**MoshProject:-**

I love modern applications have at least two parts a front end and backend the front end also called

the client is the part that runs in the web browser.And this is what the user sees and interacts with.

**Node js**

It's basically a runtime environment for executing javascript code outside the browser. Once you have node then you're going to use a tool called node package manager or NPM to install third

party libraries.

**Folders**

E-2 E which stands for end to end. And this is where we write end to end tests for our application. user so we can write code to launch our browser navigate to the page of our application click a few

links here and there fill out a form.Click a button and then assert that there is something on the page.

This is an example of an end to end test.

node modules and this is where you store all the third party libraries that ourapplication may depend upon.

source folder and this is where we have the actual source code of our application

app folder here and inside this folder we have a module and a component.

So every application has at least one module and one component

assets folder where you store the static assets of our application.

environments folder and this is where we store configuration settings for different environments.

So we have one file for the production environment and the other for the development environment.

main file which is a typescript file and this is basically the startingpoint of our application.

So all you're doing here is bootstrapping the main module of our application which is in this case app

module.

So angular loads this module and everything else starts from there.

poly Phills file which basically imports some scripts that are required for running angular

because the angler framework uses features of javascript that are not available in the current version

of javascript supported by most browsers out there.

So this poly fills fill the gap between the features of javascript that angular needs and the features

supported by the current browsers.

We know that we have styles that CSSA and this is where we add the global Stiers for our application

We know that we have test TS which is basically used for setting up more testing environments.

Anularcli.json

We load out we have Editor config.

So if you're working in a team environment you want to make sure that all developers in the team use

the same settings in their editors.

So this is where you store your settings.

You have this good ignore file which is basically for excluding certain files and folders from your

good repository.

If you have not worked with get.

Don't worry it's not something that you need to know in order to build applications with Engler.

It's basically a tool for managing and versioning your source code that believe that we have Carmo that

configure G-S which is a configuration file for Carmo which is a test runner or javascript code.

Again we're not going to worry about running tests at this stage.

Now below that we have an important file package that Jason.

Now this is a standard file that every node project has apart from a bunch of basic settings here like

the name and version of your application.

We have this setting here dependencies which determines the libraries that your application is dependent

upon.

So here you can see we have nine references to angular libraries.

All these libraries start with as angular.

And after that we have the name of the library.

So the first library here is for animations.

If you're not going to use animations in your application you can delete this year.

So in the future as we use third party libraries you will see them listed here under the dependencies

below that we also have another setting or another key called dev dependencies and these are the libraries

that we need in order to develop this application.

So we don't need these to run our application on production server.

These are purely for a developer machine.

So here we have a reference to angular CLIA which you're only familiar with.

We also have few other references to Carmo which is again a test runner with javascript code.

Now below this package Jason we have protractor config that J.S. which is basically a tool for running

and to tests for angular.

Again we're not going to worry about it.

The other file here is TS config which has a bunch of settings for your typescript compiler so the typescript

compiler looks at the settings and based on the settings is going to compile your typescript code into

javascript browsers can't understand.

Again for the most part you don't have to change any of this here.

Just be aware that if in the future in a complex project you need to change your typescript compiler

settings.

This is where you need to apply your changes.

And finally we have TS lint that Jason which includes a number of settings for TS linked.

In case you don't know slainte is a static analysis tool for typescript code so it checks your typescript

code for readability maintainability and functionality errors.

So this is the basic structure of an angular project.

**Webpack**

Anglo Seelye uses a tool called where Pike which is a builautomation tool.

It gets all our scripts and stylesheets combines them with them in a bundle and then modifies that bundle.And this is for optimization. Now this right click on this page and go to view page source on the bottom of this page.Look at these script tags.

All the bundles that were generated it also injected them into our index that HMO.

T.s

We havethe concept of Classes, Interfaces, Constructors,Access Modifiers, like public and private, Fills, Properties,Generics and so on TypeScript we catch errors at Compile-time, instead of Run-time.

Var is function scoped let is block scoped

Let count=5;

Count=”angular” error

Let a; means it is any

a=1,

a=true,

a=”a” all are true

Now we also have another type that I absolutely

love, and that's enum, so let's say you're working with a group

of related constants, like colors, so, in plain

and old, or vanilla JavaScript, we would define constant

colors like this, so color red, we can set this to zero.

Constant, ColorGreen, we set this to one,

constant, ColorBlue, set it to two.  so we can put all these related

constants in a container. So in TypeScript, we can

can declare an enum like this, enum, all lower case

**Interface:**

reuse this in multiple places letter of every word in the name of the interface should be capitalized.

**Classes:**

groups properties and functions that are highly related.

**Object:-**

An object is an instance of a class.

**Constructor**

oriented program languages, we have this concept called

Constructor. So every class can have a Constructor, which is basically

a method that is called when we create an instance of

that class.

**AccessModifiers:-**

An Access Modifier, is basically a keyword we can apply to a member of a

class to control it's access from the outside.

So in TypeScript, we have three Access Modifiers, public, private,

and protected. Public and private are the most common, and

by default, all members are public. Let me show you what I mean.

**Components**

Component encapsulates the data, the HTML markup,

and the logic for a view, which is the area of the screen the use

**Module:**

A Module, is a container for a group of related

Components. Every Angular app has at least one module which we call App Module

**Dependency Injection**

So Dependency Injection means injecting or providing the dependencies on class into its constructor.

