ANGULAR

# Introduction

Angular is a framework for building client applications in HTML, CSS, and JavaScript/ TypeScript.

**Angular in the end is a JS framework, changing your DOM at runtime.**

TypeScript is more common in Angular as Angular itself runs in TypeScript.

### Why do we need Angular?

If we build our applications using JavaScript/JQuery, as our application becomes complex vanilla JavaScript/JQuery code is hard to maintain.

We need a way to properly structure our application, sure there are JavaScript patterns (Revealing Module Pattern, Prototype Pattern) but this pattern is hard to understand by lot of beginners and also Vanilla JavaScript or JQuery are hard to test.

That’s why for the past few years various frameworks have envelope like (Angular, React.js etc.) makes web application development easier.

### Benefits of Using Angular

1. Gives our applications a clean structure.
2. Includes a lot of reusable code
3. Makes our applications more testable.

### Architecture of Angular Applications

**Front end (User Interface)** which is builds with (**HTML, CSS, TypeScript, Angular**) is react with Back-end. Here we also have **HTML templates and Presentation logic**.

**Back-end(Data processing)** **here we have one or more database as well as bunch of HTTP Services(API) to make this data available to the clients or Front End**. Here we are also implementing **business logic**

**APIs** (**Endpoints that are accessible via the HTTP protocol**)

### Setting Up the Development Environment

1. Install the latest version of NODE (**minimum version required in node for ANGULAR is v6.9**)
2. Install Angular CLI (command line interface) using NPM (**npm i -g @angular/cli**)**.** For checking angular is installed or not (**ng --version**)

### First Angular App

1. Creating a new project (**ng new hello-world**)
2. Run and compile the Angular application (**ng server** )

### Structure of Angular Project

1. **Folder e2e** (end to end) – Here we write end to end test for our application. e2e test is basically an automated test that simulates a real user. (Like launch the browser, navigate to home page etc.)
2. **Folder node\_modules** – Here we store all the third party libraries that the application may depend upon (this folder is purely for development while deployment to prod this folder libraries will bundle)
3. **Folder src** – Here we will have actual source code of our application.

* **Folder app** – Here we have module and component, every angular application will have at least have one module and component.
* **Folder assets** – Here we store the static assets of our application like (images, text file etc.)
* **Folder environments** – Here we store configuration setting for different environment.
* **File favicon.ico** – favorite which icon we displayed in browser.
* **File Index.html** - This is very simple html file that contains our angular application. Here we don’t have any references to script or stylesheet; this reference will be dynamically inserted into this page.
* **File main.ts** – It’s a typescript file basically the starting point of our application, all we doing here is bootstrapping the main module of our application.
* **File polyfills.ts** – which basically imports some scripts that are required for running angular, because angular framework uses features of JavaScript that are not available in the current version of JavaScript supported by most browser out there. So polyfills fill the gap between the features of JavaScript that Angular needs.
* **File styles.css** - Here we add the global styles for our application.
* **File test.ts** – Basically use for setting our testing environment.

1. **File .angular-cli.json** – Here we have configuration file for Angular CLI its very standard configuration.
2. **File .editorconfig** – Basically use to manage all developers to use same setting in the editor.
3. **File .gitignore** - It’s use for excluding certain file from git repository.
4. **File Karma.conf.js** – it’s a configuring file for karma which is use for testing JavaScript file.
5. **File Package.json** – This is the standard file that every NODE project will have versioning, dependency libraries , for devdependencies libraries etc.
6. tsconfig.json - This file is used for TypeScript setting which will compile TypeScript to es5 which the browser can understand.
7. tslint.json

### Webpack

Angular CLI uses the tool webpack, it gets all style sheet combines them puts in a bundle and then minimize the bundle. Whenever you change anything in Angular project webpack will automatically compiled successfully.

# TypeScript Fundamentals

### What is TypeScript?

TypeScript is not entirely new it’s a super set of JavaScript (so any valid JavaScript code will also valid for TypeScript). But TypeScript some extra features like:-

1. **Strong Typing or static typing** (like type of variables) in TypeScript this feature is optional.
2. **Object-oriented features** (class, interfaces, constructors, access-modifiers, fields, properties etc.)
3. **Compile-time errors** can be finding in TypeScript not all types of error but lots of error.
4. **Great tooling**

### Your First TypeScript Program

Installing TypeScript globally (**npm i -g typescript**) checking TypeScript version (**tsc –version**).

If we write Javascript code in TypeScript file and run with (**tsc main.ts**) it will create one more file main.js. But in Angular this process happens automatically when we run with ng serve

### Declaring Variables

In TypeScript we have two ways to declare variable

//In TypeScript we have two ways to declare variable

//Using var but don't use in TypeScript as i scope is present to nearest function

// doSomething()

function doSomething() {

for (var i = 0; i<5; i++)

{

console.log(i);

}

console.log('Finally: '+ i);

}

doSomething();

//using let we will be using let to declare variable in TypeScript as the scope i will be present inside

// for loop only not in the finally statement of console

// it will give error while running the TypeScript file but it will create correct JavaScript file

function doSomethingWithLet() {

for (let i = 0; i<5; i++)

{

console.log(i);

}

console.log('Finally: '+ i);

}

doSomethingWithLet();

### Types

let a: number;

let b: boolean;

let c: string;

let d: any;

let e: number[] = [1, 2, 3];

let f: any[] = [1,true, 'a', false]; // it is not good practice but we can do

// working with group of related constant

const ColorRed = 0;

const ColorGreen = 1;

const ColorBlue = 2;

//Enum

// Here above codes can be replace with Enum

enum Color {Red = 0, Green = 1, Blue = 3}; // best practice we should we should declare the value

let backgroundcolor = Color.Blue; // here automatically we will get 3

### Type Assertions

Setting the variable type to particular datatype, so that we can use the intelligent

// Two ways to do Type Assertions

//1)

let message;

message = 'abc';

let endwithC = (<string>message).endsWith('c');

//2)

let alternativeWay = (message as string).endsWith('c');

### Arrow Functions

let log = function(message) {

console.log(message);

}

// Using Arrow function in TypeScript

// In C# we called as Lamda expression

let arrowLog = (message) => console.log(message);

### Interfaces

// with this interface I am defining the shape of point objects

// always use Pascal naming cases

interface Point {

x: number;

y: number;

}

let drawPoint = (point: Point) => {

}

drawPoint({

x: 1,

y: 2

})

### Class

Above example in Interface **will leads to violation of Cohesion** (things that are one unit they should so together).

So we required class to Group variables (properties) and functions (methods) that are highly related.

// with this interface I am defining the shape of point objects

// always use Pascal naming cases

class Point {

x: number;

y: number;

draw() {

//..

}

getDistance(another: Point) {

//..

}

}

### Objects

If we declare a variable and use custom type will leads to runtime error.

class Point {

x: number;

y: number;

draw() {

//..

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

}

getDistance(another: Point) {

//..

}

}

let point: Point;

point.draw();

# Angular Fundamentals

### Building Blocks of Angular Apps

At the heart of Angular App we have one or more components. **Component** encapsulates the **Data**, **HTML Template** and **Logic of View which is the area of the screen that the user sees.**

**Modules** is a container for a group of related components and every angular will have at least one Module which is called App module

### Components

There are basically three steps we need to follow in order to use a component:-

1. Create a component.
2. Register it in a module.
3. Add an element in a HTML markup

* Create a Component (step by step) :-

1. Create a type script file (courses.component.ts)
2. Export a class with Component using Component module from @angular/core

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

@Component ({

selector: 'courses',// <div id ="courses"> selector will be "#courses"

// or <div class="courses" selector will be ".courses"

template: '<h2>Courses</h2>' // we can use different file for template

})

export class CoursesComponent {

}

* Register it in a module (step by step) – Currently this Hello-world having one module i.e. app module and declare the new component in **declarations**(**here only we add all the components that are part of this module**) and automatically import for new component will be appear at top

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

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

import { AppRoutingModule } from './app-routing.module';

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

import { CoursesComponent } from './courses.component';

@NgModule({

declarations: [

AppComponent,

CoursesComponent

],

imports: [

BrowserModule,

AppRoutingModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

* Add an element in a HTML markup

Here in the component selector is courses, which means anywhere have element like courses, angular will render the template of courses inside the element

@Component({

selector: 'courses',

template: '<h2>Courses<h2>'

})

But where we add this course element, in app.component.html (this is the external template for app component.

<h1>Angular</h1>

<courses></courses>

### Generating Components Using Angular CLI

In Terminal just type – **ng g c course**. Here course is component name

### Templates

In Template we can get the value declared in class using {{name of variable or function}} which is called **string interpolation**

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

@Component({

selector: 'courses',

template: '<h2>{{ getTitle() }}<h2>'

})

export class CoursesComponent {

title = "This is courses";

getTitle() {

return this.title;

}

}

### Directives

We use Directives to manipulate the DOM; we can use them to add a DOM element or remove an existing DOM element or change the class of a DOM element for its style and so on.

Eg \*ngFor we use like an attribute and **use of asterisk (\*) whenever we want to modify the structure of DOM**.

**Benefits of back tick (`) in Templates is to break the statement into multiple lines**

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

@Component({

selector: 'courses',

template: `

<h2>{{ getTitle() }}<h2>

<ul>

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

{{ course }}

</li>

</ul>

`

})

export class CoursesComponent {

title = "This is courses";

courses = ["course1","course2","course3"]; // for live application we get from database where we use Service

getTitle() {

return this.title;

}

}

### Services

|  |
| --- |
|  |

In the above example instead of hardcode the courses value, we get it from server via HTTP service.

We can do it by 2 ways:-

1. Calling the HTTP service here in the component itself. But there are some problems with this logic.
2. **First problem is that this logic will tightly couple the component to that HTTP endpoint**. In future when we want write Unit Test case, we should not depends on live HTTP endpoint which will make harder to execute Unit Test.
3. **Second problem is that may be somewhere in the application we are using the courses records, so again we have to write the HTTP endpoint logic**.
4. **Third problem is that component should not include other logic except presentation logic**.
5. **For this entire problem have use services for this in Angular**, so we are going to define a separate class which we called as service and then there we will add this logic for getting the list of courses via an HTTP service. (We can reuse in multiple places).

Create a file **course.service.ts**

// Here in the we don't need any decorator like component decorator

export class CourseService {

getCourses() {

// for now we are not calling any HTTP service for reduce complexity

// Later we will see via consuming HTTP service retrieving the courses records

return ["course1","course2","course3"];

}

}

### Dependency Injection

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

import { CourseService } from './courses.service'

@Component({

selector: 'courses',

template: `

<h2>{{ getTitle() }}<h2>

<ul>

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

{{ course }}

</li>

</ul>

`

})

export class CoursesComponent {

title = "This is courses";

courses; // for live application we get from database where we use Serice

constructor(service: CourseService) {

//let service = new CourseService();

this.courses = service.getCourses();

}

Above code **let service = new CourseService();** will leads to problems:

1) By using new operator here we have tightly coupled this courses component to the courses service

2) In future whenever we decided to add parameter to the constructor of courses service, we need lots of change in other place wherever the courses service is called.

**For this problem instead of creating the instance of course service, we can ask Angular to do**.

**So delete the above code and add parameter to this constructor**.

**With this when Angular is going to create instance of component it looks at this constructor and finds this constructor has dependency on type CourseService.** Will lead to decouple.

**We have not done yet, we need to instruct Angular to create an instance of courses service and pass to courses component. This concept is called Dependency Injection**

**We should instruct Angular to inject the dependencies of the component into its constructor.**

**Angular has dependency Injection framework built into it (so for that we need register the dependency somewhere in module)**

In **app.module.app** add the **CourseService** under **providers** as highlighted.

@NgModule({

declarations: [

AppComponent,

CoursesComponent,

CourseComponent

],

imports: [

BrowserModule,

AppRoutingModule

],

providers: [

CourseService

],

bootstrap: [AppComponent]

})

export class AppModule { }

**In memory we are going to have one single instance of courses service and Angular will pass this same instance to all this components. This is we called singleton pattern.**

**With all this implementation when Angular is going to create the instance of component. First it wills instantiated its dependencies and then it will inject those dependencies into the constructor of the class. This is all we called dependency injection.**

### Generating Services using Angular CLI

In Terminal just type – **ng g s email**. Here email is service name

Here we have something new in that typescript file, that is **injectable decorator** we need this decorator only if this **service had dependency in it constructor.**

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

@Injectable({

providedIn: 'root'

})

export class EmailService {

constructor() { }

}

# Displaying Data and Handling Events

|  |
| --- |
|  |

### Property Binding

We use string interpolation {{ }} previously, this {{}} is just a syntactical sugar, behind the screen when **Angular compile this templates it’s translate this string interpolation into property binding**.

With property binding we bind a property of DOM element like src to field or property in our component

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

@Component({

selector: 'courses',

template: `

<h2>{{ getTitle() }}<h2>

<img src="{{ imageUrl }}" />

<img [src] = "imageUrl" />

`

})

export class CoursesComponent {

title = "This is courses";

courses; // for live application we get from database where we use Serice

imageUrl = "https://media.gettyimages.com/photos/rio-de-janeiro-brazil-picture-id672036306";

}

**String Interpolation works well for adding dynamic values into headings or div, span, p (where we want to render text)**

**Property Binding will works only way component to the DOM, but any changes in DOM will not reflected in back in the component.**

### Attribute Binding

Check the DOM vs HTML in Extra section.

**Most of the attributes of HTML elements have a one to one mapping to property of DOM object like src in image.**

**But there are some attribute which don’t have HTML representation in the DOM** like colspan for this we have different syntax as highlighted below

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

@Component({

selector: 'courses',

template: `

<table>

<tr>

<td [attr.colspan]="colspan" > </td>

</tr>

</table>

`

})

export class CoursesComponent {

title = "This is courses";

colspan = 2;

}

### Class Binding

Want to add additional class to an element based on some condition, by using class binding we can do that as highlighted below.

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

@Component({

selector: 'courses',

template: `

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

`

})

export class CoursesComponent {

isActive = true;

}

### Style Binding

Style binding which is a variation of property binding and very similar to class binding

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

@Component({

selector: 'courses',

template: `

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

`

})

export class CoursesComponent {

isActive = true;

}

### Event Binding

So far we have learned property binding and its variation (class, style binding) to act something in the DOM (to display data).

Event binding which we use to handle events raise from the DOM like (keystroke, Mouse Moments, clicks etc.) **for handle events instead of [] we use () as highlighted below**

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

@Component({

selector: 'courses',

template: `

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

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

</div>

`

})

export class CoursesComponent {

onClickDiv() {

console.log("Div was clicked");

}

onSave($event) {

$event.stopPropagation();

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

}

}

Sometimes we need access to the event object that was raise in the event handler (e.g. with mouse movements event object will tell us the x and y position), if we want access to event object, then need to add as a parameter in function **onSave($event)** and also calling we also use **$event**

This $event object is something known to angular which is same use in Vanilla JavaScript. In future we will create custom event that will use in our application.

How can we stop event **bubbling**? Using **$event.stopPropagation** meansthis event will not bubble up in other word is not going to hit the second handler.

### Event Filtering

Without using filter in below code

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

@Component({

selector: 'courses',

template: `

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

`

})

export class CoursesComponent {

onKeyEvent($event) {

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

}

}

With using filter

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

@Component({

selector: 'courses',

template: `

<input (keyup.enter)="onKeyEvent()" />

`

})

export class CoursesComponent {

onKeyEvent() {

console.log("Enter key was pressed");

}

}

### Template Variables

From the above example how to print the entered value in the input field?

Without Template variables

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

@Component({

selector: 'courses',

template: `

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

`

})

export class CoursesComponent {

onKeyEvent($event) {

console.log($event.target.value);

}

}

Using Template variables

In Angular we declare Template variables using #variable\_name

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

@Component({

selector: 'courses',

template: `

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

`

})

export class CoursesComponent {

onKeyEvent(email) {

console.log(email);

}

}

### Two-way Binding

Two-way binding means it value can pass from **component -> view** as well **view -> component**

In Angular we have special syntax to implements two-way binding **[(ngModel)]**

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

@Component({

selector: 'courses',

template: `

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

`

})

export class CoursesComponent {

email = "abc@gmail.com"

onKeyEvent() {

console.log(this.email);

}

}

**Alternative of highlighted code is without using ngModel**

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

**NgModel will not unless and until we imports FormsModule**

But the above code for Model will give error (**Can't bind to 'ngModel' since it isn't a known property of 'input'**). **For that we need to import formModule app.module.ts**

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

And then we can import in our main app module

imports: [

BrowserModule,

AppRoutingModule,

FormsModule

],

### Pipes

We use pipes to format data. Two types of Pipes:-

1. Built-in pipes (Uppercase, lowercase, decimal, currency, percent etc.)
2. Custom pipes

**We use pipes with pipe | symbol, and also can use multiple pipes in single variable**

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

@Component({

selector: 'courses',

template: `

{{ course.title | uppercase}} <br/>

{{ course.rating | number:'1.1-1'}} <br/>

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

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

{{ course.releaseDate | date:'shortDate'}} <br/>

`

})

export class CoursesComponent {

course = {

title : "Angular is awesome",

rating : 8.15,

students: 33783,

price: 989.22,

releaseDate : new Date(2016,3,1)

}

}

### Custom Pipes

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

@Component({

selector: 'courses',

template: `

{{ text | summary}}

`

})

export class CoursesComponent {

text = `

dsdhjsd djjsjdh sdjhdjs hsjhdj gewyuweu ueqoodjskdk euywueysdj eyyshdjsdh tyqwywqy oikkm uhuhiiiii ahahsgdhsg shdgdhs sdhdhsdghsd shdjhsdjsdh ueueuwyuwey oiwoqieihqejd q quhqjh jsjjfj hdd hhd hdddhd dhdhyweiwiqoioqewhjscb uiieuiweu

`

}

For adding custom pipe in text called summary:-

1. Add a typescript file summary.pipe.ts
2. In this file we need to import a couple of types from angular **Pipe** decorator function and other is **PipeTransform** which is an interface ( that defines the shape of all pipes in Angular )

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

1. Just like components , services for pipe also exports a typescript class and need to **implements the PipeTransform interface** ( with this we are telling to Angular this class should be exactly same shape as the pipe transform.

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

export class SummaryPipe implements PipeTransform {

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

}

}

1. Finally we need to apply the @Pipe() decorator function and name as summary which is the keyword that we are going to use in our html markup

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

@Pipe ({

name : 'summary'

})

export class SummaryPipe implements PipeTransform {

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

}

}

1. Implement something in transform

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

@Pipe ({

name : 'summary'

})

export class SummaryPipe implements PipeTransform {

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

if(!value)

return null;

value.substr(0,10) + '...';

}

}

It’s not over we will get error “**The pipe 'summary' could not be found**”, because we define this custom class but forgot to register this in app.module.ts

@NgModule({

declarations: [

AppComponent,

CoursesComponent,

CourseComponent,

AuthorsComponent,

SummaryPipe

],

And automatically it will import the summary type on top of app.module.ts

import { SummaryPipe } from './summary.pipe';

Now how to pass parameter from component to custom pipe

1. Pass the parameter from component as show below

@Component({

selector: 'courses',

template: `

{{ text | summary:10 }}

`

})

1. In custom pipe use the second parameter to get the parameter from component

export class SummaryPipe implements PipeTransform {

transform(value: string, **limit**?: any) {

if(!value)

return null;

let actualLimit = (limit) ? limit : 15;

return value.substr(0,actualLimit) + '...';

}

}

### Async Pipes

{{appStatus | **async** }}

Async is the build in pipes , basically it recognized that appStatus is a promise, it will also work with observable , it will recognize that something change.

# Building Re-usable Components

### Component API

Suppose in the favorite example we want to get the value from server. So for that we get the value in App.component.ts from post object as below

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

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

title:string;

post = {

title: "Title",

isFavorite: "true"

}

}

In the template of app.component.ts we want to display this post and if it set as favorite, we will render the favorite icon as a full star.

Here unfortunately we cannot use property binding to bind this isFavorite field of the favorite component to post object of (post.isFavorite). Below code will give error

**“Can't bind to 'isFavorite' since it isn't a known property of 'favorite'.”**

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

Even though in favorite component we have this field **[isFavorite]** and this a public field. Because in **Angular template in order to use property binding we need define that property or field as an input property.**

**To make this favorite component more reusable we are going to add support for property and event binding. For example in future we want to notify whenever clicks on this favorite component**

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

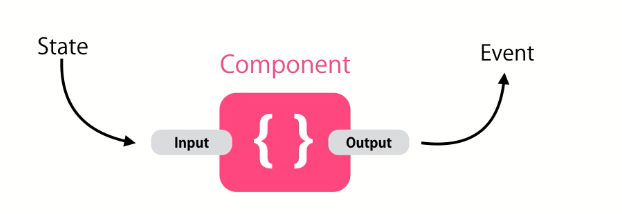
Example Favorite component raise a change event (change) and we want to be notified and call a method in the host component, in this case app.componet.ts

Inside the method **onFavoriteChange** we can call the server or do something else.

