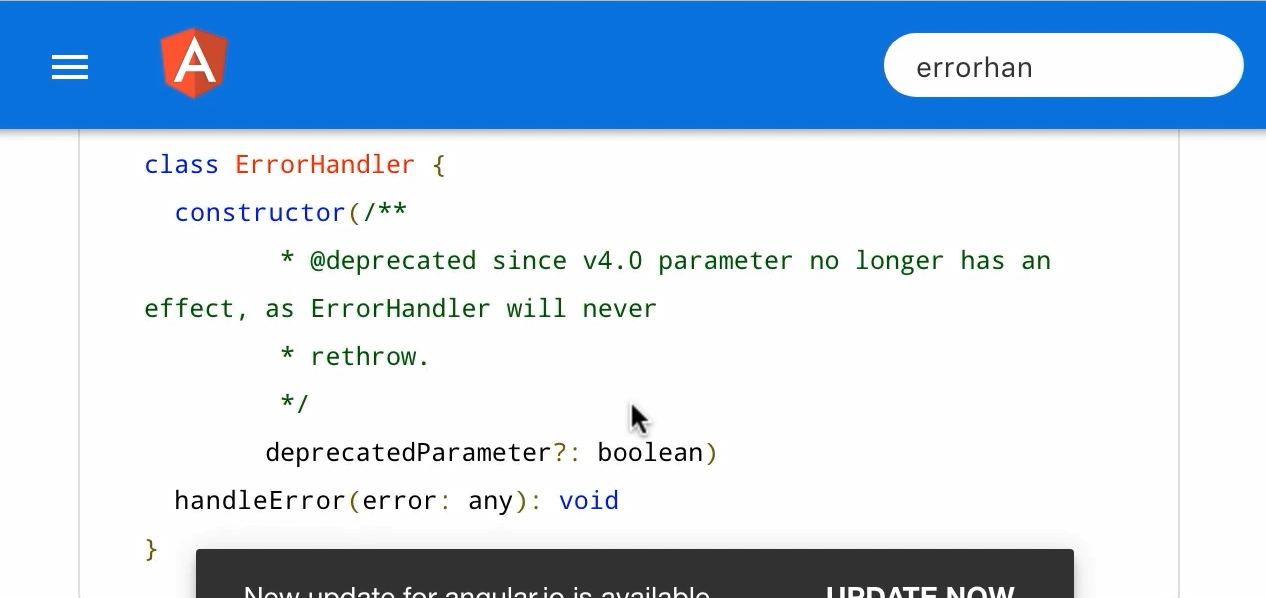
            return throwError(() => new AppError(err))

          }));

   }

Global error handler





import { ErrorHandler } from "@angular/core";

export class AppErrorHandler implements ErrorHandler {

    handleError(error: any) {

        alert("An unexpected error occured.");

        console.log(error);

    }

}

We want to tell angular to use AppErrorHandler instead of ErrorHandler globally

  providers: [

    CoursesService,

    AuthorsService,

    PostService,

    { provide: ErrorHandler, useClass: AppErrorHandler}

  ],

Provider param is the class to replace and useClass is the new class.

Instead of this we want to propogate the error :

  ngOnInit(): void {

    this.service.getPosts().subscribe(response => {

      this.posts = response;

      console.log(response);

    }, error => {

      alert("An unexpected error occured.");

      console.log(error);

    });

  }

We remove the error arrow function part so that when throwing an error it will propogates and reach ErrorHandler;

 ngOnInit(): void {

    this.service.getPosts().subscribe(response => {

      this.posts = response;

      console.log(response);

    }, error => {

      alert("An unexpected error occured.");

      console.log(error);

    });

  }

  ngOnInit(): void {

    this.service.getPosts().subscribe(response => {

      this.posts = response;

      console.log(response);

    });

  }

In another case :

We do it by throwing the error once again.

 createPostAlt(input: HTMLInputElement) {

      let post = { title: input.value }

      this.service.createPost(JSON.stringify(post))

        .subscribe({

          next: (response) => {

            post['id'] = response['id'];

            this.posts.splice(0, 0, post);

            console.log(response);

          },

          error: (err : AppError) => {

            if(err instanceof BadInput) {

              // this.form.setErrors(err.originalError);

            }

            else {

              throw err;

            }

          }

        });