Now angular as a dependency injection framework built into it.

So when it's going to create an instance of a component it can inject that dependencies.

**Displaying data and Handling Events:-**

**Property Binding:-**

to bind the text content property of h2, DOM element. Have to bind this to the title field

<!-- attr attribute

     DOM is a model of objects that represent the structure of a document

we have HTML attributes that don't have a representation in the DOM.Here,

colspan is an example of that. So when we parse

this HTML markup, and create an actual DOM object for this

td, that DOM object does not have a property called colspan,

and that's why we get this error. So colspan, -->

<!-- <table>

    <tr>

        <td [attr.colspan] ="colSpan"  >{{title}}</td>

    </tr>

</table> -->

**<p [textContent]=’title’>**

 class binding:-

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

  isActive:boolean=true;

**StyleBinding:-**

<button class="btn btn-primary"  [style.backgroundColor]="isActive ? 'blue':'green'"   >Click me</button>

  isActive:boolean=true;

**Event Filtering:-**

let's say we want tosubmit this form only if the userpresses enter. So this is the traditional way of implementingthis. If dollar event.keyCode= 13 which represents enter, and we can do a console log

<input (keyup)="onKeyUp($event)">

 onKeyUp(e)

   {

if(e.keyCode ==13){

  alert();

}

   }

**Or**

<input (keyup.enter)="onKeyUp()">

 onKeyUp()

   {

   alert();

   }

**Template Variable**

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

 onKeyUp(value)

  {

   console.log(value);

  }

**Pipes**

 We use pipes to format data.

course ={

  title:"the complete Angular course",

  rating:4.9745,

  students:30123,

  price:190.95,

  releaseDate:new Date(2016,3,1)

}

{{course.title | uppercase}} <br>

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

{{course.students | number}} <br>

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

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

<!-- number of students. Here we have 5 digits, but to make it more readable.

we can apply the decimal pipe to separate every 3

digits using a comma. So here I'm going to

apply the decimal pipe. Now the keyword for the decimal -->

<!-- we have control of the number of integer digits,

as well as the number of digits after the decimal point,

So here we have applied the number pipe, or the decimal

pipe to the rating property -->

**Custom Pipes**

**Create summary.pipe.ts**

we need to import a couple of types from Angular. In summary pipe

One is the Pipe decorator function,

and the other is PipeTransform, which is an

interface that defines the shapes of all pipes in Angular.

Now we need to import these from angular

**App.ts**

text="Virat Kohli is an Indian cricketer. He was born in Delhi, India on November 5, 1988. Virat is the first player in ICC cricket history to win all 3 ICC awards in a single year- ICC ODI player of the year, ICC Test player of the year and ICC Player of the year award in 2018. He is rated as one of the best batsmen in the world.";

**app.html**

{{text | summary}}

Summary.pipe.ts

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

@Pipe ({

    name:'summary'

})

export class SummaryPipe implements PipeTransform

{

    transform(value:string , args?:any)

    {

              if(!value)

              return null;

           return   value.substr(0,50) +'.................'

    }

}

{{text | summary:10}}

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

@Pipe ({

    name:'summary'

})

export class SummaryPipe implements PipeTransform

{

    transform(value:string , limit:number)

    {

              if(!value)

              return null;

              let actualLimit = (limit)  ? limit :50

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

    }

}

**Favourite component:-**

<span

  class="glyphicon"

  [class.glyphicon-star]="isFavorite"

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

  (click)="this.isFavorite = !this.isFavorite "

  ></span>

**Titlecase:**

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

@Pipe ({

    name:'summary'

})

export class SummaryPipe implements PipeTransform

{

    transform(value: string): any {

        if (!value) return null;

        let words = value.split(' ');

        for (var i = 0; i < words.length; i++) {

          let word = words[i];

          if (i > 0 && this.isPreposition(word))

            words[i] = word.toLowerCase();

          else

            words[i] = this.toTitleCase(word);

        }

        return words.join(' ');

      }

      private toTitleCase(word: string): string {

        return word.substr(0, 1).toUpperCase() + word.substr(1).toLowerCase();

      }

      private isPreposition(word: string): boolean {

        let prepositions = [

          'of',

          'the'

        ];

        return prepositions.includes(word.toLowerCase());

      }

}

**View Encapsulation:-**

**ngContent:-**

we can provide custom content to our reusable components.

**Input and Output:**

**Like.html**

<span

  class="glyphicon"

  [ngClass]="{

    'glyphicon-star': !isSelected,

    'glyphicon-star-empty': isSelected

  }"

  (click)="onClick()"

  ></span>

**Like.html**

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

@Component({

  selector: 'app-like',

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

  styleUrls: ['./like.component.css']

})

export class LikeComponent implements OnInit {

  @Input('isFavorite') isSelected: boolean;

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

  onClick() {

    this.isSelected = !this.isSelected;

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

  }

  ngOnInit()

  {

  }

}

export interface FavoriteChangedEventArgs{

  newValue:boolean

}

**App.html**

 <app-like   [isFavorite]="post.isFavorite" (change)="onFavoritechaned($event)"   ></app-like>