**To add support to event binding we need to define a special property in this favorite component that refer to output property**

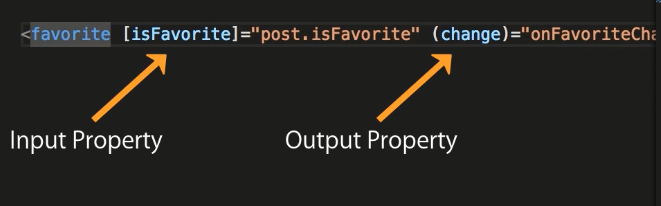
**In other words in order to make a component more reusable, we need to add a bunch of input and output properties.**

**We use input properties to pass input state to a component and we use output properties to raise events from this custom component.**

****

**The combination of input and output properties for a component makes up a public API of that component.**

**Now our favorite component does not currently have a public API because it does not have any input and output properties. In other words we cannot use property and event binding here.**

****

### Input Properties

In Favorite Component we want to mark the field **isFavorite** as an input property, so we can use as a property binding expression. Two ways to mark this as an input property:-

**Frist Approach**:-

1. We need to import a decorator function on top from angular/core library called **Input in Favorite.component.ts**

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

1. Then use this input to annotate this isFavorite field

@Input() isFavorite:boolean;

Second Approach:-

First we don’t need the Input Decorator. Instead we can declare this field as an input property in the component Meta data in Favorite.component.ts. Here in the array we can list all the fields and properties that should be input properties.

@Component({

selector: 'favorite',

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

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

inputs: ['isFavorite']

})

But there is a problem with this approach, problem is use of this magic string here this code as long as we have field or properties present in our class called isFavorite . If in future we rename this field to something due to refactoring the entire place it will update but the magic string will not update.

**So better use the First Approach**.

Complete code Favorite.Component.ts code

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

@Component({

selector: 'favorite',

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

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

})

export class FavoriteComponent implements OnInit {

@Input() isFavorite:boolean;

constructor() { }

ngOnInit() {

}

onClick()

{

this.isFavorite = !this.isFavorite;

}

}

And Favorite.component.html

<span

class="glyphicon"

[class.glyphicon-star] = "isFavorite"

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

(click) = "onClick()"

></span>

In App.Component.html

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

In App.Component.ts for the post object

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

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

title:string;

post = {

title: "Title",

isFavorite: "true"

}

}

### Aliasing Input Properties

How to give input properties an alias or nickname?

We have declared the isFavorite properties using camel case notation because in Javascript or in Typescript we always use camel case for properties. But in your application you don’t want to mark camel case in HTML markup as highlighted

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

**But instead of isFavorite we want to use as is-favorite as above in HTML Markup**. In Javascript or TypeScript we cannot have name like this. Solution is use alias to the input properties

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

**Using Alias is another benefit it keeps the contracts of API stables**. That’s means in future if you rename the property isFavorite to something else code will not break as we use alias to the Input properties.

**But one point keep in mind is that when we rename properties in component (favorite.component.ts) but the particular template field in template (favorite.component.html) is not update we have to update it manually.**

### Output Properties

Now I want to be notifying when the user clicks on the Favorite component. So raise a custom event like (change) that we can bind to a method in our host component (here app.component.ts). In app.component.ts I want a method “onFavoriteChange”, this method will render whenever the favorite component raises a change event.

<favorite [is-favorite]="post.isFavorite" (change) = "onFavoriteChange()"></favorite>

So for that we have to use output properties in Favorite.component.ts

1. First include the Output decorator function from @angular/core

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

1. Then declare a field and the name of the field should be exactly same name as the event we want to raise and decorate the field with @output decorator , also initialize this field to an instance of the event emitter class and need to import the EventEmitter at top as highlighted below:

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

@Output() change = new EventEmitter();

1. Finally in our onClick() method after we toggle the isSelected field , we want to raise an event using the emit() and we use that method to raise or publish an event

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('is-favorite') isFavorite:boolean;

@Output() change = new EventEmitter();

constructor() { }

ngOnInit() {

}

onClick()

{

this.isFavorite = !this.isFavorite;

this.change.emit();

}

}

Now after change the event in Favorite.component.ts in onClick() function , then the function onFavoriteChange() will be called in app.component.ts

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

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

title:string;

post = {

title: "Title",

isFavorite: "true"

}

onFavoriteChange()

{

console.log("Favorite changed");

}

}

### Passing Event Data

We need to pass some data when raising an event from favorite.component.ts.

So from favorite component **when we are emitting event we can optionally pass some value and this value will be available to all subscriber of this event (in this case subscriber of change event is app.component.ts)**

1. **Frist add the parameter in the event object as highlighted below:-**

In Favorite.component.ts

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('is-favorite') isFavorite:boolean;

@Output() change = new EventEmitter();

constructor() { }

ngOnInit() {

}

onClick()

{

this.isFavorite = !this.isFavorite;

this.change.emit(this.isFavorite);

}

}

1. Now in the **app.component.ts** retrieve the parameter isFavorite

onFavoriteChange(**isFavorite**)

{

console.log("Favorite changed: ", **isFavorite**);

}

1. Last in **app.component.html** when we are handling the change event , we need to pass a $event object to the event handler

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

* **Below step instead of Boolean value if we are passing object**

In the step 1 we have a pass a simple Boolean value, instead of that we need to pass object this object can have property like newValue in **favorite.component.ts**

onClick()

{

this.isFavorite = !this.isFavorite;

this.change.emit({newvalue: this.isFavorite});

}

In the step 2 of **app.component.ts** which is subscriber of the change event of our favorite.component.ts, instead of simple Boolean we are going to receive an actual object, we can call as event arguments (eventArgs)

onFavoriteChange(eventArgs)

{

console.log("Favorite changed: ", eventArgs);

}

In the step 3 $event represent actual JavaScript object that has a property called new value. Code is same as above.

### Aliasing Output Properties

Alias is required if you change the name of the event emitter in consumer component (favorite.component.ts ) but in the subscriber template app.component.html old name is referring .

So to not break the code we can use alias in output parameter of Favorite.component.ts , which will be same name as (change) as highlighted below:-

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('is-favorite') isFavorite:boolean;

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

constructor() { }

ngOnInit() {

}

onClick()

{

this.isFavorite = !this.isFavorite;

this.click.emit({newvalue: this.isFavorite});

}

}

In App.component.html

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

### Templates

We can create template using inline as well as external template

Inline Template: - Always use inline template, if you code is less than 5 lines

@Component({

selector: 'courses',

template: `

{{ text | summary:10 }}

`

})

External Template:- Use this template when you code will be more than 5.

@Component({

selector: 'favorite',

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

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

})

**When we run angular application, this external will not be present in Network. So when we run the application all typescript file will be translate to JavaScript and it placed inside main.bundle.js**

### Styles

In Angular we have three ways to apply styles:-

1. First way is using styleUrls properties in the component Meta data. So in this array we can have one or more css files

@Component({

selector: 'favorite',

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

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

})

1. Second way is using styles properties which are inline properties inside component Meta data.

We can use this way for short style change.

@Component({

selector: 'favorite',

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

styles: [

`

`

]

})

1. Third way using <style> </style> inside the external template of this component.

**And also we can use both styleUrls and styles properties together in the same component Meta data but the styleUrls value will always take.**

### View Encapsulation

In styles which we define inside a component will be present in that particular component only, they will not leak to outside environment. For that please look the concept of Shadow DOM in extra section

Don’t even bother about View Encapsulation.

### ngContent

Imagine you want to create a bootstrap panel component, so we create a panel component but as a best practice if you are building reusable component always prefix here **bootstrap-panel**

|  |
| --- |
| @Component({  selector: 'bootstrap-panel',  templateUrl: './panel.component.html',  styleUrls: ['./panel.component.css']  }) |

In Template (panel.component.html) instead of hard coding the content as highlighted I want to inject text or markup into this component from app-component.

|  |
| --- |
| <div class="panel panel-default">  <div class="panel panel-heading">Heading</div>  <div class="panel-body">Body</div>  </div> |

So for that one way is to use property binding, but this syntax is little weird here

|  |
| --- |
| <bootstrap-panel [body]="body"></bootstrap-panel> |

**So instead of using property binding with input properties, we are going to use ngContent element for that:-**

**Step 1:-**

In **panel.component.html** add the element **<ng-content></ng-content>,** this is custom element define in angular and add **the identifier as select =” you can refer here class, id or element”**

**It means if the consumer of the panel component have an element matches the selector, an element with heading class that element is going to replace right with below line.**

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

**Full code of panel.component.html**

<div class="panel panel-default">

<div class="panel panel-heading">

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

</div>

<div class="panel-body">

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

</div>

</div>

**Step 2:-**

In consumer component which is app component add two div with class **heading and body**, inside the div add the custom content or html markup

|  |
| --- |
| <bootstrap-panel>  <div class="heading">Heading</div>  <div class="body">  <h2>Body</h2>  <p>Some content...</p>  </div>  </bootstrap-panel> |

You don’t need a selector if you have **only one** ng-content.

### ngContainer

In above example

<div class="panel panel-heading">

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

</div>

Here highlighted ng-content is replace with

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

But I don’t need the complete div instead I need only Heading, **for that we can use ng-container which is another custom element in angular at runtime angular will take only the content of ng-container**

**In app.component.html (which is the consumer component)**

|  |
| --- |
| <bootstrap-panel>  <ng-container class="heading">Heading</ng-container>  <div class="body">  <h2>Body</h2>  <p>Some content...</p>  </div>  </bootstrap-panel> |

**So if we want to render something with putting inside a <div> or another kind of html element, use <ng-container> element in angular**

### Assignment

**In like.component.html**

|  |
| --- |
| <span  class="glyphicon glyphicon-heart"  [class.highlighted] ="isActive"  (click)="onClick()">  </span>  <span>{{ likesCount }}</span> |

**In like.component.css**

|  |
| --- |
| .glyphicon {  color: #ccc;  cursor: pointer;  }  .highlighted {  color: deeppink;  } |

**In like.component.ts**

|  |
| --- |
| import { Component, Input } from '@angular/core';  @Component({  selector: 'like',  templateUrl: './like.component.html',  styleUrls: ['./like.component.css']  })  export class LikeComponent{  @Input('likesCount') likesCount:number;  @Input('isActive') isActive:boolean;  onClick() {  this.likesCount += (this.isActive) ? - 1 : 1;  this.isActive = !this.isActive;  }  } |

**In app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent {  tweet = {  body: '....',  likesCount: 10,  isLiked: true  }  } |

**In app.component.html**

|  |
| --- |
| <like  [likesCount] = "tweet.likesCount"  [isActive] = "tweet.isLiked">  </like> |

# Directives

### Introduction

We use directives to modify the DOM. There are two types of directives:-

1. **Structural: Modify the structure of the DOM by adding or removing DOM elements.**
2. **Attribute: Modify the attributes of DOM elements. (They don’t add or remove elements. They only change the element they were placed on).**

|  |
| --- |
|  |

### ngIf

If we want to show or hide part of a page depending on some condition

1. Create an array of courses in app.component.ts

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent {  courses = [1,2];  } |

1. In **app.component.html** if you have any courses present render that in a list otherwise show some message. **So for that we are going to ngIf and prefix with \* (\*ngIf ) as it is change the structure of DOM (structural)**

**If condition evaluate to true particular element will be added to the DOM, otherwise it will be removed from the DOM.**

|  |
| --- |
| <div \*ngIf = "courses.length > 0">  List of courses  </div>  <div \*ngIf = "courses.length == 0">  No course yet  </div> |

**Second way**

**In Angular 4 instead of using two div, we can do it in one div using if/else and ng-template (using template variable #)**

|  |
| --- |
| <div \*ngIf = "courses.length > 0; else noCourses">  List of courses  </div>  <ng-template #noCourses>  No course yet  </ng-template> |

**Third way (this approach is good as it is more clear and consistent)**

**Using two separate ng-template**

|  |
| --- |
| <div \*ngIf = "courses.length > 0; then listOfCourses else noCourses"></div>  <ng-template #listOfCourses>  List of courses  </ng-template>  <ng-template #noCourses>  No course yet  </ng-template> |

### Hidden Property

Instead of using ngIf directive we can use the hidden attribute like we use in HTML. **So for that we can use the property binding to the hidden attribute like [hidden] = “courses.length ==0”**

|  |
| --- |
| <div [hidden] = "courses.length == 0">  List of courses  </div>  <div [hidden] = "courses.length > 0">  No course yet  </div> |

**But here the both the div will be present in DOM only one will be hidden, but using ngIf only one div be present in DOM whichever is returning true.**

|  |
| --- |
|  |

### ngSwitchCase

**If we want to select one tab instead of multiple tabs, we should go with ngSwitchCase**

**In app.component.html**

* **Here we need to use property binding to ngSwitch and bind to viewMode which we define app.componennt.ts class.**
* **And use \*ngSwitchCase in the underline to add or remove DOM element.**
* **And finally in <li> we need do class binding**

|  |
| --- |
| <ul class="nav nav-pills">  <li [class.active]="viewMode == 'map'"><a (click)="viewMode = 'map'">Map View</a></li>  <li [class.active]="viewMode == 'list'"><a (click)="viewMode = 'list'">List View</a></li>  </ul>  <div [ngSwitch] = "viewMode">  <div \*ngSwitchCase = "'map'">Map View Content</div>  <div \*ngSwitchCase = "'list'">List View Content</div>  <div \*ngSwitchDefault>Otherwises</div>  </div> |

**In app.component.ts**

Here we need to define a field to keep track of current selected TAB, so declare a variable and assign ‘map’ then we can change to “’list’

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent {  viewMode = 'map';  } |

So if we want to compare the value of field or property against multiple values use **\*ngSwithCase** directive.

### ngFor

We use this directive to render list of object.

**In app.component.ts** declare array of course object

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent {  courses = [  {id: 1, name:'course1'},  {id: 2, name:'course2'},  {id: 3, name:'course3'},  ];  } |

**In app.component.ts** reder the values

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

Now I want to render the index with course name

|  |
| --- |
| <ul>  <li \*ngFor="let course of courses; index as i">  {{i}} - {{ course.name }}  </li>  </ul> |

Now let say we want to render in table and highlight all the even rows

|  |
| --- |
| <ul>  <li \*ngFor="let course of courses; even as isEven">  {{ course.name }} <span \*ngIf="isEven">(Even)</span>  </li>  </ul> |

### ngFor and Change Detection

Now let us see how the **ngFor** directive responds to the changes in component state.

For that Add a button in template, whose functionality will be on click new course will be added to the array course object

**In app.component.html**

|  |
| --- |
| <button (click)="onAdd()">Add</button>  <ul>  <li \*ngFor="let course of courses">  {{ course.name }}  </li>  </ul> |

**In app.component.ts**

Implement the onAdd() function to push data to the customer array

|  |
| --- |
| export class AppComponent {  courses = [  {id: 1, name:'course1'},  {id: 2, name:'course2'},  {id: 3, name:'course3'},  ];  onAdd() {  this.courses.push({id:4,name:'course4'});  }  } |

So Angular has this change detection mechanism, whenever we click a button, ajax request or timer function implements angular performs a change detection.

|  |
| --- |
|  |

Similarly we can add a button with each course to remove it, click called a onRemove() with parameter passed is course object.

**In app.component.html**

|  |
| --- |
| <button (click)="onAdd()">Add</button>  <ul>  <li \*ngFor="let course of courses">  {{ course.name }}  <button (click)="onRemove(course)">Remove</button>  </li>  </ul> |

**In app.component.ts**

Create the function onRemove(course) – find the index from the course array and remove that using (**splice**)

|  |
| --- |
| export class AppComponent {  courses = [  {id: 1, name:'course1'},  {id: 2, name:'course2'},  {id: 3, name:'course3'},  ];  onAdd() {  this.courses.push({id:4,name:'course4'});  }  onRemove(course){  let index = this.courses.indexOf(course);  this.courses.splice(index,1);  }  } |

### ngFor and Trackby

* + Let see an example, add a button to load the courses and display it.

**In app.component.html**

|  |
| --- |
| <button (click)="loadCourses()">Load Courses</button>  <ul>  <li \*ngFor="let course of courses">  {{ course.name }}  </li>  </ul> |

**In app.component.ts**

Here create the function **loadCourses()**. So when we call the function from template we are going to initialize the courses field.

|  |
| --- |
| export class AppComponent {  courses;  loadCourses() {  this.courses = [  {id: 1, name:'course1'},  {id: 2, name:'course2'},  {id: 3, name:'course3'},  ];  }  } |

**So every time when we click this button we are resetting the courses field to a new array.**

**And also everything we will we click list( <ul><li></li></ul> ) is adding in the DOM by Angular. But it will be costly if is complicated list as Angular need to reconstruct the enter DOM object.**

Let see how we can optimize this:-

* Angular by default track object by their identity**. So if we re-download the course with id=1, every time we re-download the course, that course will be in different object in memory even though the content of this object is going to be equal.**

|  |
| --- |
|  |

* So we are going to instruct Angular to use different mechanism to track objects, instead of tracking by their identity or reference in memory, **we are going to track them by their ID**.
* If we re-download the exact same course from the server and none of the properties are changed, **Angular we not rerun that DOM element**.

|  |
| --- |
|  |

So doing the change in code in **app.component.html** add the **trackBy: trackCourse** (**note that here we are not calling the method, simply adding name of the method as a reference)**

|  |
| --- |
| <button (click)="loadCourses()">Load Courses</button>  <ul>  <li \*ngFor="let course of courses; trackBy: trackCourse">  {{ course.name }}  </li>  </ul> |

Now go head an implement that method (trackCourse()) in app component

|  |
| --- |
| export class AppComponent {  courses;  loadCourses() {  this.courses = [  {id: 1, name:'course1'},  {id: 2, name:'course2'},  {id: 3, name:'course3'},  ];  }  trackCourse(index,course) {  return course ? course.id : undefined;  }  } |

**So here is the lesson if we are dealing with simple don’t worry about trackBy feature.**

### The Leading Asterisk

When we use the leading Asterisk with our structural directive like \*ngIf , \*ngFor , \*ngSwitchCase . Angular is going to rewrite that block using an ng-template.

### ngClass

When we write the code for glyphicon example , we have use multiple class binding in the same <div>

|  |
| --- |
| <span  class="glyphicon"  [class.glyphicon-star] = "isFavorite"  [class.glyphicon-star-empty] = "!isFavorite"  (click) = "onClick()"  ></span> |

While approach is perfectly working fine, there is also another way using **ngClass** directive (**which is an attribute directive**) use to modify the existing DOM element

|  |
| --- |
| <span  class="glyphicon"  [ngClass]="{  'glyphicon-star': isFavorite,  'glyphicon-star-empty': !isFavorite  }"  (click) = "onClick()"  ></span> |

### ngStyle

In a button we have use **multiple style binding** as shown below. And canSave value is declare in the app component class.

|  |
| --- |
| <button  [style.backgroundColor]="canSave ? 'blue' : 'gray'"  [style.color]="canSave ? 'white':'black'"  [style.fontWeight]="canSave ? 'bold' : 'normal'"  >Save</button> |

This above template is little bit noisy, so instead of that we can [ngStyle] directive and declare everything inside it, which is lead to clear your code.

|  |
| --- |
| <button  [ngStyle] = "{  'backgroundColor': canSave ? 'blue' : 'gray',  'color': canSave ? 'white':'black',  'fontWeight': canSave ? 'bold' : 'normal'  }"  >Save</button> |

### Safe Traversal Operator

In App component we have define a field called task, we some properties

|  |
| --- |
| export class AppComponent {  task = {  title: 'Review Applications',  assignee:{  name:'Chiranjit Saha'  }  }  } |

In template of app component we write the value of assignee name using string interpolation {{}}

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

And sometimes the value of name is null which will give error. So for this we have 2 solutions for this problem:-

1. One way using \*ngIf directive , if the value of assignee present.

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

1. Using Safe Traversal operator (?) :

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

**In app component class**

|  |
| --- |
| export class AppComponent {  task = {  title: 'Review Applications',  assignee:null  }  } |

### Creating Custom Directives

We use custom directive mostly to format some input properties. Like entered phone number you want to format.

Step 1

Use Angular CLI or Step by step also we can create directive. But for now we will use angular CLI.

**ng g d input-format**

In input-format.directive.ts

But the selector has [] bracket, which basically means any elements that has this attribute.

**If Angular find an element with attribute it is going to apply this directive on that element.**

|  |
| --- |
| import { Directive } from '@angular/core';  @Directive({  selector: '[appInputFormat]'  })  export class InputFormatDirective {  constructor() { }  } |

Step 2

In **app.module.ts** add the custom directive class name inside declarations**, but if we create using Angular CLI then automatically it will add**.

|  |
| --- |
| @NgModule({  declarations: [  AppComponent,  CoursesComponent,  CourseComponent,  AuthorsComponent,  SummaryPipe,  FavoriteComponent,  PanelComponent,  LikeComponent,  InputFormatDirective  ], |

Step 3

Need to import **HostListener** from @angular/core, **this decorator allows us to subscribe the event raised from DOM element, the DOM element hosting this directive or other word DOM element has this attribute**

|  |
| --- |
| import { Directive, HostListener } from '@angular/core';  @Directive({  selector: '[appInputFormat]'  })  export class InputFormatDirective {  constructor() { }  } |

Step 4

Decorate the method with HostListner decorator, and as an argument we need to supply name of DOM event.