We can refactor and extract errorHandling in private method and use it as a delegate

   private handleError(error: Response) {

    if(error.status === 404)

      return throwError(() => new NotFoundError(error));

    return throwError(() => new AppError(error)) ;

   }

 deletePost(id) {

    return this.http.delete(this.urlAlt + '/' + id)

      .pipe(catchError(this.handleError));

   }

Refactoring and making generic the DataService.ts

export class DataService {

  constructor(private url: string, private http: HttpClient) {}

   getAll() : Observable<any> {

    return this.http.get(this.url)

      .pipe(catchError(this.handleError));

   }

   create(resource): Observable<any> {

    return this.http.post(this.url, resource)

    .pipe(catchError( this.handleError));

   }

   update(resource) {

    return this.http.patch(this.url + '/' + resource.id, JSON.stringify({ isRead: true }))

      .pipe(catchError(this.handleError));

   }

   delete(id) {

    return this.http.delete(this.url + '/' + id)

      .pipe(catchError(this.handleError));

   }

   private handleError(error: Response) {

    if(error.status === 400)

      return throwError(() => new BadInput(error.json()));

    if(error.status === 404)

      return throwError(() => new NotFoundError(error));

    return throwError(() => new AppError(error)) ;

   }

}

@Injectable({

  providedIn: 'root'

})

export class PostService extends DataService {

  constructor(http: HttpClient) {

    super('http://jsonplaceholder.typicode.com/posts', http);

   }

Map operator

With map operator we can transform the response

export class DataService {

  constructor(private url: string, private http: HttpClient) {}

   getAll() : Observable<any> {

    return this.http.get(this.url)

      .pipe(

        map(response => response),

        catchError(this.handleError)

      );

   }

  ngOnInit(): void {

    this.service.getAll()

    .subscribe(posts => this.posts = posts

    );

  }

  constructor(private url: string, private http: HttpClient) {}

   getAll() : Observable<any> {

    return this.http.get(this.url)

      .pipe(

        map(response => response),

        catchError(this.handleError)

      );

   }

   create(resource): Observable<any> {

    return this.http.post(this.url, resource)

    .pipe(

      map(response => response),

      catchError( this.handleError)

      );

   }

   update(resource) {

    return this.http.patch(this.url + '/' + resource.id, JSON.stringify({ isRead: true }))

      .pipe(

        map(response => response),

        catchError(this.handleError)

        );

   }

For delete we don’t return anything

 delete(id) {

    return this.http.delete(this.url + '/' + id)

      .pipe(catchError(this.handleError));

   }

So we send empty parameter in next

    deletePost(post) {

      this.service.delete(-200).subscribe({

        next: () => {

          let index = (this.posts as any[]).indexOf(post);

          this.posts.splice(index, 1);

        },

        error: (error : AppError) => {

          if (error instanceof NotFoundError) {

            alert("This post has already been deleted");

          }

          else{

            throw error;

          }

      }});

    }

Optimistic and pessimistic operation

In optimistic we add in client side because most of the requests are successful. If a request didn’t make it to the server it throws error and in the callback function we roll back the changes.

  createPost(input: HTMLInputElement) {

      let post = { title: input.value };

      this.posts.splice(0, 0, post); // we add on top before calling server

      input.value = ''; // notice two way binding

      this.service.create(JSON.stringify(post))

        .subscribe({

          next: (newPost) => {

            post['id'] = newPost['id'];

          },

          error: (err : AppError) => {

            this.posts.splice(0, 1); // rollback changes at the top

            if(err instanceof BadInput) {

              // this.form.setErrors(err.originalError);

            }

            else {

              throw err;

            }

          }

        });

    }

   create(resource): Observable<any> {

    return throwError(d => new AppError(d)); //simulating if server returns error

    // return this.http.post(this.url, resource)

    // .pipe(

    //   map(response => response),

    //   catchError( this.handleError)

    //   );

   }

Optimistic delete :

deletePost(post) {

      let index = (this.posts as any[]).indexOf(post);

      this.posts.splice(index, 1);

      this.service.delete(-200).subscribe({

        next: () => {

        },

        error: (error : AppError) => {

          this.posts.splice(index, 0, post);

          if (error instanceof NotFoundError) {

            alert("This post has already been deleted");

          }

          else{

            throw error;

          }

      }});

    }

 delete(id) {

    return throwError(d => new AppError(d));

    // return this.http.delete(this.url + '/' + id)

    //   .pipe(catchError(this.handleError));

   }