**App.ts**

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

import {FavoriteChangedEventArgs} from  './like/like.component';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

  post={

    title:"title",

    isFavorite: true

  }

  ngOnInit()

  {

  }

  onFavoritechaned(eventArgs:FavoriteChangedEventArgs)

  {

    console.log(eventArgs);

  }

}

**Ng-content**

**Panel.compoent.ts**

<div class="panel panel-default">

    <div class="panel-heading">

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

    </div>

    <div class="panel-body">

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

    </div>

  </div>

**app.html:-**

<panel>

     <div class="heading">

       Heading

     </div>

     <div class="body">

      <h2>Body</h2>

       <p>Some content here...</p>

     </div>

 </panel>

**Ng-container:**

**Panel.html**

<div class="panel panel-default">

    <div class="panel-heading">

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

    </div>

    <div class="panel-body">

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

    </div>

  </div>

**App.html**

<panel>

      <ng-container class="heading">

       Heading

    </ng-container>

    <ng-container class="body">

      <h2>Body</h2>

       <p>Some content here...</p>

    </ng-container>

 </panel>

**LikesTask:-**

**Like.hml**

 <span class="glyphicon glyphicon-heart"

 [class.highlighted]="isActive" (click)="onClick()"

 >

 </span>

 {{likesCount}}

**Like.ts**

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

@Component({

  selector: 'app-like',

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

  styleUrls: ['./like.component.css']

})

export class LikeComponent implements OnInit {

  @Input('likesCount') likesCount: number;

  @Input('isActive') isActive: boolean;

  ngOnInit()

  {

  }

  onClick()

  {

   this.likesCount += (this.isActive) ? 1 :-1;

   this.isActive =!this.isActive;

  }

}

**App.html**

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

**App.ts**

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

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

  tweet ={

    body:'.....',

    likesCount:10,

    isLiked:true

  }

  ngOnInit()

  {

  }

}

**ngIf:**

**app.html**

 <div   \*ngIf="courses.length >0; then courseslist else nocourses">

 </div>

<ng-template #courseslist>

    List of courses

</ng-template>

 <ng-template #nocourses>

    No Courses yet

 </ng-template>

**App.ts**

 courses=[1,2];

**ngSwitch:**

**app.html**

 <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>Otherwise</div>

  </div>

**App.ts**

viewMode="map"

**ngFor:**

  <ul>

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

       {{i}} - {{course.name}}  <span \*ngIf="isEven">(Even)</span><span \*ngIf="!isEven">(Odd)</span>

      </li>

  </ul>

 courses=[

{id:1, name:'course1'},

{id:2, name:'course2'},

{id:3, name:'course3'}

 ];

**Small crud**

**App.html**

  <button (click)="onAdd()">Add</button>

  <ul>

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

        {{course.name}}

        <button (click)="onremove(course)" >Remove</button>

      </li>

  </ul>

onremove(course)

  {

   console.log(course);

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

   this.courses.splice(index, 1);

  }

**Ngfor and trackby**

**ngClass:-**

 <button

 [ngStyle]="{

'background-color': canSave ? 'blue' :'gray',

'color':canSave ? 'white' :'black'

 }"

 >Save</button>

canSave=true;

**safe transversal operator:-**

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

task ={

  title:"review applications",

  asignee : null

}

**Creating custom directive:**

**ng g d input-format**

First on the top we need to import a decorator function,

HostListener, from angular/core.

This decorator allows us to subscribe to the event,

erased from the DOM element, that is the DOM element that is hosting

this directive, or in other words, that's the DOM element that has this

attribute. So let's see how we can use host listener.First I'm going to define a method here,onFocus. So whenever we get thfocus in our input field, we want this method to be caught. For nowI just wantto log something in the console, onFocus.Now we need to decorate this method,with the host listener . ElementRef. This is a service defined in Angularthat gives us access to the DOM object.So, let's import this on the top. You can see this isdefined in angular/core

**input-format.directive.ts:-**

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

@Directive({

  selector: '[appInputFormat]'

})

export class InputFormatDirective {

  @Input ('appInputFormat') format;

  constructor(private el:ElementRef) { }

   @HostListener ('blur')  onBlur()

   {

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

     if(this.format  == "lowercase")

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

     else

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

   }

}

**App.html**

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

**Zippy:-**

**zippy.css**

.zippy {

    border: 1px solid #ccc;

    border-radius: 2px;

}

.zippy-heading {

    font-weight: bold;

    padding: 20px;

    cursor: pointer;

}

.zippy-body {

    padding: 20px;

}

.expanded {

    background: #f0f0f0;

}

.glyphicon {

    float: right;

}

**Zippy.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>

**Zippy.ts**

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

@Component({

  selector: 'zippy',

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

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

})

export class PanelComponent implements OnInit {

  @Input('title') title: string;

  isExpanded: boolean;

  toggle() {

    this.isExpanded = !this.isExpanded;

  }

  constructor() { }

  ngOnInit(): void {

  }

}

**App.html**

<zippy title="shipping details">

   shipping details content

</zippy>

**Template Driven Forms:-**

**Building a bootstrap form:-**

<div class="container">

<form>

  <div class="form-group">

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

    <input id="firstname" type="text" class="form-control">

  </div>

  <div class="form-group">

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

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

  </div>

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

</form>

</div>

with template driven approach, we apply a directive toour input field, and Angular will create a control object and associated with that input field

because anytime we apply this ngModel onan input field, and it needs a way to distinguish thesecontrol objects. So here we set the name attributeSo I want to handle the change

event of this input field, and here we're going to call

the log method, now I need a reference to this ng

Model directive, I want to pass that model to the log method, so we can

log it on the console. For that we're going to create a template

variable. So, hashtag, let's call the template variable

firstName. You could call it anything. And as the value

I'm going to use ngModel. So when Angular sees

this, it's going to set this template variable to the ng

Model directive that is applied on this input field. Okay?