**Here we are basically implementing focus and blur functionality using custom directive.**

|  |
| --- |
| import { Directive, HostListener } from '@angular/core';  @Directive({  selector: '[appInputFormat]'  })  export class InputFormatDirective {  @HostListener('focus') onFocus(){  console.log("on Focus");  }  @HostListener('blur') onBlur(){  console.log("on Blur");  }  constructor() { }  } |

Step5

Apply the selector/attribute ( [appInputFormat] ) to the HTML element

|  |
| --- |
| <input type="text" appInputFormat> |

Step 6

**Need reference from HOST element, so in our constructor of custom directive class we need to inject an element reference object.**

**ElementRef** - This is the service define in Angular that gives us access to the DOM object.

|  |
| --- |
| import { Directive, HostListener, ElementRef } from '@angular/core';  @Directive({  selector: '[appInputFormat]'  })  export class InputFormatDirective {    constructor(private el: ElementRef) { }  @HostListener('blur') onBlur(){  console.log("on Blur");  }  } |

Now as an example we want to convert input text box to lowercase using ElementRef

|  |
| --- |
| import { Directive, HostListener, ElementRef } from '@angular/core';  @Directive({  selector: '[appInputFormat]'  })  export class InputFormatDirective {    constructor(private el: ElementRef) { }  @HostListener('blur') onBlur(){  let value:string = this.el.nativeElement.value;  this.el.nativeElement.value = value.toLowerCase();  }  } |

**How to pass format from consumer template?**

**Step 1**

**Now it will be better if we pass the format from the template , using @input properties in custom directive**

**In app.component.html**

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

**In input-format.directive.ts**

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

@Directive({

selector: '[appInputFormat]'

})

export class InputFormatDirective {

@Input('format') format;

constructor(private el: ElementRef) { }

@HostListener('blur') onBlur(){

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

if(this.format=='lowercse')

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

else

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

}

}

**Step 2**

**In app.component.html**

**Use the selector itself from the template**

|  |
| --- |
| <input type="text" [appInputFormat]="'uppercase'"> |

**In input-format.directive.ts**

**Replace the Input alias with selector custom directive name**

**So Here is the lesson:-**

1. We can custom directive to have more control over behavior of DOM elements.
2. We can pass data to directive using input properties and if you have only one property we can use the selector of this directive as an alias of this Input properties.
3. And Finally we can use the HostListener to subscriber to the event raise from the host DOM object

### Assignment

**In zippy.component.html**

|  |
| --- |
| <div class="zippy">  <div  class="zippy-heading"  [class.expanded] = "isExpanded"  (click) = "toggle()"  >  {{title}}  <span class="glyphicon"  [ngClass] = "{  'glyphicon-chevron-up' : isExpanded,  'glyphicon-chevron-down' : !isExpanded  }"  ></span>  </div>  <div \*ngIf = "isExpanded" class="zippy-body">  <ng-content></ng-content>  </div>  </div> |

**In zippy.component.ts**

|  |
| --- |
| import { Component, OnInit, Input } from '@angular/core';  @Component({  selector: 'zippy',  templateUrl: './zippy.component.html',  styleUrls: ['./zippy.component.css']  })  export class ZippyComponent {  @Input('title') title: string;  isExpanded : boolean;  toggle() {  this.isExpanded = !this.isExpanded;  }  } |

**In zippy.component.css**

|  |
| --- |
| import { Component, OnInit, Input } from '@angular/core';  @Component({  selector: 'zippy',  templateUrl: './zippy.component.html',  styleUrls: ['./zippy.component.css']  })  export class ZippyComponent {  @Input('title') title: string;  isExpanded : boolean;  toggle() {  this.isExpanded = !this.isExpanded;  }  } |

**In app.component.html**

|  |
| --- |
| <zippy title="Shipping details">  Shipping Details Content  </zippy> |

# Template-driven Forms

### Building a Bootstrap Form

Here we will create a bootstrap form below code:-

1. Input text box is decorate with **form-control** class which is new in bootstrap, that gives our input field nice modern look and feel
2. Label for=”**name is same as id name of input textbox**” that means when user clicks on the label, particular input textbox of the id name will be focused.
3. And all the label and input field should place inside the div with class **form-group**

|  |
| --- |
| <form action="">  <div class="form-group">  <label for="firstName"></label>  <input id="firstName" type="text" class="form-control">  </div>  </form> |

Complete form with textbox (firstName) , textarea (comment) and a submit button

|  |
| --- |
| <form action="">  <div class="form-group">  <label for="firstName">First Name</label>  <input id="firstName" type="text" class="form-control">  </div>  <div class="form-group">  <label for="comment">Comment</label>  <textarea id="comment" cols="30" rows="10" class="form-control"></textarea>  </div>  <button class="btn btn-primary">Submit</button>  </form> |

### Types of Forms

**FormControl**

In Angular we have class called **FormControl** for each input field in our form; we need to create the instance of control class. With this control we check:-

1. Current value in input field (**value**)
2. We can see (**touched**)
3. We can see (**untouched**)
4. Value is changed (**dirty**)
5. Value is not changed (**pristine**)
6. Valid or not (**valid**)
7. If not valid what validation error (**errors**)

|  |
| --- |
|  |

So in each input field of the form we need a control object.

**FormGroup**

Similar to control class (FormClass) we have another class called form group, which represents the group of control in a form.

Each form is essentially a control group, because it contains at least one control. And all the properties of the control class are also available in the form group class as well.

1. Current value in input field (**value**)
2. We can see (**touched**)
3. We can see (**untouched**)
4. Value is changed (**dirty**)
5. Value is not changed (**pristine**)
6. Valid or not (**valid**)
7. If not valid what validation error (**errors**)

**So we can ask a FormGroup if it’s valid or not, and that returns true if all the controls in that group are valid.**

|  |
| --- |
|  |

**Creating Controls**

There are two ways to create this control objects:-

|  |
| --- |
|  |

|  |
| --- |
|  |

### ngModel

Now we are going to add validation to the form using **Template-driven** approach. **In Template-drive approach we apply directive to input fields and angular will create object associated with that input field.**

Using **ngModel** we will do that and **if using ngModel we need name attribute to that input field.**

**So when we ngModel directive along with name attribute in an input field, Angular automatically create instance of the form control class and associate it with this input field.**

|  |
| --- |
| <form action="">  <div class="form-group">  <label for="firstName">First Name</label>  <input ngModel name="firstName" id="firstName" type="text" class="form-control">  </div>  <div class="form-group">  <label for="comment">Comment</label>  <textarea ngModel name="comment" id="comment" cols="30" rows="10" class="form-control"></textarea>  </div>  <button class="btn btn-primary">Submit</button>  </form> |

### Accessing the form state

On click on submit button we need to get the form data, so for that in template using the ngSubmit directive in the form to call the onSubmit() function in the component

In template

|  |
| --- |
|  |

In component

|  |
| --- |
|  |

### Accessing the form with @ViewChild

In template no need the pass the local refernce #f in the onSubmit() , instead we will get the in component using the viewchild()

In template

|  |
| --- |
|  |

In Component

|  |
| --- |
|  |

We using this approach when we need to access the form just at the point we submitted, but also earlier.

### Adding Validation

Now make the firstName field as required for this:-

1. Add **required** attribute which we used in HTML5.
2. Create a div from error message just below the input field and **also check variable of ngModel is not valid using \*ngIf directive** but this will give error before focused on the textbox. For that also need to add **touched** properties. Eg:- **\*ngIf =”firstName.touched && !firstName.valid”**

|  |
| --- |
| <form action="">  <div class="form-group">  <label for="firstName">First Name</label>  <input required ngModel name="firstName" #firstName="ngModel" id="firstName" type="text" class="form-control">  <div class="alert alert-danger" \*ngIf="firstName.touched && !firstName.valid">First Name is required</div>  </div>  <div class="form-group">  <label for="comment">Comment</label>  <textarea ngModel name="comment" id="comment" cols="30" rows="10" class="form-control"></textarea>  </div>  <button class="btn btn-primary">Submit</button>  </form> |

### Specific Validation Errors

In Angular we have few built in validator that are based on HTML5 validation attributes like minlength , maxlength, pattern etc.

**So when we multiple HTML5 validation attributes we cannot handle error message in one div,** need multiple div and need **errors** properties of form class like

**errors.firstName(#templateVaraibleNameOfNgModel).required**

**errors.firstName(#templateVaraibleNameOfNgModel).maxlength**

|  |
| --- |
| <form action="">  <div class="form-group">  <label for="firstName">First Name</label>  <input required minlength="3" maxlength="10" pattern="Chiranjit" ngModel name="firstName" #firstName="ngModel" id="firstName" type="text" class="form-control">  <div class="alert alert-danger" \*ngIf="firstName.touched && !firstName.valid">  <div \*ngIf="firstName.errors.required">First Name is required</div>  <div \*ngIf="firstName.errors.minlength">First Name should be more than 3 character</div>  <div \*ngIf="firstName.errors.pattern">pattern does not match </div>  </div>  </div>  <button class="btn btn-primary">Submit</button>  </form> |

Here for the minlength and maxlength error, no of characters in error message we can retrieve it dynamically by **requiredLength** using string interpolation.

Eg:- **firstName.errors.minlength.requiredLength**

|  |
| --- |
| <div \*ngIf="firstName.errors.minlength">First Name should be more than {{firstName.errors.minlength.requiredLength}} character</div> |

### Styling Invalid Input fields

As a best practice, we should highlight the invalid input fields, this improve the usability of our form.

So how to apply red border to this input fields. In style.css add the style of border red :-

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

Border: 2px solid red;

}

**ng-touched.ng-invalid** is dynamically add to our input field of textbox by Angular at runtime.

**In style.css** (which is parent css file for all component)

|  |
| --- |
| .form-control.ng-touched.ng-invalid{  border: 2px solid red;  } |

### ngForm

So we have learn when we use ngModel directive on an input field, Angular creates a from control object under the hood and associate that with this input fields.

**Coming to ngForm whenever we have <form></form>, Angular by default creates a directive called ngForm. Now create a template variable to get a reference of ngForm directive.**

**This ngForm directive has a output properties called ngSubmit and we use output properties to raise custom events**. So ngSubmit can be used in event binding expression eg:- (ngSubmit) = “Submit(f)”

|  |
| --- |
| <form #f="ngForm" (ngSubmit)="Submit(f)">  <div class="form-group">  <label for="firstName">First Name</label>  <input  required  minlength="3"  maxlength="10"  pattern="Chiranjit"  ngModel  name="firstName"  #firstName="ngModel"  id="firstName"  type="text"  class="form-control">  <div  class="alert alert-danger"  \*ngIf="firstName.touched && !firstName.valid">  <div \*ngIf="firstName.errors.required">  First Name is required  </div>  <div \*ngIf="firstName.errors.minlength">  First Name should be more than {{firstName.errors.minlength.requiredLength}} character  </div>  <div \*ngIf="firstName.errors.pattern">  pattern does not match  </div>  </div>  </div>  <button class="btn btn-primary">Submit</button>  </form> |

### ngModelGroup

Sometimes we need work with complex form, we can have multiple groups in a form, so for that just like ngModel we also have **ngModelGroup** directive.

In real time application we need to Group firstName, lastName to one group called contact, and so other group for address, billing etc.… So for that we need to group them separately using **ngModelGroup** directive.

|  |
| --- |
|  |

And also similar to ngModel we can give reference to this directive using template variable. This is useful if we want to validate the entire group as a whole. As shown below..

<div \*ngIf="!contact.valid">... Error</div>

So in our application, API may except some nested complex object like this.

|  |
| --- |
| <form #f="ngForm" (ngSubmit)="Submit(f)">  <div ngModelGroup="contact" #contact="ngModelGroup">  <div \*ngIf="!contact.valid">... Error</div>  <div class="form-group">  <label for="firstName">First Name</label>  <input  required  minlength="3"  maxlength="10"  pattern="Chiranjit"  ngModel  name="firstName"  #firstName="ngModel"  (change) = "log(firstName)"  id="firstName"  type="text"  class="form-control">  <div  class="alert alert-danger"  \*ngIf="firstName.touched && !firstName.valid">  <div \*ngIf="firstName.errors.required">  First Name is required  </div>  <div \*ngIf="firstName.errors.minlength">  First Name should be more than {{firstName.errors.minlength.requiredLength}} character  </div>  <div \*ngIf="firstName.errors.pattern">  pattern does not match  </div>  </div>  </div>  </div>  <div class="form-group">  <label for="comment">Comment</label>  <textarea ngModel name="comment" id="comment" cols="30" rows="10" class="form-control"></textarea>  </div>  <button class="btn btn-primary">Submit</button>  </form> |
|  |

### Control Classes and Directives

|  |
| --- |
|  |

We may think what the difference between ngForm is and ngModelGroup is **ngForm directive exposes an output properties called ngSubmit and we use this to handle an submit event**

### Disabling the submit button

In the above the form, there is problem that I can click this submit button without providing any values to the input fields.

**Ideally we have to disable this submit button and enable when the form is in valid state.**

For that we can use the template variable of ngForm directive and bind with [disable] property in the button by checking the valid properties.

|  |
| --- |
| <form #f="ngForm" (ngSubmit)="Submit(f)">  <button class="btn btn-primary" [disabled] = "!f.valid">Submit</button>  </form> |

### Working with check boxes

And to get the value of form and formatted as json can be done as below:-

<p>{{ f.value | json }}</p>

Checkbox you can check below as highlighted and same as textbox we need to add ngModel directive.

And in checkbox of label we don’t need for attribute.

|  |
| --- |
| <form #f="ngForm" (ngSubmit)="Submit(f)">  <div ngModelGroup="contact" #contact="ngModelGroup">  <div \*ngIf="!contact.valid">... Error</div>  <div class="form-group">  <label for="firstName">First Name</label>  <input  required  minlength="3"  maxlength="10"  pattern="Chiranjit"  ngModel  name="firstName"  #firstName="ngModel"  (change) = "log(firstName)"  id="firstName"  type="text"  class="form-control">  <div  class="alert alert-danger"  \*ngIf="firstName.touched && !firstName.valid">  <div \*ngIf="firstName.errors.required">  First Name is required  </div>  <div \*ngIf="firstName.errors.minlength">  First Name should be more than {{firstName.errors.minlength.requiredLength}} character  </div>  <div \*ngIf="firstName.errors.pattern">  pattern does not match  </div>  </div>  </div>  </div>  <div class="form-group">  <label for="comment">Comment</label>  <textarea ngModel name="comment" id="comment" cols="30" rows="10" class="form-control"></textarea>  </div>  <div class="checkbox">  <label for="">  <input type="checkbox" ngModel name="isSubscribed"> Subscribe to mailing list  </label>  </div>  <p>{{ f.value | json }}</p>  <button class="btn btn-primary" [disabled] = "!f.valid">Submit</button>  </form> |

### Working with Drop-down Lists

<div class="form-group">

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

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

<option value=""></option>

<option value=""></option>

<option value=""></option>

<option value=""></option>

</select>

</div>

**However in most real world applications we don’t hard code this <option> value here. Most of the times we call an API on the server to get the list of contact method. And populate the dropdown list dynamically.**

Now from the server get the value, as of now hardcode the value in an array think that it retrieve from server.

**In component class**

|  |
| --- |
| export class ContactFormComponent {  log(firstName) {  console.log(firstName);  }  Submit(f) {  console.log(f);  }  contactMethod = [  {id:1, name: 'Email'},  {id:2, name: 'Phone'}  ];  } |

**Now in template using \*ngFor directive retrieve the value and bind the [value] property id. And use string interpolation {{method.name}}**

|  |
| --- |
| <div class="form-group">  <label for="contactMethod">contact Method</label>  <select ngModel name="contactMethod" id="contactMethod" class="form-control">  <option \*ngFor="let method of contactMethod" [value] = "method.id"> {{method.name}}</option>  </select>  </div> |

|  |
| --- |
|  |

In most application we are going to send only this value to the server like 1 **but sometimes we may want to set the value of this contactMethod property to the actual method object that is object with two properties “id” and “name”.**

So instead of bind with [value] properties we need to bind with [**ngValue],** ngValue is an attribute directive that exposes ngValue property and we bind this with complex object like array etc.

**But most application we don’t need to do this but if we really want to set the value of a property to complex object use ngValue properties instead of standard value property.**

|  |
| --- |
| <option \*ngFor="let method of contactMethod" [ngValue]="method">{{method.name}}</option> |

**Now sometimes we need to select multiple values from dropdown list, so for that use multiple attribute**

|  |
| --- |
| <div class="form-group">  <label for="contactMethod">contact Method</label>  <select multiple ngModel name="contactMethod" id="contactMethod" class="form-control">  <option \*ngFor="let method of contactMethod" [value]="method.id">{{method.name}}</option>  </select>  </div> |

### Working with Radio Buttons

|  |
| --- |
| <div class="form-group">  <label for="contactMethod">contact Method</label>  <select multiple ngModel name="contactMethod" id="contactMethod" class="form-control">  <option \*ngFor="let method of contactMethod" [value]="method.id">{{method.name}}</option>  </select>  </div>  <div class="radio">  <label>  <input ngModel type="radio" name="contactMethod" value="1">  Email  </label>  </div>  <div class="radio">  <label>  <input ngModel type="radio" name="contactMethod" value="2">  Phone  </label>  </div> |
|  |

Here above we have hard coded the radio button value, but if I want to render dynamically using \*ngFor and bind with [value] property

|  |
| --- |
| <div class="form-group">  <label for="contactMethod">contact Method</label>  <select multiple ngModel name="contactMethod" id="contactMethod" class="form-control">  <option \*ngFor="let method of contactMethod" [value]="method.id">{{method.name}}</option>  </select>  </div>  <div \*ngFor="let method of contactMethod" class="radio">  <label>  <input ngModel type="radio" name="contactMethod" [value]="method.id">  {{method.name}}  </label>  </div> |

### Set Default values with ngModel Property Binding

In ngModel directive we can use the property binding[] to get the default value from the component.

**In template**

|  |
| --- |
|  |

**In component**

|  |
| --- |
|  |

### Using ngModel with Two-Way Binding

Sometime we need to check something or simply repeat whatever the user enters. We can do this with two-way binding.

**In template**

Here in the example what the user in the textarea will be shown below in <p></p>

|  |
| --- |
|  |

**In Component**

|  |
| --- |
|  |

### Setting and Patching Form Values

SetValue :- is used to set the whole form.

patchValue :- to override the parts of the form.

|  |
| --- |
|  |

### Using Form Data

**In Template**

You want to show the all the input values entered in the form at the end of the form.

Here submitted value is set at the component when the button is click.

|  |
| --- |
|  |

**In component**

|  |
| --- |
|  |

|  |
| --- |
|  |

### Resetting the forms

It will not only empty the inputs and also it will reset the states like (valid, touched etc..)

|  |
| --- |
|  |

# Reactive Forms

If we want to build dynamic form where input fields are render based on data structure that we get from the server. Then create the control object in the code.

### Benefits

* + More control over form structure.
  + More control over form behavior.
  + Easier to Unit Test.

**In Reactive Forms we no need to import FormsModule, instead need to import ReactiveFormsModule.**

|  |
| --- |
|  |

### Creates Controls Programmatically

We have simple bootstrap form, now we will convert this to Angular Form using the **reactive approach.**

|  |
| --- |
| <form>  <div class="form-group">  <label for="username">Username</label>  <input  id="username"  type="text"  class="form-control">  </div>  <div class="form-group">  <label for="password">Password</label>  <input  id="password"  type="text"  class="form-control">  </div>  <button class="btn btn-primary" type="submit">Sign Up</button>  </form> |

**Now when building reactive form we should create this form control object explicitly in the code.**

Step 1:-

In signup-form component import the below class from ‘@angular/forms’

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

Step 2:-

In Angular **AbstractControl** is base class for **FormControl** and **FormGroup**, all the properties that are common in both the class are actually define in AbstractControl class.

|  |
| --- |
|  |

Create a variable form and set it new instance of FormGroup object and as first argument we need to pass an object and that object should have one or more key value pair , **key should be string and values should be AbstractControl (FormControl(), FormGroup())object**.

And here username is valid property in Javascript, so no need to put in single quote.

Otherwise if we some special character like user-name and it is not valid property, so need to put in single quote.

|  |
| --- |
| 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()  });  } |

Step 3 :-

In template we need to associate the input fields with the form control object created in component at step 2.

1. Use a directive of form element called **[formGroup]** in <form> associate with actual form group object that we create in our component. (**this tells to use my form which is created in te component**)
2. Use a another directive called **formControlName** in input fields and associate with name of the key in form group object created in our component **(username, password**)

|  |
| --- |
| <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> |

Step 4:-

In main app module import the **ReativeFormsModule.** All the directives for building reactive forms are define in a separate module called **ReativeFormsModule** that need to explicitly need to import into our main module

In app.module.ts

|  |
| --- |
| imports: [  BrowserModule,  AppRoutingModule,  FormsModule,  ReactiveFormsModule  ], |

And automatically it will imports at top in **@’angular/forms’**

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

### Submitting the Form

Here the difference in reactive form we don’t get the form via the local reference, that will not work here because we are not using the angular auto creation mechanism.