And then, you can pass this variable to our log method.

So, here's the log method, it has, first

Name. Now, let's go and implement this method. So back

in our component, I'm going to delete the stuff that we don't need,

let's just keep it simple, here's the log method.

Let's give it a parameter called x, and simply log it on the console,

this is just for a diagnostics. Now back in the browser,

I'm going to type my name here, press tab,

let's take a look at the console. Now here's our ng

Model. So you can see ngModel is an object.

with these properties. And look at these properties, control. This

is the control object that I was telling you about. Now if you click on the value.

Look, it's an instance of the form control class

in Angular. Let's expand this.

Here we have a bunch of properties and many of these properties come

in pairs. For example, we have this dirty, which determines

if the value of the input field is changed from the moment

the form was initialized. In this case, because I type my name here.

Dirty is true, now the opposite property,

is pristine. Pristine is clean. So,

dirty should be true, and pristine is false.

Okay? Now we have another pair. Invalid, and

valid, in this case because we haven't implemented validation yet,

this input field is considered valid, so in

invalid should be false, and valid should be true.

Now if you have any validation errors, they are available

here. Currently errors is null because we don't have

any errors. We have another pair which is touched and

untouched. In this case, because I touched the input field, or in other

words, I put the focus there and then moved away, touched

should be true, and untouched should be false.

We have another important property value, which returns the

current value in the input field. So what I want to note here is

that we use this form control class, to track state changes.

And the validity of input fields. When we

apply the ngModel directive, along with the name attribute

input field, Angular automatically creates an instance of

the form control class, and associates it with that field.

Now to finish up this lecture I'm going to apply ngModel,

on our text area as well, and set the name attribute

to comment. In the next lecture, we're going to add some validation

here.

**ngModel**

<div class="container">

<form>

  <div class="form-group">

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

    <input id="firstname"  ngModel name="firstName" (change)="log(firstName)"

     #firstName="ngModel" type="text" class="form-control">

  </div>

  <div class="form-group">

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

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

  </div>

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

</form>

</div>

log(x) { console.log(x); }

**Adding validation:-**

<div class="container">

<form>

  <div class="form-group">

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

    <input id="firstname"  ngModel name="firstName" (change)="log(firstName)"

     #firstName="ngModel" type="text" class="form-control" required>

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

  </div>

  <div class="form-group">

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

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

  </div>

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

</form>

</div>

**you want to set a minimumlengths for our first name**

**specific validation errors:-**

 <div class="form-group">

    <label for="firstName">First Name</label>

    <input

      required

      minlength="3"

      maxlength="10"

      pattern="banana"

      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 minimum {{ firstName.errors.minlength.requiredLength }} characters.

      </div>

      <div \*ngIf="firstName.errors.pattern">

        First name doesn't match the pattern.

      </div>

    </div>

  </div>

**Styling invalid input fields:**

.form-control.ng-touched.ng-invalid {

    border: 2px solid red;

}

**Final**

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

  <div class="form-group">

    <label for="firstName">First Name</label>

    <input

      required

      minlength="3"

      maxlength="10"

      pattern="banana"

      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 minimum {{ firstName.errors.minlength.requiredLength }} characters.

      </div>

      <div \*ngIf="firstName.errors.pattern">

        First name doesn't match the pattern.

      </div>

    </div>

  </div>

  <div class="form-group">

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

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

  </div>

  <div class="checkbox">

    <label>

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

    </label>

  </div>

  <div class="form-group">

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

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

      <option value=""></option>

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

    </select>

  </div>

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

    <label>

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

      {{ method.name }}

    </label>

  </div>

  <p>

    {{ f.value | json }}

  </p>

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

</form>

 contactMethods = [

    { id: 1, name: 'Email' },

    { id: 2, name: 'Phone' },

  ];

  log(x) { console.log(x); }

  submit(f) {

  }

**Reactive Forms:-**

**Basic form:-**

 <div class="container">

   <form>

     <div class="form-group">

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

       <input

        type="text"

        id="username"

        class="form-control">

     </div>

     <div class="form-group">

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

       <input

       type="password"

       id="password"

       class="form-control">

     </div>

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

   </form>

 </div>

**Creating controls programmatically:-**

Form-group It has key is a string, and the value is an AbstractControl. AbstractControl is the base class for FormControl andFormGroup. All the properties that are common between these two classes

are actually defined in the AbstractControl class. So these classes

simply inherent these properties from their parent. as the first argument here, we need to pass an object, and these objects

should have one or more key value pairs, keys should be string

and values should be AbstractControl objects.

So we add an object, here we need two key value pairs.