**In templates**

|  |
| --- |
|  |

**In Component**

|  |
| --- |
|  |

### Adding Validation

In template-driven form we use HTML5 validation attribute (required, maxlength etc). But while building Reactive Forms we don’t use HTML5 attribute, **we assign validators when creating form control object**

**Step 1:-**

We assign one or more validators when creating form control objects. Now in the constructor of FormControl(‘’, **Validatos** class

|  |
| --- |
|  |

This highlighted Validators class is define under **@’angular/forms’** which we need to import as shownbelow

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

**All the validators in our template driven form is exist here as well. This validators are method defined in Validators class**

|  |
| --- |
| export class SignupFormComponent {  form = new FormGroup({  username: new FormControl('',Validators.required),  password: new FormControl('',Validators.required)  });  } |

**Step 2:**

In Template add the validation error message inside a div and we want to render this div only if this field is touched and invalid using \*ngIf

**In template driven form we use using #template variable binding with ngModel but in reactive form this approach does not work because here we have not applied ngModel directive, so instead we need to get access to the underline form control object that we created in our component using the method called get “form.get(‘username’)”**

|  |
| --- |
| <form [formGroup]="form">  <div class="form-group">  <label for="username">Username</label>  <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>  </div>  <button class="btn btn-primary" type="submit">Sign Up</button>  </form> |

Above highlighted code is little bit lengthy. Now to clean up this noisy code in component class **we can define a property that gives us to this username form control object**

|  |
| --- |
| export class SignupFormComponent {  form = new FormGroup({  username: new FormControl('',Validators.required),  password: new FormControl('',Validators.required)  });  get username() {  return this.form.get('usename');  }  } |

Now in template

|  |
| --- |
| <form [formGroup]="form">  <div class="form-group">  <label for="username">Username</label>  <input  formControlName="username"  id="username"  type="text"  class="form-control">  <div \*ngIf="username.touched && username.invalid" class="alert alert-danger">UserName is required</div>  </div>  <button class="btn btn-primary" type="submit">Sign Up</button>  </form> |

**Summary:-**

To add validation to reactive form:-

1. We assign one or more validators function when creating form control object.
2. In template use a div to render validation error messages exactly like template driven form.

### Specific Validation Errors

Now let us see how to add multiple validators as we earlier the second parameter of the constructor of FormControl class required either a validator function or an array of validator function.

|  |
| --- |
| export class SignupFormComponent {  form = new FormGroup({  username: new FormControl('',[  Validators.required,  Validators.minLength(3)  ]),  password: new FormControl('',Validators.required)  });  get username() {  return this.form.get('username');  }  } |

**Now in template display specific error messages**

|  |
| --- |
| <form [formGroup]="form">  <div class="form-group">  <label for="username">Username</label>  <input  formControlName="username"  id="username"  type="text"  class="form-control">  <div \*ngIf="username.touched && username.invalid" class="alert alert-danger">  <div \*ngIf="username.errors.required">UserName is required</div>  <div \*ngIf="username.errors.minlength">UserName should be minimum {{username.errors.minlength.requiredLength}} characters</div>  </div>  </div>  <button class="btn btn-primary" type="submit">Sign Up</button>  </form> |

### Implementing Custom Validation

|  |
| --- |
|  |

Any function matches the above signature is consider a validator function; signature means type and order of parameter as well as return type. **We can define that anywhere in application but it is better to put all the validator function inside a class this way we encapsulate in a single place.**

**Step 1:-**

Add a new typescript file and create a class to define the custom validation using method **where the parameter should be AbstractControl, and this method return either ValidationErrors or null ( should be same as above interface)**

**And this ValidatorErrors represent an object that has one or more key value pairs, key are string and values can be anything**

|  |
| --- |
|  |

In order to access this method from outside without any creating the instance of the particular class, **we decorator the method with static**

|  |
| --- |
| import { AbstractControl, ValidationErrors } from '@angular/forms';  export class UserNameValidators {  static cannotContainSpace(control:AbstractControl) : ValidationErrors | null {  if((control.value as string).indexOf(' ') >=0)  return { cannotContainSpace: true };    return null;  }  } |

**Step 2:**

Now we need to import the custom validator as created in step 1 into our component.

|  |
| --- |
| import { UserNameValidators } from './username.validator';  import { Component } from '@angular/core';  import { FormGroup, FormControl, Validators } 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('',[  Validators.required,  Validators.minLength(3),  UserNameValidators.cannotContainSpace  ]),  password: new FormControl('',Validators.required)  });  get username() {  return this.form.get('username');  }  } |

**Note**

We put the custom validator typescript class inside signup-form folder because this is the only place we are using this validator but in real time application we need to use this custom validator in different component, so need to put in common place where it can be use by any component.

**Step 3**

In template use the div for custom error.

|  |
| --- |
| <form [formGroup]="form">  <div class="form-group">  <label for="username">Username</label>  <input  formControlName="username"  id="username"  type="text"  class="form-control">  <div \*ngIf="username.touched && username.invalid" class="alert alert-danger">  <div \*ngIf="username.errors.required">UserName is required</div>  <div \*ngIf="username.errors.minlength">UserName should be minimum {{username.errors.minlength.requiredLength}} characters</div>  <div \*ngIf="username.errors.cannotContainSpace">UserName cannot contain space</div>  </div>  </div>  <button class="btn btn-primary" type="submit">Sign Up</button>  </form> |

### Asynchronous Operation

Sometime we need to call server to validate a given value, Eg: - to check the UserName input is taken or not.

Let go ahead and implement custom validator.

**Step 1:-**

In **username.validator.ts** add another static method (here we not calling any server) – Just check some name (hardcode name).

|  |
| --- |
| static shouldBeUnique(control:AbstractControl): ValidationErrors | null {  if(control.value == 'chiranjit')  return { shouldBeUnique: true };  return null;  } |

But what should we do to simulate to call to the server. Calling the server is what we classified as

**Asynchronous Operation**: - When we called a server there is going to a bit of delay it might be ½ , 2, 30 secs .. Depending on the connection string. In situation like that the process that is executing our code doesn’t wants to block while waiting for the result that coming back from the server, if the process block the user cannot interact with the browser. So that process is going to call the server behind the scene and when the result is ready it going to display to user. **Asynchronous means non-blocking**

|  |
| --- |
|  |

Let see the below example of Asynchronous operation:-

setTimeout(() => {

console.log('ok');

},2000);

Let add this in static method shouldBeUnique

|  |
| --- |
| static shouldBeUnique(control:AbstractControl): ValidationErrors | null {  setTimeout(() => {  console.log('ok');  },2000);  return null;  } |

This is will always return null, so for that we need a function which returns different parameter.

### Asynchronous Validation

In FormControl we have a parameter asyncValidator which is also an optional parameter, it is either AsyncValidatorFn or array of AsyncValidatorFn[].

|  |
| --- |
|  |

An AsyncValidator is an interface with below declaration.

|  |
| --- |
|  |

**What are the classes Promise and Observable?**

In Javascript we use these classes to work with asynchronous operation. For now we will be use Promise<validationErrors|null>.

Now change the shouldBeUnique static return type to Promise<>. Below is the constructor of Promise Object.

|  |
| --- |
|  |

For executor- Just remember resolve that takes a value and return void.

For reject parameter – it takes an optional parameter called reason and return void.

|  |
| --- |
| static shouldBeUnique(control:AbstractControl): Promise<ValidationErrors | null> {  return new Promise((resolve,reject)=>{  setTimeout(() => {  if(control.value == 'chiranjit')  resolve({shouldBeUnique : true});  else  resolve(null);  },2000);  });  } |

Finally registered the static function in the consuming component of FormControl in third parameter.

|  |
| --- |
| export class SignupFormComponent {  form = 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('username');  }  } |

In Template add the validation div

|  |
| --- |
| <form [formGroup]="form">  <div class="form-group">  <label for="username">Username</label>  <input  formControlName="username"  id="username"  type="text"  class="form-control">  <div \*ngIf="username.touched && username.invalid" class="alert alert-danger">  <div \*ngIf="username.errors.required">UserName is required</div>  <div \*ngIf="username.errors.minlength">UserName should be minimum {{username.errors.minlength.requiredLength}} characters</div>  <div \*ngIf="username.errors.cannotContainSpace">UserName cannot contain space</div>  <div \*ngIf="username.errors.shouldBeUnique">UserName already taken</div>  </div>  </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> |

### Showing a Loader Image

When working on asynchronous validator we may need a loader image, while asynchronous is in progress.

In template add a div with some message or we can use glyphicon. So using **pending** property we can do that.

|  |
| --- |
| <form [formGroup]="form">  <div class="form-group">  <label for="username">Username</label>  <input  formControlName="username"  id="username"  type="text"  class="form-control">  <div \*ngIf="username.pending">Checking for uniqueness..</div>  <div \*ngIf="username.touched && username.invalid" class="alert alert-danger">  <div \*ngIf="username.errors.required">UserName is required</div>  <div \*ngIf="username.errors.minlength">UserName should be minimum {{username.errors.minlength.requiredLength}} characters</div>  <div \*ngIf="username.errors.cannotContainSpace">UserName cannot contain space</div>  <div \*ngIf="username.errors.shouldBeUnique">UserName already taken</div>  </div>  </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> |

### Validating the Form Input Upon Submit

While AsynchValidator validate are useful for validating the input field in place, sometimes we need to do validation upon submitting the form to the server.

For example image instead of signup form here, we are building a login form, in this case we need to submit both the username and password to the server.

**Step 1:**

In template add the event for ngSubmit and call a function where inside the server we will call the server.

|  |
| --- |
| <form [formGroup]="form" (ngSubmit)="login()">  ...  </form> |

**Step 2:**

Create the login() function, as of now we are not calling any server in future we will do that

|  |
| --- |
| login() {  this.form.setErrors({  invalidLogin:true;  });  } |

Here setErrors is prefined method present in abstractcontrol level which means it can applied to FormControl as well as FormGroup. Here we applied to FormGroup, if we applied to FormControl then it will be this.username.SetErrors.

Add login() it will always send errors, in future we will see the server call.

**Step 3:**

In template add a div to display validation error. We can also add multiple div for different kind of validation error.

|  |
| --- |
| <form [formGroup]="form" (ngSubmit)="login()">  <div \*ngIf="form.errors" class="alert alert-danger">  Username or Password is invalid.  </div>  <div class="form-group">  <label for="username">Username</label>  <input  formControlName="username"  id="username"  type="text"  class="form-control">  <div \*ngIf="username.pending">Checking for uniqueness..</div>  <div \*ngIf="username.touched && username.invalid" class="alert alert-danger">  <div \*ngIf="username.errors.required">UserName is required</div>  <div \*ngIf="username.errors.minlength">UserName should be minimum {{username.errors.minlength.requiredLength}} characters</div>  <div \*ngIf="username.errors.cannotContainSpace">UserName cannot contain space</div>  <div \*ngIf="username.errors.shouldBeUnique">UserName already taken</div>  </div>  </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> |

### Nested FormGroups

In the previous we have only one FormGroup, but in large complex application we can have a Form with multiple subgroups.

**Step 1: Create a new FormGroup (nested FormGroup)**

Now UserName and Password will be move to one more FormGroup called account and passing username properties pass with full path as highlighted below

|  |
| --- |
| import { UserNameValidators } from './username.validator';  import { Component } from '@angular/core';  import { FormGroup, FormControl, Validators } from '@angular/forms';  @Component({  selector: 'signup-form',  templateUrl: './signup-form.component.html',  styleUrls: ['./signup-form.component.css']  })  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)  })  });    login() {  this.form.setErrors({  invalidLogin:true  });  }    get username() {  return this.form.get('account.username');  }  } |

**Step 2:**

But the above will give error like username (username control is not found) because username is no longer a root control in template now it part of a group called account.

**In Angular just like formControlName we have another group called formGroupName. Now create another div (add the directive formGroupName and assign to account) and move the form-group inside the div.**

|  |
| --- |
| <form [formGroup]="form" (ngSubmit)="login()">  <div \*ngIf="form.errors" class="alert alert-danger">  Username or Password is invalid.  </div>  <div formGroupName="account">  <div class="form-group">  <label for="username">Username</label>  <input  formControlName="username"  id="username"  type="text"  class="form-control">  <div \*ngIf="username.pending">Checking for uniqueness..</div>  <div \*ngIf="username.touched && username.invalid" class="alert alert-danger">  <div \*ngIf="username.errors.required">UserName is required</div>  <div \*ngIf="username.errors.minlength">UserName should be minimum {{username.errors.minlength.requiredLength}} characters</div>  <div \*ngIf="username.errors.cannotContainSpace">UserName cannot contain space</div>  <div \*ngIf="username.errors.shouldBeUnique">UserName already taken</div>  </div>  </div>  <div class="form-group">  <label for="password">Password</label>  <input  formControlName="password"  id="password"  type="text"  class="form-control">  </div>  </div>  <button class="btn btn-primary" type="submit">Sign Up</button>  </form> |

### FormArray

Sometimes we need to work with the array of object in a form. It holds a array of controls

Eg: - want to enter multiple values in a textbox.

**Here instead of FormControl we use FormArray class**

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { FormGroup, FormArray, FormControl } from '@angular/forms';  @Component({  selector: 'new-course-form-component',  templateUrl: './new-course-form-component.component.html',  styleUrls: ['./new-course-form-component.component.css']  })  export class NewCourseFormComponentComponent {  form = new FormGroup({  topics: new FormArray([])  });  removeTopic(topic: FormControl){  let index = this.topics.controls.indexOf(topic);  this.topics.removeAt(index);  }  addTopic(topic:HTMLInputElement){  this.topics.push(new FormControl(topic.value));  topic.value = '';  }  get topics(){  return this.form.get('topics') as FormArray;  }  } |

In template

|  |
| --- |
| <form action="">  <input  type="text"  class="form-control"  (keyup.enter)="addTopic(topic)" #topic>  <ul class="list-group">  <li  \*ngFor="let topic of topics.controls"  (click) = "removeTopic(topic)"  class="list-group-item">  {{ topic.value }}  </li>  </ul>  </form> |

### FormBuilder

In the below class we have all the derivate of abstractControlClass

|  |
| --- |
| export class NewCourseFormComponentComponent {  form = new FormGroup({  name: new FormControl(),  contact: new FormGroup({  email: new FormControl(),  phone: new FormControl()  }),  topics: new FormArray([])  }); |

There is a cleaner to write the above, in Angular we have a class called FormBuilder that we use for building forms as highlighted below:

|  |
| --- |
| export class NewCourseFormComponentComponent {  form;    constructor(fb: FormBuilder){  this.form= fb.group({  name: ['',Validators.required],  contact: fb.group({  email: [],  phone:[]  }),  topics:fb.array([])  })  }  } |

# Consuming HTTP Services

### Introduction

* Perform CRUD Operations (Create, Read, Update and Delete).
* Extract reusable data service.
* Properly handle different kinds of error.
* Build application with proper separation of concern.

|  |
| --- |
|  |

### JSONPlaceHolder

**Jsonplaceholder.typicode.com** (we will be use this service as a back end of our application ) – Here we have list of JSON objects which we will be using for create http request from Angular project. And as we know this is fake service there is no database as back end it’s just a fake HTTP service.

|  |
| --- |
|  |

### Getting Data

Now let’s see how to write code to get all the posts object from JSONPlaceHolder and display them on a page.

**Step 1:-**

In app.module.ts need to import new module HttpModule

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

imports: [

BrowserModule,

AppRoutingModule,

FormsModule,

ReactiveFormsModule,

HttpModule

],

**We can use HttpClientModule also**

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

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

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

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

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

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

@NgModule({

declarations: [AppComponent],

imports: [BrowserModule, FormsModule, HttpClientModule],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule {}

Execute the statement “ **npm install @angular/http –save “** as httpModule is removed in Angular 7.

**All the classes for working with HTTP services are encapsulated in this Module.**

**Step 2:-**

Now in the consuming component create a constructor which passes a parameter of type HTTP and import the class HTTP at the top.

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { Http } from '@angular/http';  @Component({  selector: 'posts',  templateUrl: './posts.component.html',  styleUrls: ['./posts.component.css']  })  export class PostsComponent {  constructor(http: Http) {  http.get('http://jsonplaceholder.typicode.com/posts');  }  } |

We use this HTTP class to send Http Request to the backend. **Also we can use HttpClient**

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { HttpClient } from '@angular/common/http';  import { map } from 'rxjs/operators';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {  loadedPosts = [];  constructor(private http: HttpClient) {}  ngOnInit() {  this.fetchPosts();  }  onCreatePost(postData: { title: string; content: string }) {  // Send Http request  this.http  .post(  'http://localhost:62286/Api/TitleDescription/InsertTitleDescription',  postData  )  .subscribe(responseData => {  console.log(responseData);  });  }    } |

**But if we move HttpModule from app.module.ts, then the consuming component will give error as we inject Http class in the component, but we have not register this HttpModule as provider.So dependency injection in angular doesnot work unless we register the HttpModule in app.module.ts.**

**Step 3:- Now call the get method to get some data from the server.**

* **As an argument in GET method we need to set the URL**.
* **And the method return the Observable<Response>** as we know, promises and observable we use when we work with asynchronous operation (non-blocking operation).
* **Now this observable has a method called subscribe** with this we basically subscribing this observable. So when the result is ready we will be notifying.
* **And also we will convert the response object to Json. As we will Json object to display data to our view.**
* **And assign the json object to an array (posts). Now we can go to a template and render all this posts using a branch of list item <li>**

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { Http } from '@angular/http';  @Component({  selector: 'posts',  templateUrl: './posts.component.html',  styleUrls: ['./posts.component.css']  })  export class PostsComponent {  posts: any[];  constructor(http: Http) {  http.get('http://jsonplaceholder.typicode.com/posts')  .subscribe(response => {  this.posts = response.json();  });  }  } |

**Step 4:- Use the posts array and display in template**

|  |
| --- |
| <ul class="list-group">  <li  \*ngFor="let post of posts"  class="list-group-item">  {{post.title}}  </li>  </ul> |

### Creating Data

Now let see how to create a new post using POST method.

**Step 1:-** In template add an input property textbox and use **keyup.enter event** to call a method and pass the input field using **template variable.**

|  |
| --- |
| <input  (keyup.enter) = "createPost(title)" #title  type="text" class="form-control">  <ul class="list-group">  <li  \*ngFor="let post of posts"  class="list-group-item">  {{post.title}}  </li>  </ul> |

Step 2:- In component implement the new method createPost() as shown above.

* Inside the function call the POST method to insert data to the server.
* Now the POST method pass the parameters

url

body which will be in JSON format

|  |
| --- |
| import { Component, OnInit, Input } from '@angular/core';  import { Http } from '@angular/http';  @Component({  selector: 'posts',  templateUrl: './posts.component.html',  styleUrls: ['./posts.component.css']  })  export class PostsComponent {  posts: any[];  private url = 'http://jsonplaceholder.typicode.com/posts';  constructor(private http: Http) {  http.get(this.url)  .subscribe(response => {  this.posts = response.json();  });  }  // here always annotate the input to  // HTMLInputElement as we receive from browser  createPost(input:HTMLInputElement) {  // post is a JSON object to store the value  // which we receive from input field  let post:any = {title : input.value }  //clearing the input field  input.value = '';  // first parameter is URL  // second paramater is the post we want to push after  // converting to JSON to string using stringify  this.http.post(this.url, JSON.stringify(post))  .subscribe(response => {  // get the id which we inserted from response object  post.id = response.json().id;    // Finally push the id to Posts array to display  // in push get method to display at the end of the list  // or use splice to display at beginning of the list  this.posts.splice(0,0,post);  });  }  } |

All the method of Http class returns observable, which means we need to subscribe to the observable

### Updating Data

**Step1:-**

Create a button and call the updatepost method and pass the actual post method as the parameter.

|  |
| --- |
| <input  (keyup.enter) = "createPost(title)" #title  type="text" class="form-control">  <ul class="list-group">  <li  \*ngFor = "let post of posts"  class="list-group-item">  <button  (click) = "updatePost(post)"  >  Update</button>    {{post.title}}  </li>  </ul> |

Step2:-

Create the updatePost method in the component and the http.put or http.patch

**Difference between put and patch is**

* We use patch method to update few properties of an object.
* We use put when to pass the whole post method.

|  |
| --- |
| this.http.patch(this.url,JSON.stringify({isRead:true}))  //or  this.http.put(this.url,JSON.stringify(post)) |

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { Http } from '@angular/http';  @Component({  selector: 'posts',  templateUrl: './posts.component.html',  styleUrls: ['./posts.component.css']  })  export class PostsComponent implements OnInit {  posts: any[];  private url = 'https://jsonplaceholder.typicode.com/posts';  constructor(private http: Http) {  http.get(this.url)  .subscribe( response =>{  this.posts = response.json();  });  }  createPost(input: HTMLInputElement){  let post: any = {title: input.value};  input.value ='';  this.http.post(this.url,JSON.stringify(post))  .subscribe(response =>{  post.id = response.json().id;  this.posts.splice(0,0,post);    });  }  updatePost(post){  //Here first argument is the url  // second argument is the body of the request(here instead  //of sending whole post object we can send the particular  // property we want to modify)  //Here also we need append the id of the post in the url  this.http.patch(this.url + '/'+ post.id,JSON.stringify({isRead:true}))  .subscribe(response =>{  console.log(response.json());  })  }  ngOnInit() {  }  } |

Here in the Request Payload, you can see the isRead property which we pass using patch method. But if you used put then whole post method properties will be visible here.

|  |
| --- |
|  |

### Deleting Data

**Step1:-**

Create one more button which will call deletePost(post) method on click of the button.

|  |
| --- |
| <input  (keyup.enter) = "createPost(title)" #title  type="text" class="form-control">  <ul class="list-group">  <li  \*ngFor = "let post of posts"  class="list-group-item">  <button  (click) = "updatePost(post)"  >  Update</button>  <button  (click) = "deletePost(post)"  >  Delete</button>  {{post.title}}  </li>  </ul> |

**Step2:-**

Implement the deletePost(post) method where we will call delete http method to delete the particular post id.