One for the username, and one for password. So,

App.html

 <div class="container">

   <form  [formGroup]="form">

     <div class="form-group">

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

       <input

       formControlName="username"

        type="text"

        id="username"

        class="form-control">

     </div>

     <div class="form-group">

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

       <input

       formControlName="password"

       type="password"

       id="password"

       class="form-control">

     </div>

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

   </form>

 </div>

Ts

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

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

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

title:"welcome to Angualr"

form =new FormGroup({

  username:new FormControl(),

  password:new FormControl()

});

  ngOnInit()

  {

  }

}

**Adding Validation:-**

  <div class="container">

   <form  [formGroup]="form">

     <div class="form-group">

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

       <input

       formControlName="username"

        type="text"

        id="username"

        class="form-control">

        <div \*ngIf="username.touched && username.invalid" class="alert alert-danger">Username is required</div>

     </div>

     <div class="form-group">

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

       <input

       formControlName="password"

       type="password"

       id="password"

       class="form-control">

     </div>

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

   </form>

 </div>

**Ts**

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

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

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

title:"welcome to Angualr"

form =new FormGroup({

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

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

});

  ngOnInit()

  {

  }

  get username()

  {

    return this.form.get('username')

  }

}

**Specific validation errors:-**

the second

parameter of the constructor of the FormControl class requires

either a validator function, or an array of validator

functions. So here, if you want to have multiple validators,

you can add them in an array

 <div class="container">

   <form  [formGroup]="form">

     <div class="form-group">

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

       <input

       formControlName="username"

        type="text"

        id="username"

        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>

     <div class="form-group">

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

       <input

       formControlName="password"

       type="password"

       id="password"

       class="form-control">

     </div>

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

   </form>

 </div>

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

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

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

title:"welcome to Angualr"

form =new FormGroup({

  username:new FormControl('',[Validators.required,Validators.minLength(3)]),

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

});

  ngOnInit()

  {

  }

  get username()

  {

    return this.form.get('username')

  }

}

**Implementin custom validation:-**

**Create username.validators.ts**

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;

    }

}

**App.ts**

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

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

import { UsernameValidators } from './username.validators';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

title:"welcome to Angualr"

form =new FormGroup({

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

                              Validators.minLength(3),

                              UsernameValidators.cannotContainSpace

                            ]),

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

});

  ngOnInit()

  {

  }

  get username()

  {

    return this.form.get('username')

  }

}

**App.html**

 <div class="container">

   <form  [formGroup]="form">

     <div class="form-group">

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

       <input

       formControlName="username"

        type="text"

        id="username"

        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>

     <div class="form-group">

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

       <input

       formControlName="password"

       type="password"

       id="password"

       class="form-control">

     </div>

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

   </form>

 </div>

**Asynchronous operations:-**

 What is an asynchronous operation?

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;

    }

    static shouldBeUnique(control:AbstractControl) : ValidationErrors | null

    {

        setTimeout(() => {

            if(control.value == 'mosh')

            return {shouldBeUnique: true};

            return null;

        }, 2000);

            return null;

    }

}

**Asynchronous validators:-**

**Username.validator.ts**

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

import { promise } from 'protractor';

import { rejects } from 'assert';

export class UsernameValidators{

 static   cannotContainSpace(control:AbstractControl):ValidationErrors | null

    {

        if((control.value as string).indexOf(' ') >=0)

        return {cannotContainSpace:true}

        return null;

    }

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

    {

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

            setTimeout(() => {

                if(control.value == 'mosh')

                resolve( {shouldBeUnique: true});

             else

               resolve (null);

            }, 2000);

        });

    }

}

**App.ts**

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

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

import { UsernameValidators } from './username.validators';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

title:"welcome to Angualr"

form =new FormGroup({

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

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

});

  ngOnInit()

  {

  }

  get username()

  {

    return this.form.get('username')

  }

}

**App.html**

 <div class="container">

   <form  [formGroup]="form">

     <div class="form-group">

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

       <input

       formControlName="username"

        type="text"

        id="username"

        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"

       type="password"

       id="password"

       class="form-control">

     </div>

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

   </form>

 </div>

**Showing loader image:-**

 <div class="container">

   <form  [formGroup]="form">

     <div class="form-group">

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

       <input

       formControlName="username"

        type="text"

        id="username"

        class="form-control">

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

        <div \*ngIf="username.atouched && 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"

       type="password"

       id="password"

       class="form-control">

     </div>

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

   </form>

 </div>

**Validating form input upon form submit**

**App.html**

 <div class="container">

   <form  [formGroup]="form"  (ngSubmit)="login()" >

     <div \*ngIf="form.errors"  class="alert alert-danger">

       The username or password is inalid

     </div>

     <div class="form-group">

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

       <input

       formControlName="username"

        type="text"

        id="username"

        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"

       type="password"

       id="password"

       class="form-control">

     </div>

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

   </form>

 </div>

**App.ts**

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

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

import { UsernameValidators } from './username.validators';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

title:"welcome to Angualr"

form =new FormGroup({

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

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

});

  ngOnInit()

  {

  }

  get username()

  {

    return this.form.get('username')

  }

  login()

  {

    this.form.setErrors({

      invalidLoin:true

    })

    // let isValid= authService.login(this.form.value);

    // if(!isValid) {

    //   this.form.setErrors({

    //     invalidLoin:true

    //   })