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { Http } from '@angular/http';  @Component({  selector: 'posts',  templateUrl: './posts.component.html',  styleUrls: ['./posts.component.css']  })  export class PostsComponent implements OnInit {  posts: any[];  private url = 'https://jsonplaceholder.typicode.com/posts';  constructor(private http: Http) {  http.get(this.url)  .subscribe( response =>{  this.posts = response.json();  });  }  createPost(input: HTMLInputElement){  let post: any = {title: input.value};  input.value ='';  this.http.post(this.url,JSON.stringify(post))  .subscribe(response =>{  post.id = response.json().id;  this.posts.splice(0,0,post);    });  }  updatePost(post){  //Here first argument is the url  // second argument is the body of the request(here instead  //of sending whole post object we can send the particular  // property we want to modify)  //Here also we need append the id of the post in the url  this.http.patch(this.url + '/'+ post.id,JSON.stringify({isRead:true}))  .subscribe(response =>{  console.log(response.json());  })  }  deletePost(post){  //In delete we don't have body  // then we need to subscribe  this.http.delete(this.url +'/' +post.id)  .subscribe(response =>{  //Get the index of post method  let index = this.posts.indexOf(post);  //use the splice to delete using index and 1 object  this.posts.splice(index,1);  })  }  ngOnInit() {  }  } |

### OnInit Interface

As we seen in the above the get http method we called inside the constructor, but as we known constructor should be light and should not perform expenses operation like calling the server.

Component in Angular have life cycle hooks, there are some method that we can add to our component and Angular will automatically call these methods as specify time during the life cycle of the component

|  |
| --- |
|  |

Don’t call http service inside the constructor of the component and if you need initialization use ngOnInit method.

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { Http } from '@angular/http';  @Component({  selector: 'posts',  templateUrl: './posts.component.html',  styleUrls: ['./posts.component.css']  })  export class PostsComponent implements OnInit {  posts: any[];  private url = 'https://jsonplaceholder.typicode.com/posts';  constructor(private http: Http) {  }  ngOnInit() {  this.http.get(this.url)  .subscribe( response =>{  this.posts = response.json();  });  }  } |

### Separation of concerns

In the above component itself we have all the http service, which lead to the violation of separation principle. As we know our classes should have a single responsibility.

As a result unit testing is not possible, as in the automated test we never all the http service.

Solution is to create another class/service and this class will be responsible with the backend.

|  |
| --- |
|  |

### Extracting a service

**“mv post.service.\* services” – move the service files to services folder**

**Step1:-**

Create a service “ng g s posts”

**Step2:-**

Add the service inside the providers array of the app.module.

|  |
| --- |
| providers: [PostService], |

**Step3:-**

**Add the http method inside the service**

|  |
| --- |
| import { Http } from '@angular/http';  import { Injectable } from '@angular/core';  @Injectable({  providedIn: 'root'  })  export class PostService {  private url = 'https://jsonplaceholder.typicode.com/posts';  constructor(private http: Http) { }  getPosts(){  return this.http.get(this.url);  }  createPost(post){  return this.http.post(this.url,JSON.stringify(post));  }  updatePost(post){  return this.http.patch(this.url + '/'+ post.id,JSON.stringify({isRead:true}))  }  deletePost(id){  return this.http.delete(this.url +'/' + id)  }  } |

**Step 5:-**

**In component first inject the service inside the constructor and just call the particular method**

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { PostService } from '../services/post.service';  @Component({  selector: 'posts',  templateUrl: './posts.component.html',  styleUrls: ['./posts.component.css']  })  export class PostsComponent implements OnInit {  posts: any[];  constructor(private service:PostService) {  }  ngOnInit() {  this.service.getPosts()  .subscribe( response =>{  this.posts = response.json();  });  }  createPost(input: HTMLInputElement){  let post: any = {title: input.value};  input.value ='';  this.service.createPost(post)  .subscribe(response =>{  post.id = response.json().id;  this.posts.splice(0,0,post);    });  }  updatePost(post){  this.service.updatePost(post)  .subscribe(response =>{  console.log(response.json());  })  }  deletePost(post){  this.service.deletePost(post.id)  .subscribe(response =>{  let index = this.posts.indexOf(post);  this.posts.splice(index,1);  })  }  } |

### Handling Errors:

We have two types of http error:- unexpected and expected error.

|  |
| --- |
|  |

### Handling Unexpected Error

Subscriber method second parameter is used for handling error.

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { PostService } from '../services/post.service';  @Component({  selector: 'posts',  templateUrl: './posts.component.html',  styleUrls: ['./posts.component.css']  })  export class PostsComponent implements OnInit {  posts: any[];  constructor(private service:PostService) {  }  ngOnInit() {  this.service.getPosts()  //this subscribe has another optional parameter  //error as indicated by the ? mark  .subscribe( response =>{  this.posts = response.json();  },  error =>{  // here instead of alert we can toast notification  alert('An unexpected error occurs');  console.log(error);  });  }  createPost(input: HTMLInputElement){  let post: any = {title: input.value};  input.value ='';  this.service.createPost(post)  .subscribe(response =>{  post.id = response.json().id;  this.posts.splice(0,0,post);  },  error=>{  alert('An unexpected error occurs');  console.log(error);  });  }  updatePost(post){  this.service.updatePost(post)  .subscribe(response =>{  console.log(response.json());  },  error=>{  alert('An unexpected error occurs');  console.log(error);  });  }  deletePost(post){  this.service.deletePost(post.id)  .subscribe(response =>{  let index = this.posts.indexOf(post);  this.posts.splice(index,1);  },  error=>{  alert('An unexpected error occurs');  console.log(error);  });  }  } |

### Handling Expected error

In error handler we need to check the status of the response

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { PostService } from '../services/post.service';  @Component({  selector: 'posts',  templateUrl: './posts.component.html',  styleUrls: ['./posts.component.css']  })  export class PostsComponent implements OnInit {  posts: any[];  constructor(private service:PostService) {  }  ngOnInit() {  this.service.getPosts()  .subscribe(  response =>{  this.posts = response.json();  },  error =>{  alert('An unexpected error occurs');  console.log(error);  });  }  createPost(input: HTMLInputElement){  let post: any = {title: input.value};  input.value ='';  this.service.createPost(post)  .subscribe(  response =>{  post.id = response.json().id;  this.posts.splice(0,0,post);  },  (error:Response)=>{  if(error.status==400){  //Now we set the error inside a form  // this.form.setErrors(error.json())  }  else{  alert('An unexpected error occurs');  console.log(error);  }  }); }  updatePost(post){  this.service.updatePost(post)  .subscribe(  response =>{  console.log(response.json());  },  error=>{  alert('An unexpected error occurs');  console.log(error);  });  }  deletePost(post){  this.service.deletePost(post.id)  .subscribe(  response =>{  let index = this.posts.indexOf(post);  this.posts.splice(index,1);  },  //Here we need to pass anotate error  //with response class and put in ()  (error: Response) =>{  //Now we can get all the error which we  //want to define (expected error)  if(error.status == 404)  {  alert('This post has already been deleted');  }  else{  alert('An unexpected error occurs');  console.log(error);  }  });  }  } |

### Throwing Application Specific Errors

In the above implementation leads to vialotion of separation of concern as highlighted below.

|  |
| --- |
| deletePost(post){  this.service.deletePost(post.id)  .subscribe(  response =>{  let index = this.posts.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 occurs');  console.log(error);  } |

**Step1:**

So we will move this logic inside the service itself. So inside the service and we will use here catchError and throw inside the service.

|  |
| --- |
| import { AppError } from './../posts/app-error';  import { Http } from '@angular/http';  import { Injectable } from '@angular/core';  import { catchError, retry } from 'rxjs/operators';  import { Observable } from 'rxjs';  import { NotFoundError } from '../not-found-error';  deletePost(id){  return this.http.delete(this.url +'/' + id)  //Here we have to catch the error.  // then send the different kind of error  // object to the consumer of the service  // ANd also we have to return an observable that has error.  .pipe( catchError((error:Response)=>{  if(error.status==404)  return Observable.throw(new NotFoundError())  //this throw method returns a new  //observable that has an error  //type of the error should be something  //specific to our application not the response object  // Here pass the AppError object  return Observable.throw(new AppError(error));  }));  } |

**Step2:**

And we are creating a new class to represent application specific error.

|  |
| --- |
| export class AppError{  constructor(public originalError?: any){}  } |

**Step3:-**

Create a type script file called **non-found-error.ts** with class NotFoundError and **extends it fro AppError class.**

|  |
| --- |
| import { AppError } from './posts/app-error';  export class NotFoundError extends AppError{  } |

**Step4:-**

In the component we will pass the type as App Error

|  |
| --- |
| deletePost(post){  this.service.deletePost(post.id)  .subscribe(  response =>{  let index = this.posts.indexOf(post);  this.posts.splice(index,1);  },  //Here pass the AppError instance  (error: AppError) =>{  //Here instance of NotFoundError  if(error instanceof NotFoundError)  {  alert('This post has already been deleted');  }  else{  alert('An unexpected error occurs');  console.log(error);  }  });  } |

### Handling Bad Request

**Step1:-**

**Change the service to catch and throw the error.**

|  |
| --- |
| updatePost(post){  return this.http.patch(this.url + '/'+ post.id,JSON.stringify({isRead:true}))  .pipe(catchError((error:Response)=>{  if(error.status==400)  return Observable.throw(new BadRequestError(error.json))  return Observable.throw(new AppError(error.json()));  }));  } |

**Step2:-**

**Create a class file BadRequestError which extends from AppError.**

|  |
| --- |
| import { AppError } from './posts/app-error';  export class BadRequestError extends AppError{  } |

**Step3:**

**Change the component to get the Error.**

|  |
| --- |
| createPost(input: HTMLInputElement){  let post: any = {title: input.value};  input.value ='';  this.service.createPost(post)  .subscribe(  response =>{  post.id = response.json().id;  this.posts.splice(0,0,post);  },  (error:AppError)=>{  if(error instanceof BadRequestError){  //Now we set the error inside a form  //this.form.setErrors(error.originalError);  }  else{  alert('An unexpected error occurs');  console.log(error);  }  });  } |

### Importing Observable Operators and Factory Methods

While throw method we are using observable.throw() as throw() method is calling on the observable class not a observable object (this is a static method).

We call this method as Factory Methods as they create object.

Here import from observable instead of operator.

|  |
| --- |
| import 'rxjs/add/observable/throw'; |

### Global Error Handling

How to handle unexpected error Globally.

**Step1:-**

Create a new ts file in common folder (app-error-handler.ts)

Here in the ts file we will export a class that will responsible for handling all unexpected error in our application.

|  |
| --- |
| import { ErrorHandler } from '@angular/core'  export class AppErrorHander implements ErrorHandler{  //Here ErrorHandler is the predifined class  //handleError() method is part of it  handleError(error){  alert('An unexpected error occurs');  console.log(error);  }  } |

**Step2:--**

Register the above class as a dependency or provider in app-module.ts.

**We should not just import in the provider, we should also inform which ever class is using this errorHander.**

**With this object we are telling Angular were ever we are using error handler, instead use this new class.**

|  |
| --- |
| providers: [  PostService,  {provide:ErrorHandler,useClass:AppErrorHander}  ], |

**Step3:-**

**In the component where we are handling error cleanup all repeative code. And throw the error which we handle here like 400 and 404**

|  |
| --- |
| import { BadRequestError } from './../bad-request-error';  import { AppError } from './app-error';  import { Component, OnInit } from '@angular/core';  import { PostService } from '../services/post.service';  import { NotFoundError } from 'src/app/not-found-error';  @Component({  selector: 'posts',  templateUrl: './posts.component.html',  styleUrls: ['./posts.component.css']  })  export class PostsComponent implements OnInit {  posts: any[];  constructor(private service:PostService) {  }  ngOnInit() {  this.service.getPosts()  .subscribe(  response =>{  this.posts = response.json();  });  }  createPost(input: HTMLInputElement){  let post: any = {title: input.value};  input.value ='';  this.service.createPost(post)  .subscribe(  response =>{  post.id = response.json().id;  this.posts.splice(0,0,post);  },  (error:AppError)=>{  if(error instanceof BadRequestError){  }  else{  // We have to rethrow the error  //As the error handler handle here  // will never been throw to global error handler  throw error;  //alert('An unexpected error occurs');  //console.log(error);  }  });  }  updatePost(post){  this.service.updatePost(post)  .subscribe(  response =>{  console.log(response.json());  });  }  deletePost(post){  this.service.deletePost(post.id)  .subscribe(  response =>{  let index = this.posts.indexOf(post);  this.posts.splice(index,1);  },  (error: AppError) =>{  if(error instanceof NotFoundError)  {  alert('This post has already been deleted');  }  else throw error;  });  }  } |

### Extracting a Reusable Error Handling Method

In the post we used many repetitive codes for check the error difference is 400 or 404.

**So for that we will create a separate private method called HandleError instead of repetiting the error handling in every method. I am going to simply delegate the error handling with the new method.**

|  |
| --- |
| import { BadRequestError } from './../bad-request-error';  import { AppError } from './../posts/app-error';  import { Http } from '@angular/http';  import { Injectable } from '@angular/core';  import { catchError} from 'rxjs/operators';  import { Observable } from 'rxjs';  import { NotFoundError } from '../not-found-error';  @Injectable({  providedIn: 'root'  })  export class PostService {  private url = 'https://jsonplaceholder.typicode.com/posts';  constructor(private http: Http) { }  getPosts(){  return this.http.get(this.url)  .pipe(catchError(this.handleError));  }  createPost(post){  return this.http.post(this.url,JSON.stringify(post))  .pipe(catchError(this.handleError));  }  updatePost(post){  return this.http.patch(this.url + '/'+ post.id,JSON.stringify({isRead:true}))  .pipe(catchError(this.handleError));  }  //Previous code  /\*deletePost(id){  return this.http.delete(this.url +'/' + id)  .pipe( catchError((error:Response)=>{  if(error.status==404)  return Observable.throw(new NotFoundError())  return Observable.throw(new AppError(error));  }));  }\*/  deletePost(id){  return this.http.delete(this.url +'/' + id)  //here instead of passing error function  //we will pass reference to this new private method  // Note I am not calling the method I a simply passing reference  .pipe( catchError(this.handleError));  }  //this method takes an error which of type Response  private handleError(error:Response){  if(error.status==400)  return Observable.throw(new BadRequestError(error.json));  if(error.status==404)  return Observable.throw(new NotFoundError());    return Observable.throw(new AppError(error));  }  } |

### Extracting a Reusable Data Service

Now in real world application we will have different service like courseService, EmployeeService etc. So in every service we need to write the code for getAllCourse, CreateCouse, UpdateCourse, DeleteCourse and handleError(error:Response) as shown in above section.

**Instead we want to extract a resuable service for working with http endpoints. And inherit the resuable service to the other service.**

**Step1:-**

**In service folder create a typescript file data.service.ts and copy all the code from postService to the new file. In this new service there is nothing specify to any entity like post, customer, employee etc.**

|  |
| --- |
| import { BadRequestError } from './../bad-request-error';  import { AppError } from './../posts/app-error';  import { Http } from '@angular/http';  import { Injectable } from '@angular/core';  import { catchError} from 'rxjs/operators';  import { Observable } from 'rxjs';  import { NotFoundError } from '../not-found-error';  @Injectable({  providedIn: 'root'  })  export class DataService {  //Here we are pass the url and http so that we can get  //from derived class.  constructor(private url: string,private http: Http) { }  //Here all the method name should be generic.  getAll(){  return this.http.get(this.url)  .pipe(catchError(this.handleError))  }  Create(resource){  return this.http.post(this.url,JSON.stringify(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 Observable.throw(new BadRequestError(error.json));  if(error.status==404)  return Observable.throw(new NotFoundError());    return Observable.throw(new AppError(error));  }  } |

**Step2:-**

**Delete all the method from the post service as we define them in the data service. Now I want the service to inherit the data service class.**

|  |
| --- |
| import { DataService } from './data.service';  import { Http } from '@angular/http';  import { Injectable } from '@angular/core';  @Injectable({  providedIn: 'root'  })  export class PostService extends DataService{  constructor(http: Http) {  //In order to create the instance of derived class  // first we need to create instance of base class  //Here in the constructor we need to call  //constructor of base class  super('https://jsonplaceholder.typicode.com/posts',http);  }  } |

**This implementation makes the post service to three lines. So next time you create another service will also have less line of code as we will inherit from Base class(Data Service)**

**Step3:-**

**In the component change the method name which was defined the data service(parent class)**

|  |
| --- |
| import { BadRequestError } from './../bad-request-error';  import { AppError } from './app-error';  import { Component, OnInit } from '@angular/core';  import { PostService } from '../services/post.service';  import { NotFoundError } from 'src/app/not-found-error';  @Component({  selector: 'posts',  templateUrl: './posts.component.html',  styleUrls: ['./posts.component.css']  })  export class PostsComponent implements OnInit {  posts: any[];  constructor(private service:PostService) {  }  ngOnInit() {  this.service.getAll()  .subscribe(  response =>{  this.posts = response.json();  });  }  createPost(input: HTMLInputElement){  let post: any = {title: input.value};  input.value ='';  this.service.Create(post)  .subscribe(  response =>{  post.id = response.json().id;  this.posts.splice(0,0,post);  },  (error:AppError)=>{  if(error instanceof BadRequestError){  }  else throw error;  });  }  updatePost(post){  this.service.update(post)  .subscribe(  response =>{  console.log(response.json());  });  }  deletePost(post){  this.service.delete(post.id)  .subscribe(  response =>{  let index = this.posts.indexOf(post);  this.posts.splice(index,1);  },  (error: AppError) =>{  if(error instanceof NotFoundError)  {  alert('This post has already been deleted');  }  else throw error;  });  }  } |

### The Map Operator

We have still have one issue here in Post.component. In all the method of the class where we are subscribe the observable we are working with response object.

As we know we should not work with response object in our component. **Here instead ideally instead of getting the response object then calling the JSON method on it, I want to get the array of object**

**Now in data-service we will work with one of the observable operator called map. With Map operator we can transfor the item in an observable.**

**Step1:-**

**Can the data service to convert response object to object of array**

|  |
| --- |
| import { BadRequestError } from './../bad-request-error';  import { AppError } from './../posts/app-error';  import { Http } from '@angular/http';  import { Injectable } from '@angular/core';  import { catchError, map } from 'rxjs/operators';  import { Observable } from 'rxjs';  import { NotFoundError } from '../not-found-error';  @Injectable({  providedIn: 'root'  })  export class DataService {  constructor(private url: string,private http: Http) { }  getAll(){  return this.http.get(this.url)    .pipe(  //Here we are transforming the response object  //to an array of javascript object  map(response => response.json()),  catchError(this.handleError)  );  }  Create(resource){  return this.http.post(this.url,JSON.stringify(resource))  .pipe(  map(response => response.json()),  catchError(this.handleError));  }  update(resource){  return this.http.patch(this.url + '/'+ resource.id,JSON.stringify({isRead:true}))  .pipe(  map(response => response.json()),  catchError(this.handleError));  }  delete(id){  return this.http.delete(this.url +'/' + id)  .pipe(  map(response => response.json()),  catchError(this.handleError));  }  private handleError(error:Response){  if(error.status==400)  return Observable.throw(new BadRequestError(error.json));  if(error.status==404)  return Observable.throw(new NotFoundError());    return Observable.throw(new AppError(error));  }  } |

Step2:-

In component change the response object to array to object

|  |
| --- |
| import { BadRequestError } from './../bad-request-error';  import { AppError } from './app-error';  import { Component, OnInit } from '@angular/core';  import { PostService } from '../services/post.service';  import { NotFoundError } from 'src/app/not-found-error';  @Component({  selector: 'posts',  templateUrl: './posts.component.html',  styleUrls: ['./posts.component.css']  })  export class PostsComponent implements OnInit {  posts: any[];  constructor(private service:PostService) {  }  ngOnInit() {  this.service.getAll()  //Here instead of getting response  // we will get array of object  .subscribe(posts => this.posts = posts);  }  createPost(input: HTMLInputElement){  let post: any = {title: input.value};  input.value ='';  this.service.Create(post)  .subscribe(  newPost =>{  post.id = newPost.id;  this.posts.splice(0,0,post);  },  (error:AppError)=>{  if(error instanceof BadRequestError){  }  else throw error;  });  }  updatePost(post){  this.service.update(post)  .subscribe(  updatePost =>console.log(updatePost))}  deletePost(post){  this.service.delete(post.id)  .subscribe(  () =>{  let index = this.posts.indexOf(post);  this.posts.splice(index,1);  },  (error: AppError) =>{  if(error instanceof NotFoundError)  {  alert('This post has already been deleted');  }  else throw error;  });  }  } |

### Optimistic vs Pessimistic updates

Here we will always use Optimistic update means while updating if any error occurs we will rollback the changes.

So in the component we will first update/delete any object at the beginning only. And in the error handler function if any error occurs to will update/add back the object.