    }

  }

**Nestedformgroups:**

 <div class="container">

   <form  [formGroup]="form"  (ngSubmit)="login()" >

     <div formGroupName="account">

     <div \*ngIf="form.errors"  class="alert alert-danger">

       The username or password is inalid

     </div>

     <div class="form-group">

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

       <input

       formControlName="username"

        type="text"

        id="username"

        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"

       type="password"

       id="password"

       class="form-control">

     </div>

    </div>

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

   </form>

 </div>

**App.ts**

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

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

import { UsernameValidators } from './username.validators';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

title:"welcome to Angualr"

form =new FormGroup({

  account: new FormGroup({

    username:new FormControl(''  ),

    password:new FormControl('' )

  })

});

  ngOnInit()

  {

  }

  get username()

  {

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

  }

  login()

  {

    this.form.setErrors({

      invalidLoin:true

    })

    // let isValid= authService.login(this.form.value);

    // if(!isValid) {

    //   this.form.setErrors({

    //     invalidLoin:true

    //   })

    }

  }

**FormArray:-**

**App.html**

<form>

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

  <ul>

    <li \*ngFor="let topic of  Topics.controls"   (click)="removeTopic(topic)"  class="form-control">

      {{ topic.value }}

    </li>

  </ul>

</form>

**App.ts**

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

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

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent   {

title:"welcome to Angualr"

form =new FormGroup({

  topics:new FormArray([])

})

addTopic(topic:HTMLInputElement)

{

 this.Topics.push(new FormControl(topic.value));

topic.value="";

}

get Topics()

{

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

}

removeTopic(topic:FormControl)

{

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

this.Topics.removeAt(index);

}

// form;

// constructor(fb: FormBuilder){

//   this.form = fb.group({

//     name: ['', Validators.required],

//     contact: fb.group({

//       email: [],

//       phone: []

//     }),

//     topics: fb.array([])

//   });

}

**Angular Material2:-**

Angular.material.io visit this site

npm i --save @angular/cdk @angular/material @angular/animations hammerjs

Routin**g:-**

import { NavbarComponent } from './navbar/navbar.component';

import { HomeComponent } from './home/home.component';

import { GithubprofileComponent } from './githubprofile/githubprofile.component';

import { NotfoundComponent } from './notfound/notfound.component';

import { GithubfollowersComponent } from './githubfollowers/githubfollowers.component';

import { PostsComponent } from './posts/posts.component';

export const routes:Routes=[

{path:'',component:HomeComponent},

{path:'followers',component:GithubfollowersComponent},

{path:'followers/:username',component:GithubprofileComponent},

{path:'posts',component:PostsComponent},

{path:'\*\*',component:NotfoundComponent}

]

**Getting route parameters:**

**route.paramMap.** That's the property that gives us all the parameters. We use getAll to get the value of all route parameters. Get method returns a string. If you need to work with a number,

a technique that we use in JavaScript is to put a plusbefore this. This will convert this string to a number.

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

import {Params,ActivatedRoute} from '@angular/router';

@Component({

  selector: 'app-githubprofile',

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

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

})

export class GithubprofileComponent implements OnInit {

   id:number ;

  constructor(private route:ActivatedRoute) { }

  ngOnInit()  {

     this.route.paramMap

     .subscribe(params => {

      this.id=+params.get('id')

     console.log(  this.id );

     })

  }

}

Why route parameters are observables:

**Query Parameters:**

how to add optional parameters to your routes.

Navbar.html

 <nav class="navbar navbar-default">

    <div class="container-fluid">

      <div class="collapse navbar-collapse">

        <ul class="nav navbar-nav">

          <li routerLinkActive="active" ><a [routerLink]="['followers']"

            [queryParams]="{page:1, order:'newest'}"

            >Followers</a></li>

          <li  routerLinkActive="active" ><a  [routerLink]="['posts']">Posts</a></li>

        </ul>

      </div>

    </div>

  </nav>

 ngOnInit(): void {

     this.route.paramMap.subscribe(params =>{

     })

    // let id= this.route.snapshot.paramMap.get("id");

   this.route.queryParamMap.subscribe(params =>{

   });

//  let page= this.route.snapshot.queryParamMap.get("page");

how can we get both the required and optional

query parameters, and then call the server to get the list of followers?

**Subscribing to multiple observables:-**

these two observables, you want to combine

these observables into a new observable, and then we'll

subscribe to that observable

 Observable.combineLatest([

    this.route.paramMap,

    this.route.queryParamMap

   ])

   .subscribe(combined =>{

   let id=combined[0].get("id");

   let page= combined[1].get('page');

  //  this.service.getAll({id:id , page:page})

   this.service.getAll().subscribe(...)

   })

Not good code

Switchmapoperator:-

return type of getAll. Now it returns an observableof response. So here we have a collection, andin this collection we'e going to get a response object that arrives asynchronously, when we apply the map operator.

Let's look at the return type of this method, now instead of

observable of response, we get observable of any.

Programatic Navigation:-

this.router.naviate(['/follower'] , {queryParams:{page:1,order:'newest'}})

**Consuming Http services:-**

<http://jsonplaceholder.typicode.com/> fake API Service

appmodule.ts

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

  imports: [

HttpClientModule

],

**Create component posts:-**

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

constructor(private http: HttpClient ) { }

  ngOnInit(): void {

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

  }

**Get gives us observable response**

I'm going touse this http object to send a request to our backend. So,http.get and with this we cansend an http get request to the server. So we use promisesand observables, and with these classes when the result is ready we'll be notified. Observable, has a method called subscribe.  So when the result is ready, we'll be notified. ubscribe method has three

overloads, which basically means there are three ways we can use this

subscribe method. If you press the up and down arrows,

you can see different overloads of the subscribe method.