### Observables vs Promises

* **Observables are lazy** otherwise nothing happens until you subscribe to that.
* **Promises are eager** as soon we create a promise the code is executed.
* So convert observables to Promises we use the operator called “**toPromise**”.
* **retry**(3) is an operator which we can use to reconnect the server for n times if it fails.
* With all this use of operator leads to reactive programming which is coming from ‘rxjs’.

We always convert observables to Promises. Prefer Observables they allow reactive programming. They provide a bunch of useful operators.

### Use RxJS operator to transform the Response Data

**Using map Operator**

As we pipe() is a method used in to funnel your observable data through multiple operators before they reach the subscribe method.

Map operator allows getting some data and returning new data which is automatically re-wrapped into an observable, so that we still subscribe to it.

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { HttpClient } from '@angular/common/http';  import { map } from 'rxjs/operators';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {  loadedPosts = [];  constructor(private http: HttpClient) {}  ngOnInit() {  this.fetchPosts();  }  onCreatePost(postData: { title: string; content: string }) {  // Send Http request  this.http  .post(  'http://localhost:62286/Api/TitleDescription/InsertTitleDescription',  postData  )  .subscribe(responseData => {  console.log(responseData);  });  }  onFetchPosts() {  // Send Http request  this.fetchPosts();  }  onClearPosts() {  // Send Http request  }  private fetchPosts(){  this.http.get('http://localhost:62286/Api/TitleDescription/AllTitleDescription')  .pipe(map(responseData =>{  const postsArray = [];  for( const key in responseData){  if(responseData.hasOwnProperty(key)){  postsArray.push({...responseData[key], id:key });  }  }  return postsArray;  })  )  .subscribe(post => {  console.log(post);  });  }  } |

### Using the Types with the HttpClient

We can define the type of the data receive in the http verb

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { HttpClient } from '@angular/common/http';  import { map } from 'rxjs/operators';  import { Post } from './post.model';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {  loadedPosts = [];  constructor(private http: HttpClient) {}  ngOnInit() {  this.fetchPosts();  }  onCreatePost(postData: Post) {  // Send Http request  this.http  .post<{Post}>(  'http://localhost:62286/Api/TitleDescription/InsertTitleDescription',  postData  )  .subscribe(responseData => {  console.log(responseData);  });  }  onFetchPosts() {  // Send Http request  this.fetchPosts();  }  onClearPosts() {  // Send Http request  }  private fetchPosts(){  this.http  .get<{[key: string]: Post}>('http://localhost:62286/Api/TitleDescription/AllTitleDescription')  .pipe(map(responseData => {  const postsArray: Post[] = [];  for( const key in responseData){  if(responseData.hasOwnProperty(key)){  postsArray.push({...responseData[key], id:key });  }  }  return postsArray;  })  )  .subscribe(post => {  console.log(post);  });  }  } |

Her Post is the interface define in the model

|  |
| --- |
| export interface Post {  title: string,  content: string,  id?: string  } |

### Showing a Laoding Indicator

While loading the GET request, the loading Indicator will show unless and until the request is completed.

**In Component:-**

Declare a variable to false and make true will the request is processing.

|  |
| --- |
| import { Observable } from 'rxjs';  import { Component, OnInit } from '@angular/core';  import { HttpClient } from '@angular/common/http';  import { map } from 'rxjs/operators';  import { Post } from './post.model';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {  loadedPosts: Post[]= [];  isFetching = false;  constructor(private http: HttpClient) {}  ngOnInit() {  this.fetchPosts();  }  onCreatePost(postData: Post) {  // Send Http request  this.http  .post<{Post}>(  'http://localhost:62286/Api/TitleDescription/InsertTitleDescription',  postData  )  .subscribe(responseData => {  console.log(responseData);  });  }  onFetchPosts() {  // Send Http request  this.fetchPosts();  }  onClearPosts() {  // Send Http request  }  private fetchPosts(){  this.isFetching = true  this.http.get<Post[]>('http://localhost:62286/Api/TitleDescription/AllTitleDescription')  .subscribe(  post => {  this.isFetching = false;  this.loadedPosts = post;  //console.log(this.loadedPosts);  });  }  } |

**In Template**

Use this variable and create a paragraph which will be enabled when the request is processing.

|  |
| --- |
| <div class="container">  <div class="row">  <div class="col-xs-12 col-md-6 col-md-offset-3">  <form #postForm="ngForm" (ngSubmit)="onCreatePost(postForm.value)">  <div class="form-group">  <label for="title">Title</label>  <input  type="text"  class="form-control"  id="title"  required  ngModel  name="title"  />  </div>  <div class="form-group">  <label for="content">Content</label>  <textarea  class="form-control"  id="Content"  required  ngModel  name="Content"  ></textarea>  </div>  <button  class="btn btn-primary"  type="submit"  [disabled]="!postForm.valid"  >  Send Post  </button>  </form>  </div>  </div>  <hr />  <div class="row">  <div class="col-xs-12 col-md-6 col-md-offset-3">  <button class="btn btn-primary" (click)="onFetchPosts()">  Fetch Posts  </button>  |  <button  class="btn btn-danger"  [disabled]="loadedPosts.length < 1"  (click)="onClearPosts()"  >  Clear Posts  </button>  </div>  </div>  <div class="row">  <div class="col-xs-12 col-md-6 col-md-offset-3">  <p \*ngIf="loadedPosts.length < 1 && !isFetching" >No posts available!</p>  <ul class="list-group" \*ngIf = "loadedPosts.length >=1 && !isFetching">  <li class="list-group-item" \*ngFor ="let post of loadedPosts;">  <h3>{{post.Title}}</h3>  <p>{{post.Content}}</p>  </li>  </ul>  <p \*ngIf ="isFetching">Loading..</p>  </div>  </div>  </div> |

### Setting Headers

Sometimes we need to set some special header while sending request to server, forexample we have backend that required authorization which looks for authorization header or need to send content type or need to attach a custome header because the API to whom you are sending the request needs it.

|  |
| --- |
| import { HttpClient, HttpHeaders } from '@angular/common/http';  import { Injectable } from '@angular/core';  import { Post } from './post.model';  import { catchError } from 'rxjs/operators';  import { throwError } from 'rxjs';  @Injectable({providedIn:'root'})  export class PostsService{  constructor(private http: HttpClient){}  CreateAndStorePost(title: string, content: string){  const postData: Post = {  title : title,  content : content  }  this.http  .post<{Post}>(  'http://localhost:62286/Api/TitleDescription/InsertTitleDescription',  postData  )  .subscribe(responseData => {  console.log(responseData);  });  }  fetchPosts(){  return this.http.get<Post[]>(  'http://localhost:62286/Api/TitleDescription/AllTitleDescription',  {  headers: new HttpHeaders({'Custom-Header': 'Hello'})  }  );  }  deletePost(){  return this.http.delete('http://localhost:62286/Api/TitleDescription/DeleteTitleDescriptions');  }  } |

We can add to any Http Verb the Headers.

### Adding Query Params

We can query parameter to the url of the http verb we are request to API.

|  |
| --- |
| import { HttpClient, HttpHeaders, HttpParams } from '@angular/common/http';  import { Injectable } from '@angular/core';  import { Post } from './post.model';  import { catchError } from 'rxjs/operators';  import { throwError } from 'rxjs';  @Injectable({providedIn:'root'})  export class PostsService{  constructor(private http: HttpClient){}  CreateAndStorePost(title: string, content: string){  const postData: Post = {  title : title,  content : content  }  this.http  .post<{Post}>(  'http://localhost:62286/Api/TitleDescription/InsertTitleDescription',  postData  )  .subscribe(responseData => {  console.log(responseData);  });  }  fetchPosts(){  let searchParams = new HttpParams();  searchParams = searchParams.append('print','chiranjit');  searchParams = searchParams.append('Hello','Hi');  return this.http.get<Post[]>(  'http://localhost:62286/Api/TitleDescription/AllTitleDescription',  {  headers: new HttpHeaders({'Custom-Header': 'Hello'}),  params: searchParamss  }  );  }  deletePost(){  return this.http.delete('http://localhost:62286/Api/TitleDescription/DeleteTitleDescriptions');  }  } |

### Observing different types of responses

Sometimes we need access to the entire response objects no just the extracted body data, like status code , response headers

|  |
| --- |
| CreateAndStorePost(title: string, content: string){  const postData: Post = {  title : title,  content : content  }  this.http  .post<{Post}>(  'http://localhost:62286/Api/TitleDescription/InsertTitleDescription',  postData,  {  observe:'response'  }  )  .subscribe(responseData => {  console.log(responseData);  });  } |

By adding the highlighted code, we can get the complete reponse object

|  |
| --- |
|  |

### Changing the Response Body Type

|  |
| --- |
| fetchPosts(){  let searchParams = new HttpParams();  searchParams = searchParams.append('print','chiranjit');  searchParams = searchParams.append('Hello','Hi');  return this.http.get<Post[]>(  'http://localhost:62286/Api/TitleDescription/AllTitleDescription',  {  headers: new HttpHeaders({'Custom-Header': 'Hello'}),  params: searchParams,  responseType:'json'  }  );  } |

### Introducing Interceptors

Suppose you want to attach certain params, headers to every outgoing request, so that the backend can read that. We don’t want to manually configure every request, for that we can add Interceptors

Check the max video 271

# Routing and Navigation

### Routing in a Nutshell

We are going to use **Route** Module for Routing. **Three steps to implement navigation**:-

1. **Configure the routes** in our application. Each route determines what components should be visible to the user navigates to certain URL. (**Route is the mapping of a path to a component**)
2. **Add a router outlet** that’s where we display the corresponding component when a given routes become active.
3. **Add links.**

### Configuring Routes

**Step 1:-**

Import RouteModule from ‘angular/router’ in app.module.ts

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

**Step 2:**

Define routes in imports array of app.module.ts

imports: [

BrowserModule,

FormsModule,

ReactiveFormsModule,

HttpModule,

RouterModule.forRoot()

]

* **forRoot()** is a static method define in the router module class and we use this to define root route of the application.
* And inside forRoot we pass **array of routes** and **each route is an object with two properties :- 1) path and 2) component**

|  |
| --- |
| imports: [  BrowserModule,  FormsModule,  ReactiveFormsModule,  HttpModule,  RouterModule.forRoot([  {path:'', component: HomeComponent}  ])  ] |

* **And empty path represents the home page or default route.**
* And if we want to pass parameter to the URL, have to use **:parametername**

{path:'profile/:username',component:GithubProfileComponent}

* **How to display a nonfound page when user type invalid url, using path as \*\* ( which is default route) and it should place at the end of route.**

{

path:'\*\*',

component: NotFoundComponent

}

**Complete code:**

|  |
| --- |
| imports: [  BrowserModule,  FormsModule,  ReactiveFormsModule,  HttpModule,  RouterModule.forRoot([  {  path:'',  component: HomeComponent  },  {  path:'followers/:username',  component:GithubProfileComponent  },  {  path:'followers',  component:GithubFollowersComponent  },  {  path:'posts',  component:PostsComponent  },  {  path:'\*\*',  component: NotFoundComponent  }  ])  ], |

**MAX :- Also from Max lecture we can declare everything in constant and then used in Router.forRoot()**

|  |
| --- |
| import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule } from '@angular/forms';  import { AppComponent } from './app.component';  import { HomeComponent } from './home/home.component';  import { UsersComponent } from './users/users.component';  import { ServersComponent } from './servers/servers.component';  import { UserComponent } from './users/user/user.component';  import { EditServerComponent } from './servers/edit-server/edit-server.component';  import { ServerComponent } from './servers/server/server.component';  import { ServersService } from './servers/servers.service';  import { Routes, RouterModule } from '@angular/router'  const appRoutes: Routes = [  { path: '', component: HomeComponent },  { path: 'users', component: UserComponent},  { path: 'servers', component: ServerComponent},  ];  @NgModule({  declarations: [  AppComponent,  HomeComponent,  UsersComponent,  ServersComponent,  UserComponent,  EditServerComponent,  ServerComponent  ],  imports: [  BrowserModule,  FormsModule,  RouterModule.forRoot(appRoutes)  ],  providers: [ServersService],  bootstrap: [AppComponent]  })  export class AppModule { } |

### RouteOutlet

* Add a router outlet and that’s where angular router is going to display a component that is associated with the route

**<router-outlet></router-outlet>** - this is the directive present in router module.

<navbar></navbar>

<router-outlet></router-outlet>

### RouterLink

In Angular in hyperlink (<a href=”#”>Follewers</a>) **we don’t use href because with this navigation every time we click on the link is downloaded and Angular app is reinitialized, that’s why the page goes for just a split second. As your application grows that delay will be even more.**

**Now when we click on the link we want to download the content of that page only not the entire application. So fix this issue instead of href attribute, we are going to use directive called** **routerLink**

|  |
| --- |
| <nav class="navbar navbar-default">  <div class="container-fluid">  <div class="collapse navbar-collapse">  <ul class="nav navbar-nav">  <li class="active"><a routerLink="/followers">Followers</a></li>  <li><a routerLink="/posts">Posts</a></li>  </ul>  </div>  </div>  </nav> |

Now the single page is downloaded from server and as the user navigates to one page to another, only the content of the target page is downloaded. This is referred as **Single Page Application (SPA).**

**Now how to work with dynamic URL or Pass parameter? When we are dealing with route parameter instead of using routerLink as an attribute we should use property binding syntax.**

|  |
| --- |
| <div \*ngFor="let follower of followers" class="media">  <div class="media-left">  <a href="#">  <img class="avatar media-object" src="{{ follower.avatar\_url }}" alt="...">  </a>  </div>  <div class="media-body">  <h4 class="media-heading">  <a [routerLink]="['followers',follower.id]">{{ follower.login }}</a>  </h4>  <a href="follower.html\_url">{{ follower.html\_url }}</a>  </div>  </div> |

### RouteLinkActive

<nav class="navbar navbar-default">

<div class="container-fluid">

<div class="collapse navbar-collapse">

<ul class="nav navbar-nav">

<li class="active"><a routerLink="/followers">Followers</a></li>

<li><a routerLink="/posts">Posts</a></li>

</ul>

</div>

</div>

</nav>

In the above example the Followers link will always be highlighted as we mark the bootstrap class to active.

Now how to apply it dynamically, we have directive called **routerLinkActive** and value in the routerLinkActive includes the list of CSS classes. In the below example we use two css classes active and current.

|  |
| --- |
| <nav class="navbar navbar-default">  <div class="container-fluid">  <div class="collapse navbar-collapse">  <ul class="nav navbar-nav">  <li routerLinkActive="active current"><a routerLink="/followers">Followers</a></li>  <li routerLinkActive="active current"><a routerLink="/posts">Posts</a></li>  </ul>  </div>  </div>  </nav> |

### RouterLinkActiveOptions

See max 129 lecture

If by default the Home tab should not be selected all the time.

|  |
| --- |
| <div class="container">  <div class="row">  <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">  <ul class="nav nav-tabs">  <li role="presentation"  routerLinkActive="active"  [routerLinkActiveOptions]="{exact: true}">  <a routerLink="/">Home</a>  </li>  <li role="presentation"  routerLinkActive="active">  <a routerLink="/users">Users</a>  </li>  <li role="presentation"  routerLinkActive="active">  <a [routerLink]="['servers']">Servers</a>  </li>  </ul>  </div>  </div>  <div class="row">  <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">  <router-outlet></router-outlet>  </div>  </div>  </div> |

### Getting the Route Parameters

Eg

|  |
| --- |
| const appRoutes: Routes = [  { path: '', component: HomeComponent },  { path: 'users', component: UsersComponent},  { path: 'users/:id', component: UsersComponent},  { path: 'servers', component: ServersComponent},  ]; |

**Step 1:-**

We need to inject activated route class in our constructor of the particular component

|  |
| --- |
| import {ActivatedRoute} from '@angular/router';  export class GithubProfileComponent implements OnInit {  constructor(route: ActivatedRoute) { }  ngOnInit() {  }  } |

**Step 2:-**

In ngOnInit get the route parameter from this object, as the **paramMap is observable** we need to subscribe and get the value using get method.

**Here + used to convert string to number.**

|  |
| --- |
| export class GithubProfileComponent implements OnInit {  constructor(private route: ActivatedRoute) { }  ngOnInit() {  this.route.paramMap  .subscribe(params =>{  let id = +params.get('id');  });  }  Or  //it will not work for the same page  this.user ={  id:this.route.snapshot.params[‘id’];  }  Or  //to work route parameters reactively  this.route.params  .subscribe(  (params:Params) => {  this.user.id = params[‘id’]  }  ); |

### Why Route Parameters are Observable

Observable are a feature add by other third party package but heavily used by angular which allows us to easily works with asynchronous task (the task may occur anytime in future , we don’t know when , howmuch time it will take)

|  |
| --- |
|  |

In above case angular is going to destroy component A remove it from the DOM and it will going to create or initialize Component B, and renders its template in the DOM.

|  |
| --- |
|  |

Here in Component A we have a button that allows the user away from this page and also get to the same page.

Example:

|  |
| --- |
|  |

On Transaction details page we have two buttons previous and next, when the user clicks on either of the buttons (but technically they are on the same page). In this case it doesn’t make sense for Angular to destroy this component only to recreate it straight way after.

**In the particular instance what we have is the same component instance but different route parameter, so Angular is not going to destroy this component and is going to keep it in the DOM. This is exact reasons why route parameter is defined as observable.**

**Observables is technically a collection of asynchronous data that arrives over time.While consuming HTTP services we simply subscribe to observable and get the response from the server, in that particular case we have one item object in our observable that was response from the server.**

**But in our sure that users has to navigate away from the page goes some else and then come back. In that instead of subscribing an observable, we can use a snapshot.**

**And here paramMap will not be observable.**

|  |
| --- |
| ngOnInit() {  this.route.snapshot.paramMap.get(‘id’);  //this.route.paramMap  //.subscribe(params =>{  //let id = +params.get('id');  //console.log(id);  //});  } |

### Routes with Multiple Parameters

**Step 1:**

Add the parameter in the app.module.ts

|  |
| --- |
| RouterModule.forRoot([  {  path:'',  component:HomeComponent  },  {  path:'profile/:id:username',  component:GithubProfileComponent  }, |

Step 2:-

Add the parameter in routerLink inside the template, here first element is the page and the subsequent elements are router parameter.

|  |
| --- |
| <div \*ngFor="let follower of followers" class="media">  <div class="media-left">  <a href="#">  <img class="avatar media-object" src="{{ follower.avatar\_url }}" alt="...">  </a>  </div>  <div class="media-body">  <h4 class="media-heading">  <a [routerLink]="['/followers',follower.id,follower.login]">{{ follower.login }}</a>  </h4>  <a href="follower.html\_url">{{ follower.html\_url }}</a>  </div>  </div> |

### Query Parameters

How to add optional parameter to the route?

**Step1:- (sending part)**

Bind with queryparams property to an object and of Couse value of the key can be dynamic.

|  |
| --- |
| <nav class="navbar navbar-default">  <div class="container-fluid">  <div class="collapse navbar-collapse">  <ul class="nav navbar-nav">  <li routerLinkActive="active current"><a routerLink="/followers"  [queryParams]="{ page: 1, order: 'newest' }">Followers</a></li>  <li routerLinkActive="active current"><a routerLink="/posts">Posts</a></li>  </ul>  </div>  </div>  </nav> |

**Step2:- (receiving the parameter)**

Instead of paramMap property we use queryParamMap property

|  |
| --- |
| import { ActivatedRoute } from '@angular/router';  import { GithubFollowersService } from './../services/github-followers.service';  import { Component, OnInit } from '@angular/core';  @Component({  selector: 'github-followers',  templateUrl: './github-followers.component.html',  styleUrls: ['./github-followers.component.css']  })  export class GithubFollowersComponent implements OnInit {  followers: any[];  constructor(  private route: ActivatedRoute,  private service: GithubFollowersService) { }  ngOnInit() {  this.route.queryParamMap.subscribe();  //or  this.route.snapshot.queryParamMap.get('page');  this.service.getAll()  .subscribe(followers => this.followers = followers);  }  } |

### Add Fragments(#) in the URL

Check max video 135

|  |
| --- |
| <div class="row">  <div class="col-xs-12 col-sm-4">  <div class="list-group">  <a  [routerLink] ="['/servers',5, 'edit']"  [queryParams] = "{allowEdit: 1}"  fragment = "loading"  href="#"  class="list-group-item"  \*ngFor="let server of servers">  {{ server.name }}  </a>  </div>  </div> |

### Programmatically how to pass queryParams and Fragment

* + url will looks like <http://localhost:4200/servers/1/edit?allowEdit=1#loading>
  + And onLoadServers will call on button click

|  |
| --- |
| onLoadServers(id: number){  this.router.navigate(['/servers',id, 'edit'], {queryParams:{allowEdit:'1'},fragment:'loading'});  } |

### Retreiving QueryParams and Fragments

* + Using ActivatedRoute we need to do this.
  + And Using SnapShot or Subscribe

1. Using SnapShot

|  |
| --- |
| import { ActivatedRoute } from '@angular/router';  import { Component, OnInit } from '@angular/core';  import { ServersService } from '../servers.service';  @Component({  selector: 'app-edit-server',  templateUrl: './edit-server.component.html',  styleUrls: ['./edit-server.component.css']  })  export class EditServerComponent implements OnInit {  server: {id: number, name: string, status: string};  serverName = '';  serverStatus = '';  constructor(private serversService: ServersService,  private route: ActivatedRoute) { }  ngOnInit() {  //this will only run on update at the time this component is created  //so their is chance of changing the query params from the page we are currently on  //we might not this approach  console.log(this.route.snapshot.queryParams);  console.log(this.route.snapshot.fragment); } |

1. Another approach using queryparams, fragment and subscribe

|  |
| --- |
| import { ActivatedRoute } from '@angular/router';  import { Component, OnInit } from '@angular/core';  import { ServersService } from '../servers.service';  @Component({  selector: 'app-edit-server',  templateUrl: './edit-server.component.html',  styleUrls: ['./edit-server.component.css']  })  export class EditServerComponent implements OnInit {  server: {id: number, name: string, status: string};  serverName = '';  serverStatus = '';  constructor(private serversService: ServersService,  private route: ActivatedRoute) { }  ngOnInit() {  //this will only run on update at the time this component is created  //so their is chance of changing the query params from the page we are currently on  //we might not this approach  console.log(this.route.snapshot.queryParams);  console.log(this.route.snapshot.fragment);  this.route.queryParams.subscribe();  this.route.fragment.subscribe();    this.server = this.serversService.getServer(1);  this.serverName = this.server.name;  this.serverStatus = this.server.status;  }  onUpdateServer() {  this.serversService.updateServer(this.server.id, {name: this.serverName, status: this.serverStatus});  }  } |

### Subscribing to Multiple Observables

We have combine two/multiple observable to a single observable, then we need to subscribe.