And each overload has different parameters, the one that

we're going to use is the third overload here. So here we have

two parameters, nix and error, and as you can see

with this question mark, they are both optional. Now look at the

type of the next parameter. We have an error function here. So this

unction, takes a value, which is a response, and it returns

void. So here, I can pass an error function,

response goes to a code block.

So this is what we call a subscription function. And

 ngOnInit()  {

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

     console.log(response);

     console.log(response.json());

    })

  }

 this response object,has a method called json. And with this we can convert the response to the json object

**But here defaultly got json object**

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

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

@Component({

  selector: 'posts',

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

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

})

export class PostsComponent implements OnInit {

  posts:any=[];

  constructor(private http: HttpClient ) { }

  ngOnInit()  {

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

      this.posts=response;

    })

  }

}

 <div class="container">

     <div class="row">

         <div class="col-sm-12  ">

 <ul class="list-group">

     <li class="list-group-item" \*ngFor="let post of posts">

{{post.title}}

     </li>

 </ul>

</div>

</div>

</div>

**Creating data:**

his parameter title is not the title

object or a string. It's an HTML input element. So,

we want to avoid confusion for other people reading this code, and also

improve component time checkin. So this is an HTML

InputElement.

  <div class="row">

         <div class="col-sm-12  ">

    <input class="form-control"  #title (keyup.enter)="createPost(title.value)">

 <ul class="list-group">

     <li class="list-group-item" \*ngFor="let post of posts">

{{post.title}}

     </li>

 </ul>

</div>

</div>

</div>

**Ts**

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

createPost(title: HTMLInputElement)

  {

        console.log(title);

      this.http.post(this.url)

  }

in our **CreatePost** method, I'm going to

pass this.url. Now as the second argument

for this method. We need to pass the body of the request.

When calling HTTP services for data, the body of

the request should be a json object, if that's the object we want to send to the server.

So, I want to create a post object

here, like this. This post should have a title,

and the value of this title should be the value we have in our

title input field. So title.value.

Now I don't write like this code, because look, title is set to

title.value. It's a little bit weird. So I would rather change

the name of this parameter to input,

or title input. I prefer input because input is shorter.

Right? So now we have a post object and of course here I've set only

  createPost(input: HTMLInputElement)

  {

       let post={title:input.value}

      this.http.post(this.url, JSON.stringify(post)).subscribe(response =>{

        post['id']= response;

        this.posts.splice(0,0,post)

      })

  }

**Updating post:-**

ttp request to update the data. Now here we have two choices.

We can either call this.http

.put or we can call the path method.

What is the difference? We use the patch method to update only a few

properties in an object. So instead of sending the complete representation

of that object to a server, we send only the properties

that should be modified. It's easier to show you in code.

So here, let's say each post has a property called is red.

 updatePost(post)

  {

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

     console.log(response);

   }) ;

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

  }

<ul class="list-group">

     <li class="list-group-item" \*ngFor="let post of posts">

{{post.title}}  <button class="btn btn-default btn-sm" (click)="updatePost(post)" >Update</button>

     </li>

**Delete Post:**

we need to reference a specific post. So we need to upend

via /, and then the idea of the post. So post,

.id. Now note that here we don't have that body

parameter. Because by convention, HTTP delete requests

**delete:-**

<button class="btn btn-default btn-sm" (click)="deletePost(post)" >Delete</button>

deletePost(post)

  {

    this.http.delete(this.url + '/' +post.id ).subscribe(response =>{

    let index= this.posts.indexOf(post);

    console.log(index);

    this.posts.splice(index,1)

    })

  }

**Oninit interface:**

is an interface that we refer to as a life cycle hook.

Seperation of concerns:

update post, and delete post. There is a problem with thisimplementation. The problem is this implementation violates Separation of Concerns principle.

**Extracting a service:-**

**Create service post under services folder.**

Post.service

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

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

@Injectable({

  providedIn: 'root'

})

export class PostService {

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

  constructor(private http:HttpClient) { }

  getPosts()

  {

    return this.http.get(this.url)

  }

  addPost(obj)

  {

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

  }

  remove(id)

  {

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

  }

  editPost(id)

  {

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

  }

}

Post.html

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;

   });

  }

  createPost(input:HTMLInputElement)

  {

   console.log(input.value);

   let post={title:input.value}

   this.service.addPost(post).subscribe(response =>{

     input.value="";

   post["id"]=response;

   this.posts.splice(0,0,post)

   })

  }

  deletePost(post)

  {

    console.log(post);

    this.service.addPost(post.id).subscribe(response =>{

   let index=this.posts.indexOf(post);

   console.log(index);

   this.posts.splice(index,1)

    })

  }

  updatePost(post)

  {

   this.service.editPost(post.id).subscribe(response =>{

     console.log(response);

   })