**Step1:-**

Import observable to the component.

|  |
| --- |
| import { Observable } from 'rxjs/Observable'; |

And import the factory method , we can use this to create new observable.

|  |
| --- |
| import 'rxjs/add/observable/combineLatest'; |

Step2:-

Create the new observable and subscribe

|  |
| --- |
| import { Observable } from 'rxjs/Observable';  import 'rxjs/add/observable/combineLatest';  export class GithubFollowersComponent implements OnInit {  followers: any[];  constructor(  private route: ActivatedRoute,  private service: GithubFollowersService) { }  ngOnInit() {  Observable.combineLatest([  this.route.paramMap,  this.route.queryParamMap  ])  .subscribe(combine =>{  let id =combine[0].get('id');  let page = combine[1].get('page');  this.service.getAll()  .subscribe(followers => this.followers = followers);  });  }  } |

### SwitchMap Operator

Firstly for getAll() of service it will return observable<response> , then after applying map operator it will return observable<any>

|  |
| --- |
|  |

Now after applying map in paramMap[] it will return followers array

|  |
| --- |
|  |

Below example is with two subscribe in a observable

|  |
| --- |
| ngOnInit() {  Observable.combineLatest([  this.route.paramMap,  this.route.queryParamMap  ])  .subscribe(combine =>{  let id =combine[0].get('id');  let page = combine[1].get('page');  this.service.getAll()  .subscribe(followers => this.followers = followers);  });  } |

Below using Map operator to transform the object in our observable

|  |
| --- |
| ngOnInit() {  Observable.combineLatest([  this.route.paramMap,  this.route.queryParamMap  ])  .map(combined =>{  let id =combined[0].get('id');  let page = combined[1].get('page');  return this.service.getAll()  })  .subscribe(followers => this.followers = followers);  } |

Above when we subscribe to this observable every item in this observable is going to be an observable of any. It likes collection inside another collection.

In the subscriber we want followers array not observable of followers array.

So to fix this problem instead of map operator we will use switchMap

|  |
| --- |
| ngOnInit() {  Observable.combineLatest([  this.route.paramMap,  this.route.queryParamMap  ])  .switchMap(combined =>{  let id =combined[0].get('id');  let page = combined[1].get('page');    return this.service.getAll()  })  .subscribe(followers => this.followers = followers);  } |

### Programmatic Navigation (router.navigate)

Now to navigate to user

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import {Router} from '@angular/router';  @Component({  selector: 'app-github-profile',  templateUrl: './github-profile.component.html',  styleUrls: ['./github-profile.component.css']  })  export class GithubProfileComponent implements OnInit {  constructor(private router: Router) { }  ngOnInit() {  }  submit(){  //first argument in the array is path and subsequently  //the other will be parameter  // this.router.navigate(['/followers',param1,param2])  //with query parameter  this.router.navigate(['/followers'],{  queryParams:{page:1,order:'newest'}  }  } |

### Using Relative Paths in Programmatic navigation

|  |
| --- |
| import { Router, ActivatedRoute } from '@angular/router';  import { Component, OnInit } from '@angular/core';  import { ServersService } from '../servers.service';  @Component({  selector: 'app-server',  templateUrl: './server.component.html',  styleUrls: ['./server.component.css']  })  export class ServerComponent implements OnInit {  server: {id: number, name: string, status: string};  constructor(private serversService: ServersService,  private router: Router,  private route: ActivatedRoute) { }  ngOnInit() {  this.server = this.serversService.getServer(1);  }  onReload(){  this.router.navigate(['servers'],{relativeTo: this.route});  }  } |

### Setting up Child (Nested) Routes

Check the video of Max 139

|  |
| --- |
| const appRoutes: Routes = [  { path: '', component: HomeComponent },  { path: 'users', component: UsersComponent},  { path: 'users/:id/:name', component: UserComponent},  { path: 'servers', component: ServersComponent,children:[  { path: ':id', component: ServerComponent},  { path: ':id/edit', component: EditServerComponent}  ]}  ]; |

And also we need separate router-outlet inside the component we added child component instead of calling separately.

|  |
| --- |
| <div class="row">  <div class="col-xs-12 col-sm-4">  <div class="list-group">  <a  [routerLink] ="['/servers', server.id]"  [queryParams] = "{allowEdit: 1}"  fragment = "loading"  href="#"  class="list-group-item"  \*ngFor="let server of servers">  {{ server.name }}  </a>  </div>  </div>  <div class="col-xs-12 col-sm-4">  <router-outlet></router-outlet>  <!--<a routerLink="../servers">Reload</a>  <app-edit-server></app-edit-server>  <hr>  <app-server></app-server> -->  </div>  </div> |

### queryParamsHandling

Is use to preserve the queryparams value after redirecting to other page.

|  |
| --- |
| onEdit(){  this.router.navigate(['edit'],{relativeTo:this.route, queryParamsHandling:'preserve'});  } |

### Redirecting and wildcard Routes

Step 1:- Create a new component like page-not-found

Step 2:- Use that component to display the error

Using path = \*\* which tells any wildcard search is not allowed and it should at the end of all route.

|  |
| --- |
| const appRoutes: Routes = [  { path: '', component: HomeComponent },  { path: 'users', component: UsersComponent, children:[  { path: ':id/:name', component: UserComponent}  ]},    { path: 'servers', component: ServersComponent,children:[  { path: ':id', component: ServerComponent},  { path: ':id/edit', component: EditServerComponent}  ]},  {path:'notFound', component: PageNotFoundComponent},  {path:'\*\*', redirectTo:'/notFound'}  ]; |

### Outsourcing the Route Configuration

So instead of writing the route configuration inside the app.module.ts we can create new module for the routing

**app-routing.module.ts**

|  |
| --- |
| import { NgModule } from "@angular/core";  import { Routes, RouterModule } from '@angular/router';  import { HomeComponent } from "./home/home.component";  import { UsersComponent } from "./users/users.component";  import { UserComponent } from "./users/user/user.component";  import { ServersComponent } from "./servers/servers.component";  import { ServerComponent } from "./servers/server/server.component";  import { EditServerComponent } from "./servers/edit-server/edit-server.component";  import { PageNotFoundComponent } from "./page-not-found/page-not-found.component";  const appRoutes: Routes = [  { path: '', component: HomeComponent },  { path: 'users', component: UsersComponent, children:[  { path: ':id/:name', component: UserComponent}  ]},    { path: 'servers', component: ServersComponent,children:[  { path: ':id', component: ServerComponent},  { path: ':id/edit', component: EditServerComponent}  ]},  {path:'notFound', component: PageNotFoundComponent},  {path:'\*\*', redirectTo:'/notFound'}  ];    @NgModule({  imports:[  RouterModule.forRoot(appRoutes)  ],  exports: [RouterModule]  })  export class AppRoutingModule {    } |

**In app.module.ts**

Remove everything only need to import the newly created module.

|  |
| --- |
| imports: [  BrowserModule,  FormsModule,  AppRoutingModule  ], |

### Protecting Routes with canActivate

**Step 1:-**

Create a new guard which is nothing but a typescript file

Auth-guard.service.ts

Inside create a class which implements the interface CanActivate Interface, which can active return observable<Boolean> or Promise<Boolean> or Boolean depends on the asynchronous (observable<Boolean> or Promise<Boolean>) and syschronous(Boolean) operation

|  |
| --- |
| import { AuthService } from './auth.service';  import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router } from "@angular/router";  import { Observable } from "rxjs";  import { Injectable } from "@angular/core";  @Injectable()  export class AuthGuard implements CanActivate{  constructor(private authGuard: AuthService, private route: Router) {}  canActivate(route: ActivatedRouteSnapshot,  state: RouterStateSnapshot) : Observable<boolean> | Promise<boolean>| boolean{    return this.authGuard.isAuthenticated()  .then(  (authenticated: boolean) => {  if(authenticated){  return true;  }  else{  this.route.navigate['/'];  }  }  );  }  } |

**Fake Auth Service**

|  |
| --- |
| import { resolve } from "path";  import { reject } from "q";  export class AuthService {  loggenIn = false;  isAuthenticated(){  const promise = new Promise(  (resolve, reject) => {  setTimeout(() => {  resolve(this.loggenIn)  }, 800);  }  );  return promise;  }  login(){  this.loggenIn = true;  }  logout(){  this.loggenIn = false;  }  } |

**Step 2:-**

Apply the guard in the route you want to apply automatically applied to the child route.

|  |
| --- |
| const appRoutes: Routes = [  { path: '', component: HomeComponent },  { path: 'users', component: UsersComponent, children:[  { path: ':id/:name', component: UserComponent}  ]},    { path: 'servers', canActivate:[AuthGuard], component: ServersComponent,children:[  { path: ':id', component: ServerComponent},  { path: ':id/edit', component: EditServerComponent}  ]},  {path:'notFound', component: PageNotFoundComponent},  {path:'\*\*', redirectTo:'/notFound'}  ]; |

### Protecting Child Nested Routes with canActivateChild

**Step 1**

In Auth Guard implement canActivateChild interface

|  |
| --- |
| import { AuthService } from './auth.service';  import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router, CanActivateChild } from "@angular/router";  import { Observable } from "rxjs";  import { Injectable } from "@angular/core";  @Injectable()  export class AuthGuard implements CanActivate, CanActivateChild{  constructor(private authGuard: AuthService, private route: Router) {}  canActivate(route: ActivatedRouteSnapshot,  state: RouterStateSnapshot) : Observable<boolean> | Promise<boolean>| boolean{    return this.authGuard.isAuthenticated()  .then(  (authenticated: boolean) => {  if(authenticated){  return true;  }  else{  this.route.navigate['/'];  }  }  );  }  canActivateChild(route: ActivatedRouteSnapshot,  state: RouterStateSnapshot) : Observable<boolean> | Promise<boolean>| boolean {  return this.canActivate(route, state);  }  } |

**Step 2:-**

Add the guard with canActivateChild instead of canActivate

|  |
| --- |
| const appRoutes: Routes = [  { path: '', component: HomeComponent },  { path: 'users', component: UsersComponent, children:[  { path: ':id/:name', component: UserComponent}  ]},    { path: 'servers',  //canActivate:[AuthGuard],  canActivateChild:[AuthGuard],  component: ServersComponent,  children:  [  { path: ':id', component: ServerComponent},  { path: ':id/edit', component: EditServerComponent}  ]  },  {path:'notFound', component: PageNotFoundComponent},  {path:'\*\*', redirectTo:'/notFound'}  ]; |

### Controlling Navigation with canDeactivate

Step 1:-

Create a new auth guard (can-deactive-guard.service.ts) inside create an interface and implements the interface canDeactivate

|  |
| --- |
| import { Observable } from "rxjs/Observable";  import { CanDeactivate, ActivatedRouteSnapshot, RouterStateSnapshot } from "@angular/router";  export interface canComponentDeactive{  canDeactive:() => Observable<boolean> | Promise<boolean> | boolean;  }  export class canDeactiveGuard implements CanDeactivate<canComponentDeactive>{  canDeactivate(component: canComponentDeactive,  currentRoute: ActivatedRouteSnapshot,  currentState: RouterStateSnapshot,  nextState?: RouterStateSnapshot) : Observable<boolean> | Promise<boolean> | boolean{    return component.canDeactive();  }  } |

Step 2:-

In Route add the canDeactive route in the particular component.

|  |
| --- |
| const appRoutes: Routes = [  { path: '', component: HomeComponent },  { path: 'users', component: UsersComponent, children:[  { path: ':id/:name', component: UserComponent}  ]},    { path: 'servers',  //canActivate:[AuthGuard],  canActivateChild:[AuthGuard],  component: ServersComponent,  children:  [  { path: ':id', component: ServerComponent},  { path: ':id/edit', component: EditServerComponent, canDeactivate:[canDeactiveGuard]}  ]  },  {path:'notFound', component: PageNotFoundComponent},  {path:'\*\*', redirectTo:'/notFound'}  ]; |

Step 3:

Implements the particular interface (deActive in the component)

|  |
| --- |
| import { canComponentDeactive } from './can-deactive-guard.service';  import { ActivatedRoute, Params, Router } from '@angular/router';  import { Component, OnInit } from '@angular/core';  import { ServersService } from '../servers.service';  import { Observable } from 'rxjs/Observable';  @Component({  selector: 'app-edit-server',  templateUrl: './edit-server.component.html',  styleUrls: ['./edit-server.component.css']  })  export class EditServerComponent implements OnInit, canComponentDeactive {  server: {id: number, name: string, status: string};  serverName = '';  serverStatus = '';  allowEdit = false;  ChangesSaved = false;  constructor(private serversService: ServersService,  private route: ActivatedRoute,  private router: Router) { }  ngOnInit() {  //this will only run on update at the time this component is created  //so their is chance of changing the query params from the page we are currently on  //we might not this approach  console.log(this.route.snapshot.queryParams);  console.log(this.route.snapshot.fragment);  this.route.queryParams  .subscribe(  (queryParams: Params) => {  this.allowEdit = queryParams['allowEdit'] === '1' ? true : false;  }  );  this.route.fragment.subscribe();    this.server = this.serversService.getServer(1);  this.serverName = this.server.name;  this.serverStatus = this.server.status;  }  onUpdateServer() {  this.serversService.updateServer(this.server.id, {name: this.serverName, status: this.serverStatus});  this.ChangesSaved = true;  this.router.navigate(['../'], {relativeTo: this.route});  }  canDeactive() : Observable<boolean> | Promise<boolean>| boolean {  if(!this.allowEdit){  return true;  }  if((this.serverName != this.server.name || this.serverStatus != this.server.status) && !this.ChangesSaved){  return confirm('Do you want to discard the changes')  }  else{  return true;  }  }  } |

### Passing Static Data to a Route.

**Step 1:-**

In route add the data property to pass message to the particular component

|  |
| --- |
| const appRoutes: Routes = [  { path: '', component: HomeComponent },  { path: 'users', component: UsersComponent, children:[  { path: ':id/:name', component: UserComponent}  ]},    { path: 'servers',  //canActivate:[AuthGuard],  canActivateChild:[AuthGuard],  component: ServersComponent,  children:  [  { path: ':id', component: ServerComponent},  { path: ':id/edit', component: EditServerComponent, canDeactivate:[canDeactiveGuard]}  ]  },  // {path:'notFound', component: PageNotFoundComponent},  {path:'notFound', component:ErrorPageComponent, data:{Message:"Page not found"}},  {path:'\*\*', redirectTo:'/notFound'}  ]; |

**Step 2:-**

Subscribe the message value using ActivatedRoute

|  |
| --- |
| import { ActivatedRoute, Data } from '@angular/router';  import { Component, OnInit } from '@angular/core';  @Component({  selector: 'app-error-page',  templateUrl: './error-page.component.html',  styleUrls: ['./error-page.component.css']  })  export class ErrorPageComponent implements OnInit {  errorMessage: string;  constructor(private route: ActivatedRoute) { }  ngOnInit() {  //this.errorMessage = this.route.snapshot.data['Message'];  this.route.data.subscribe((data:Data)=>{  this.errorMessage = data['Message'];  })  };  } |

### Resolving Dynamic Data with the resolve guard

Resolver is used to run some code before the route is render

Step 1:-

Create a resolver particular to the component then implements the interface resolver

# Observables

### What is Observable?

Observable is used to handle asynchronous task like (User Input) Events, Https Requests etc. This tasks we don’t when it will happends or end. Therefore we need methods to handle this asynchronous task . Observable has major advantages their operators.

All observable need to subscribe.

|  |
| --- |
|  |

### Getting closer to the core of Observables

* + Observables features are not build in JavaScript or TypeScript. Instead it is added by the package name rxjs.
  + So if we see the package.json which lists all its dependency we find the rxjs also.
  + All observable need to un-scribe at the end otherwise it leads to memory leak. But some observable (like params) which is provided by angular no need to unscribe explicitly as angular automatically unscribe it

### Custom Observable

Check video max 172

Not Very often we need to create custom observable.

### Error and Completion

Check video max 173

### Operators

|  |
| --- |
|  |

If we have an observable and observer which offcourse got data we listen to that through subscription that what we learn?

But sometimes we don’t want the raw data but instead we need to filter, transform etc

|  |
| --- |
|  |

All operators of observable need to import from

Import { “operator\_name” } from ‘rxjs/operators’

Eg :- map operator is used suppose we are getting complex data from web server but need to transform before we can use it, then we need map operator.

Pipe -> we can add one or more operator.

Filer operator for filer some data.

### Subjects

Instead of EventEmitter we can use subjects and instead of emit we use next.

We know observable which needs to subscribe, but core idea is something like we need to wrap, callback or event etc.

Subjects are also an object we can subscribe but it is more active because we can activily call next on it from outside. So it is perfect if we use as an active event emitter, which don’t have a passive event source like http request, DOM event.

|  |
| --- |
|  |

Just like own observable we need to unscribe the subject also using onDestroy.

And also subject can use only when cross communication between component through services. But @output we still need to use eventemitter not the subject.

# Authentication and Authorization

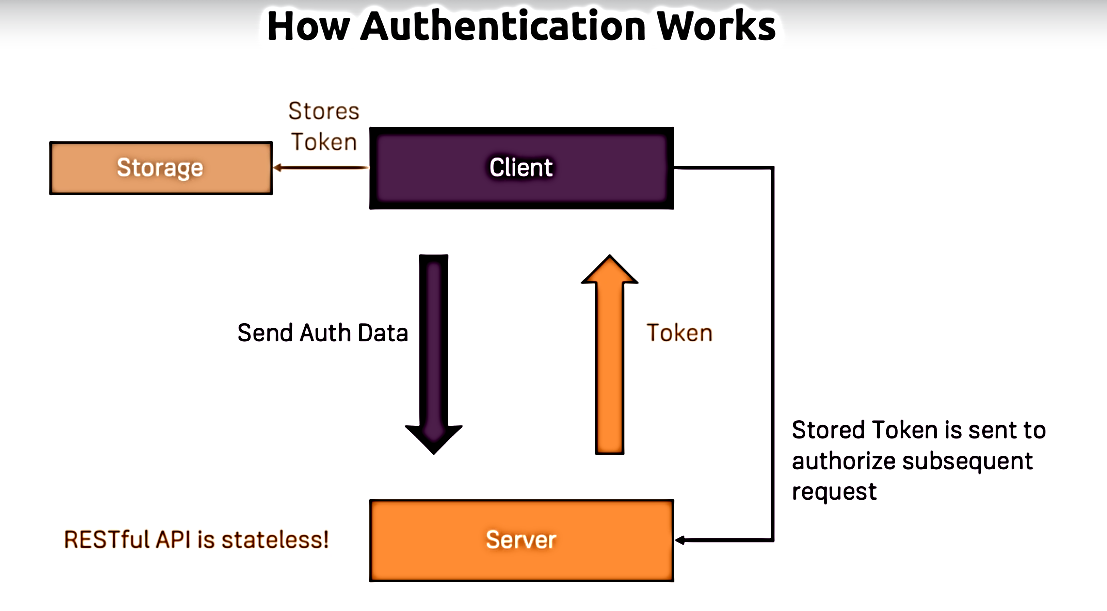
### Introduction (Architecture)

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |



### JSON Web Tokens

<https://jwt.io/>

In JSON web Token

Header: - Red

Payload: Purple

Digital Signature: Blue

|  |
| --- |
|  |

**JWTs have a header, a payload and a digital signature.**

### Implementing Login

Here in the signIn we get the credentials object which is the value behind our form. In the signIn method we call the login method of authService and pass the credentials. This returns an observable, so we subscribe to that if the result is true we navigate the user to home page or we set invalidLogin to true and we this InvalidLogin field to toggle the validation error.

**Login.component**

|  |
| --- |
| import { AuthService } from './../services/auth.service';  import { Component } from '@angular/core';  import { Router } from "@angular/router";  @Component({  selector: 'login',  templateUrl: './login.component.html',  styleUrls: ['./login.component.css']  })  export class LoginComponent {  invalidLogin: boolean;  constructor(  private router: Router,  private authService: AuthService) { }  signIn(credentials) {  this.authService.login(credentials)  .subscribe(result => {  if (result)  this.router.navigate(['/']);  else  this.invalidLogin = true;  });  }  } |

**Login.html**

|  |
| --- |
| <form class="form-signin" #f="ngForm" (ngSubmit)="signIn(f.value)">  <h2 class="form-signin-heading">Please sign in</h2>    <div \*ngIf="invalidLogin" class="alert alert-danger">Invalid username and/or password.</div>  <label for="inputEmail" class="sr-only">Email address</label>  <input type="email" id="inputEmail" name="email" ngModel class="form-control" placeholder="Email address" required autofocus>  <label for="inputPassword" class="sr-only">Password</label>  <input type="password" id="inputPassword" name="password" ngModel class="form-control" placeholder="Password" required>  <button class="btn btn-lg btn-primary btn-block" type="submit">Sign in</button>  </form> |

Now authService.ts login method. Here in the login method we are sending a post request and body of the request we are including the credentials which is the object with two properties email and password.

**As we know this post method returns an observable of response but in our login I don’t to expose the response object, I want to return true or false so we will use map operator to get the response and convert to true or false.**

Here if the credentials is correct from backend response will be send as JSON web token and for incorrect it will send null.

Now get the response and convert to JSON store in a result variable , check the result if it is true and has token we are going to store in **local storage.**

**localStorage.setItem(‘key’,data)**

|  |
| --- |
| import { Injectable } from '@angular/core';  import { Http } from '@angular/http';  import 'rxjs/add/operator/map';  @Injectable()  export class AuthService {  constructor(private http: Http) {  }  login(credentials) {  return this.http.post('/api/authenticate',  JSON.stringify(credentials))  .map(response=>{  let result = response.json();  if(result && result.token){  localStorage.setItem('token',result.token);  return true;  }  else  {  return false;  }  });  }  logout() {  }  isLoggedIn() {  return false;  }  } |

### Implementing Logout

**Step1:-**

Bind the click event to the authservice of the logout method in the template. But in the component of the constructor authservice should inject.

|  |
| --- |
| <h1>Home Page</h1>  <p>  Welcome [NAME]  </p>  <ul>  <li><a routerLink="/admin">Admin</a></li>  <li><a routerLink="/login">Login</a></li>  <li><a (click)="authService.logout()">Logout</a></li>  </ul> |

**Step2:-**

Implement the logout method in the authservice.ts. All we have to do here is to delete the token from the local storage.

|  |
| --- |
| logout() {  localStorage.removeItem('token');  } |

### Showing and Hiding Elements

If we have valid token and this token is not expired that means user is logged in here we should true, otherwise we should false.

In order to work with JSON web token we are going to use one of the libraries called **angular2-jwt**

**npm install angular2-jwt**

**Step1:-**

**Inside the iSLoggened we are going to use class called JwtHelper from angular2-jwt**

**In authService isLoggedIn method.**

|  |
| --- |
| import { JwtHelper } from 'angular2-jwt';    isLoggedIn() {  let jwtHelper = new JwtHelper();  let token = localStorage.getItem('token')  if(!token)  return false;  let isExpired = jwtHelper.isTokenExpired(token);  return !isExpired;  } |

**Instead of the above step of JwtHelper, angular2-jwt provide a inbuilt function called tokenNotExpired()**

|  |
| --- |
| isLoggedIn() {  return tokenNotExpired();  } |

**Step2:-**

Check the isLoggenIn in the template of the component, to show or hide based on the value. Default value of authService.isLoggedIn() is true.

|  |
| --- |
| <h1>Home Page</h1>  <p>  Welcome [NAME]  </p>  <ul>  <li \*ngIf="authService.isLoggedIn()"><a routerLink="/admin">Admin</a></li>  <li \*ngIf="!authService.isLoggedIn()"><a routerLink="/login">Login</a></li>  <li \*ngIf="authService.isLoggedIn()"><a (click)="authService.logout()">Logout</a></li>  </ul> |

### Show or Hiding Based on the User’s Role (Authorization)

**Step1:-**

**In the authService create the method currentUser and get the value**

|  |
| --- |
| get currentUser(){  let token = localStorage.getItem('token');  if(!token) return null;  return new JwtHelper().decodeToken(token);  } |

**Step2:-**

**In the template of the component put the condition based on the currentUSer().**

|  |
| --- |
| <h1>Home Page</h1>  <p>  Welcome [NAME]  </p>  <ul>  <li \*ngIf="authService.isLoggedIn() && authService.currentUser().admin">  <a routerLink="/admin">Admin</a>  </li>  <li \*ngIf="!authService.isLoggedIn()"><a routerLink="/login">Login</a></li>  <li \*ngIf="authService.isLoggedIn()"><a (click)="authService.logout()">Logout</a></li>  </ul> |

### Getting the Current User name

In the template just use the string interpolation

|  |
| --- |
| <h1>Home Page</h1>  <p \*ngIf="authService.isLoggedIn()">  Welcome {{authService.currentUser.name}}  </p>  <ul>  <li \*ngIf="authService.isLoggedIn() && authService.currentUser.admin">  <a routerLink="/admin">Admin</a>  </li>  <li \*ngIf="!authService.isLoggedIn()"><a routerLink="/login">Login</a></li>  <li \*ngIf="authService.isLoggedIn()"><a (click)="authService.logout()">Logout</a></li>  </ul> |

### canActivate Interface

This interface is used to restrict the user to type the url manually to access any template.

We use route guard to protect the user from anonymous user.

**Step1:-**

Create a service auth-guard “ng g s services/auth-guard”

**Step2:-**

Add the Service inside the providers array of the app module, so that we can inject inside the constructor.

**Step3:-**

Implement the CanActivate interface.

**And inside the canActivate we need to check if the user logged in or not, if logged in return true otherwise navigate the user login page and return false.**

In order to check the user is logged in or not, we need to inject authService inside the constructor of the auth-guard component.

|  |
| --- |
| import { Router } from '@angular/router';  import { AuthService } from './auth.service';  import { Injectable } from '@angular/core';  import { CanActivate } from '@angular/router';  @Injectable()  export class AuthGuard implements CanActivate {  constructor(  private authService:AuthService,  private router: Router  ) { }  canActivate(){  if(this.authService.isLoggedIn()) return true;  this.router.navigate(['/login']);  return false;  }  } |

Step4:-

Apply this auth-guard on the admin route/or the route you want to protect inside the app.module

|  |
| --- |
| imports: [  BrowserModule,  FormsModule,  HttpModule,  RouterModule.forRoot([  { path: '', component: HomeComponent },  { path: 'admin', component: AdminComponent, canActivate:[AuthGuard] },  { path: 'login', component: LoginComponent },  { path: 'no-access', component: NoAccessComponent }  ])  ], |

### Redirecting Users after Logging in

Suppose we type a url(http://localhost/admin) in the browser without login, first it will redirect to login page and login we need to redirect to the same url which was type previously. Let see how to do.

**Step1:-**

In the route.navigate() pass the second parameter as **queryparams** and set this to object of **returnUrl and set it to url.state.**

So for getting the we need to pass the **state:RouterStateSnapShot** as second parameter in the canActivate Method.

|  |
| --- |
| import { Router, RouterStateSnapshot } from '@angular/router';  import { AuthService } from './auth.service';  import { Injectable } from '@angular/core';  import { CanActivate } from '@angular/router';  @Injectable()  export class AuthGuard implements CanActivate {  constructor(  private authService:AuthService,  private router: Router  ) { }  canActivate(route,state:RouterStateSnapshot){  if(this.authService.isLoggedIn()) return true;  this.router.navigate(['/login'],{queryParams:{returnUrl:state.url}});  return false;  }  } |

Step2:-

Now in login component get this query parameter. In order to access router parameter we need to inject activatedRoute service into the constructor of the class.

Now in our signIn method when the user successfully logged in instead of navigating to the home page, we have to check the query parameter and if you have returnUrl send the user to that url, otherwise we are going to send them to the home page.

In login.component.ts

|  |
| --- |
| export class LoginComponent {  invalidLogin: boolean;  constructor(  private router: Router,  private authService: AuthService,  private route: ActivatedRoute) { }  signIn(credentials){  this.authService.login(credentials)  .subscribe(result => {  if(result){  //here we no need to subscribe to queryParamMap observable  //instead we can use snapshot.  let returnUrl = this.route.snapshot.queryParamMap.get('returnUrl');  this.router.navigate([ returnUrl || '/']);  }  else  this.invalidLogin= true;  });  } |

### Protecting Routes Based on User’s Role

Now we want to implement the route should be access by admin only.

**Step1:-**

Create a admin-route-guard service “ng g s services/admin-auth-guard”

**Step2:-**

Add the Service inside the providers array of the app module, so that we can inject inside the constructor.

**Step3:-**

Implement the canActivate interface and check if admin or not otherwise redirect to no-access route.

Here also we need to check for current user is present or not

|  |
| --- |
| import { AuthService } from './auth.service';  import { CanActivate, Router } from '@angular/router';  import { Injectable } from '@angular/core';  @Injectable()  export class AdminAuthGuard implements CanActivate {  constructor(  private authService: AuthService,  private router:Router  ) { }  canActivate(){  let user = this.authService.currentUser;  if(user && user.admin) return true;  this.router.navigate(['/no-access']);  return false;  }  } |

Step4:-

Add the admin-auth-guard to the route of admin

|  |
| --- |
| imports: [  BrowserModule,  FormsModule,  HttpModule,  RouterModule.forRoot([  { path: '', component: HomeComponent },  { path: 'admin', component: AdminComponent, canActivate:[AuthGuard,AdminAuthGuard] },  { path: 'login', component: LoginComponent },  { path: 'no-access', component: NoAccessComponent }  ])  ], |

### Accessing Protected API resources

Let see how to add authorization headers to a request in angular.

1. First create the headers object and append with
2. Second create RequestOptions object pass the headers.
3. Third pass the requestOptions as the second argument of get method.

|  |
| --- |
| import { Http, RequestOptions, Headers } from '@angular/http';  import { Injectable } from '@angular/core';  import 'rxjs/add/operator/map';  @Injectable()  export class OrderService {  constructor(private http: Http) {  }  getOrders() {  let headers= new Headers();  let token = localStorage.getItem('token');  headers.append('Authorization','Bearer' + token);    let options = new RequestOptions({headers:headers});    return this.http.get('/api/orders', options)  .map(response => response.json());  }  } |

**Server:-**

So while building API on the server and if you want to secure certain API endpoints, we should there is an authorization headers in the request and value of the bearer a valid JSON web token.

**Client:**

And this means on the client your angular app whenever we want to access protected API we should always supply this above authorization headers.

**As we see writing this in multiple place is repetitive and time consuming. So for that we need inject AuthHttp instead of Http from angular2-jwt and delete the other code.**

|  |
| --- |
| import { AuthHttp } from 'angular2-jwt';  import { Http, RequestOptions, Headers } from '@angular/http';  import { Injectable } from '@angular/core';  import 'rxjs/add/operator/map';  @Injectable()  export class OrderService {  constructor(private http: AuthHttp) {  }  getOrders() {  return this.http.get('/api/orders')  .map(response => response.json());  }  } |

# Unit Testing

### Types of Tests

|  |
| --- |
|  |

|  |
| --- |
|  |

**Unit Tests are:-**

1. Easier to write.
2. Super fast.
3. Don’t give us much confidence about the functionality of our application.

|  |
| --- |
|  |

Classic definition of integration Tests in Angular is testing the component along with the template.

|  |
| --- |
|  |

### Unit Testing Fundamentals

Command for testing is “ng test” which will karma

* For unit test we have to create test file for each component like “compute.spec.ts”
* We use two function most of the time like
  + - * describe() means suite which is group of related test.
      * It() to define spec or test.

# Redux

It is a library that helps you to manage the state of your application. It is something to use in medium to large single page application with complex data flow.

In typical Angular app without Redux we know that each component maintain the state and the logic behind the view. This model allows encapsulation model of object oriented programming.

**However it can be problem when we have multiple views that are working with same piece of data and don’t have parent-child relationship.**

|  |
| --- |
|  |

In this situation we often have multiple copies of the same data that are independent of each other.

|  |
| --- |
|  |

Since this view are independent, if we want to keep then in sync we have to do some extra work to use events sonner or later than will leads to event spargati. The problem with this approach is that data can be updated in an unpredictable way and also adding a new feature becomes a challenge because once again we don’t know what will be the impact of the new feature on the application state.

Facebook has this problem in back 2014. So they introduce the flux architecture. And Redux is the simplified library of this architecture provide and aligement solution to the problem. Manage the application state in a predictable way.

|  |
| --- |
|  |

**Benefits of Redux**

1. **Predictable application state**.
2. **Decoupled architecture** from a presentation framework like angular. So we implement big chunk of your application and its presentation logic using simple function that are completely decoupled from angular or any other presentation framework. So then we can decide we want to use angular or may be you want to use react
3. **Testability**
4. **Great tooling**
5. **Undo/redo**

**When to use Redux?**

1. Independent copies of the same data in multiple places.
2. Multiple views that need to work with same data and be in sync.
3. Data can be updated by multiple users.
4. Data can be updated by multiple actors. (Like it can change as a result of user or at the same time it can arrive from the server).

### Building Blocks

|  |
| --- |
|  |

**Store** – Is a single JS object that contains the state of the application. Think of a local client side database.

If different component needs to work with the same store.

There will be one copy of the same store through out the application. So once the component modify the changes will be immeadility visible to other component.

|  |
| --- |
|  |

**Actions** – Is a plain JS objects that represent something that has happened.

**Reducer** – A function that specifies how the state changes in response to an action. It should be pure function.

### Pure Function

The function is pure if you give the same input; always get the same output no matter how many times we call that function. It should not have any side affects.

|  |
| --- |
|  |

|  |
| --- |
|  |

In Reducer which is a pure function it always takes two arguments, state and action

|  |
| --- |
|  |

**Benefits:-**

1. Easy testability.
2. Easy undo/redo.
3. Time travel debugging.

# Max Course

### ViewChild() and ElementRef

Without two binding if we pass local reference pass to a method or local reference fetch through **ViewChild**()

**ElementRef**(reference to an element) is a type of element, in elementRef has a useful property nativeelemnt property

In ts file

|  |
| --- |
| import { Component, OnInit, EventEmitter, Output, ViewChild, ElementRef } from '@angular/core';  @Component({  selector: 'app-cockpit',  templateUrl: './cockpit.component.html',  styleUrls: ['./cockpit.component.css']  })  export class CockpitComponent implements OnInit {  @Output() serverCreated = new EventEmitter<{serverName:string,serverContent: string}>();  @Output('bpCreated') blueprintCreated = new EventEmitter<{serverName:string,serverContent: string}>();  // newServerName = '';  //newServerContent = '';  @ViewChild('serverContentInput',{static:true}) serverContentInput:ElementRef;  constructor() { }  ngOnInit() {  }  onAddServer(nameInput:HTMLInputElement){  this.serverCreated.emit({  serverName : nameInput.value,  serverContent: this.serverContentInput.nativeElement.value  });    }  onAddBlueprint(nameInput:HTMLInputElement){  this.blueprintCreated.emit({  serverName :nameInput.value,  serverContent: this.serverContentInput.nativeElement.value  });  }    } |

In template

|  |
| --- |
| <div class="row">  <div class="col-xs-12">  <p>Add new Servers or blueprints!</p>  <label>Server Name</label>  <!-- <input type="text" class="form-control" [(ngModel)]="newServerName">-->  <input type="text"  class="form-control"  #serverNameInput>  <label>Server Content</label>  <!--<input type="text" class="form-control" [(ngModel)]="newServerContent">-->  <input type="text"  class="form-control"  #serverContentInput>  <br>  <button  class="btn btn-primary"  (click)="onAddServer(serverNameInput)">Add Server</button>  <button  class="btn btn-primary"  (click)="onAddBlueprint(serverNameInput)">Add Server Blueprint</button>  </div>  </div> |

### @contentChild()

To get the content which is store in another component but then pass on via <ngContent>

### Understanding the component Lifecycle

|  |
| --- |
|  |

### Using Renderer to build a better attribute directive

|  |
| --- |
| import { ElementRef } from '@angular/core';  import { Directive, Renderer2, OnInit } from '@angular/core';  @Directive({  selector: '[appBetterHighlight]'  })  export class BetterHighlightDirective implements OnInit {  constructor(private eleRef: ElementRef, private renderer: Renderer2) { }  ngOnInit(){  this.renderer.setStyle(this.eleRef.nativeElement,'background-color','blue');  }  } |

### HostListener

With HostListener we can react to user event or any event.

|  |
| --- |
| import { ElementRef, HostListener } from '@angular/core';  import { Directive, Renderer2, OnInit } from '@angular/core';  @Directive({  selector: '[appBetterHighlight]'  })  export class BetterHighlightDirective implements OnInit {  constructor(private eleRef: ElementRef, private renderer: Renderer2) { }  ngOnInit(){  // this.renderer.setStyle(this.eleRef.nativeElement,'background-color','blue');  }  //mouseenter is one of the event supported by DOM element  //that directive seats on.  @HostListener('mouseenter') mouseover(eventData:Event){  this.renderer.setStyle(this.eleRef.nativeElement,'background-color','blue');  }    @HostListener('mouseleave') mouseleave(eventData:Event){  this.renderer.setStyle(this.eleRef.nativeElement,'background-color','transparent');  }  } |

### HostBinding

Check the video of max 95

Instead of Renderer we can use HostBinding.

Through HostListener and HostBinding is a great way for working with the element inside the directive.

### Injectable

We need this Injectable metadata when we call service inside another service, otherwise it will give error.

And also we need to add the service in the providers array of appModule.

|  |
| --- |
| import { LoggingService } from "./logging.service";  import { Injectable } from "@angular/core";  @Injectable()  export class AccountService {  accounts = [  {  name: 'Master Account',  status: 'active'  },  {  name: 'Testaccount',  status: 'inactive'  },  {  name: 'Hidden Account',  status: 'unknown'  }  ];  constructor(private loggingService: LoggingService){}    addAccount(name: string, status: string){  this.accounts.push({name: name, status: status });  this.loggingService.logStatusChange(status);  }  updateStatus(id: number, status: string){  this.accounts[id].status = status;  this.loggingService.logStatusChange(status);  }  } |

# Naming Convention, Short cut Key, Extra in TypeScript s

### Class Name for component

It should be in pascal e.g. class CoursesComponent

### Short cut key

1. **Control + p** will able to enter the file name
2. ul>li will generate the ul and l complete
3. Short of below div class is “div.panel.panel-default“ and press tab.

<div class="panel panel-default"></div>

div.panel.panel-default>div.panel.panel-heading+div.panel-body

And press tab will give

|  |
| --- |
| <div class="panel panel-default">  <div class="panel panel-heading"></div>  <div class="panel-body"></div>  </div> |

1. Short of below

<li><a href=""></a></li>

<li><a href=""></a></li>

Is

(li>a)\*2

1. Short of below

<form action="">

<label for=""></label><input type="text" class="form-control">

</form>

Is

form>label+input[type='text'].form-control

1. Short of below is

<div class="checkbox">

<label for="">

<input type="checkbox" name="" id="">

</label>

</div>

is

div.checkbox>label>input['checkbox']

### Extra

* + **DOM Vs HTML**

DOM is a model of object that represents the structure of a document; it’s essentially a tree of object in memory.

HTML is a markup language that we use to represent DOM in text. So when a browser parses a HTML document it creates a tree of object in memory that we refer to as a DOM.

We can also create tree of object programmatically using vanilla JavaScript.

* + **Shadow DOM**

It basically a specification to enable DOM tree encapsulation. **Allows us to apply scoped styles to elements without bleeding out to the outer world.**

**Sample Javascript Code:**

**Eg:**

var el = document.querySelector(‘favorite’);

el.innerHTML = `

<style> h1 {color: red} </style>

<h1>Hello<h1>

`;

**Here in the above it will leak the color to red to other h1 element also.**

**So for that we need create createShowRoot and instead of inner html use inner html of shaow root**

Eg:

var el = document.querySelector(‘favorite’);

**var root = el.createShadowRoot();**

**root**.innerHTML = `

<style> h1 {color: red} </style>

<h1>Hello<h1>

`;

### BootStrap Class

* **form-control** – that gives our input field nice modern look and feel.
* **form-group** – is used in div inside the div we will put the label and textbox
* **btn btn-primary –** is used in button class which is give some blue color button.

|  |
| --- |
| <form action="">  <div class="form-group">  <label for="firstName">First Name</label>  <input id="firstName" type="text" class="form-control">  </div>  <button class="btn btn-primary">Submit</button>  </form> |

* **alert alert-danger –** is used in validation error message which will put in red box.
* **checkbox -** checkbox in bootstrap

<div class="checkbox">

<label for="">

<input type="checkbox" name="" id="">

</label>

</div>

* **radio –** radio button in bootstrap

<div class="radio">

<label>

<input type="radio">

</label>

</div>

* **list-group, list-group-item –** used from <ul> and <li>

<li class="list-group">

<ul class="list-group-item"></ul>

</li>