  }

}

Post.html

 <div class="container">

    <div class="row">

         <div class="col-sm-12  ">

    <input class="form-control"  #title (keyup.enter)="createPost(title)">

 <ul class="list-group">

     <li class="list-group-item" \*ngFor="let post of posts">

{{post.title}}

<button class="btn btn-default btn-sm" (click)="updatePost(post)" >Update</button>

<button class="btn btn-default btn-sm" (click)="deletePost(post)" >Delete</button>

</li>

 </ul>

</div>

</div>

 </div>

**Handling Errors:-**

**Handling Unexcepted Erros:-**

  posts:any=[];

  constructor(private service: PostService ) { }

  ngOnInit()  {

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

   this.posts=response;

   }, error=>{

     alert("An Unexcepted error has occured");

     console.log(error);

   });

  }

  createPost(input:HTMLInputElement)

  {

   console.log(input.value);

   let post={title:input.value}

   this.service.addPost(post).subscribe(response =>{

     input.value="";

   post["id"]=response;

   this.posts.splice(0,0,post)

   },error =>{

   alert("An unexcepted error has occured");

   console.log(error);

   })

  }

  deletePost(post)

  {

    console.log(post);

    this.service.addPost(post.id).subscribe(response =>{

   let index=this.posts.indexOf(post);

   console.log(index);

   this.posts.splice(index,1)

    },error =>{

      alert("An unexcepted error has occured");

      console.log(error);

    })

  }

  updatePost(post)

  {

   this.service.editPost(post.id).subscribe(response =>{

     console.log(response);

   },error =>{

     alert("An unexcepted error has occureds");

     console.log(error);

   })

  }

**Handling excepted 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().subscribe(response=>{

   this.posts=response;

   }, error=>{

     alert("An Unexcepted error has occured");

     console.log(error);

   });

  }

  createPost(input:HTMLInputElement)

  {

   console.log(input.value);

   let post={title:input.value}

   this.service.addPost(post).subscribe(response =>{

     input.value="";

   post["id"]=response;

   this.posts.splice(0,0,post)

   },  (error:Response) =>{

    if(error.status === 400)

    {

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

    }

    else

    {

    alert("An unexcepted error has occured");

    }

    console.log(error);

  })

  }

  deletePost(post)

  {

    console.log(post);

    this.service.addPost(121212122).subscribe(response =>{

   let index=this.posts.indexOf(post);

   console.log(index);

   this.posts.splice(index,1)

    },  (error:Response) =>{

      if(error.status === 404)

      {

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

      }

      else

      {

      alert("An unexcepted error has occured");

      }

      console.log(error);

    })

  }

  updatePost(post)

  {

   this.service.editPost(post.id).subscribe(response =>{

     console.log(response);

   },error =>{

     alert("An unexcepted error has occureds");

     console.log(error);

   })

  }

}

**Throwing Application specific errors:**

So, let's go back to our post service.

Here in deletePost method, if there's an exception, if there's an

error. I want to catch that error, and instead of sending the response object

this delete method returns and

observable. Observable is a type that is part of a third party library check for the status of the error,

so if it's 404, or not found, we want to return

a different kind of error

service.ts

import {Observable} from 'rxjs/Observable';

import  'rxjs/add/operator/catch';

 import { AppError } from './app.error';

deletepost(post){

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

 .catch((error:Response )=>{

 return  Observable.throw(new AppError(error))

 })

}

Create app.error.ts

export class AppError

{

    constructor(public originalError?:any)

    {

    }

}

Create not-found-error-ts

import { AppError } from './app.error';

export class NotFoundError extends AppError

{

}

deletepost(post){

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

 .catch((error:Response )=>{

   if(error.status === 404)

   return Observable.throw(new NotFoundError())

 return  Observable.throw(new AppError(error))

 })

}

App.ts

deletepost(post)

{

 this.service.deletepost(post).subscribe(response =>{

   console.log(response);

  let i= this.postList.indexOf(post);

  this.postList.splice(i,1)

 },(error:AppError) =>{

   if(error instanceof NotFoundError)

   {

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

   }

   else{

    alert("an unexcepted error has occured");

    console.log(error);

   }

 })

}

**Handling bad Request Errors:-**

in thepost service, in create post method, we want to catch any potential errors,and instead throw application specific errors.

Create bad-input.ts

import { AppError } from './app.error';

export class BadInput extends AppError

{

}

postdata(obj)

{

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

.catch((error:Response)=>{

if(error.status === 404)

 return Observable.throw(new BadInput(error.json()));

 return Observable.throw(new AppError(error.json()));

})

}

App.ts

onkeyUp(input)

{

  this.manageRolesErrorMessage="loading";

  let post={title:input.value}

  this.service.postdata(post).subscribe(response =>{

    console.log(response);

     post["id"]=response;

     this.postList.splice(0,0,post)

     this.manageRolesErrorMessage="";

  },(error:AppError) =>{

    if(error instanceof BadInput){

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

    }

    else{

      alert("an unexcepted error has occured");

      console.log(error);

    }

  })

}

Importing observable operators and factory methods:-

Service.ts

import 'rxjs/add/observable/throw';

Globa;l error handling:

Create file app.error-handler.ts

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

export class AppErrorHandler implements ErrorHandler

{

    handleError(error) {

          alert("an unexcepted error has occured");

console.log(error);

}

}

Appmodule

import  {AppErrorHandler} from './app.error-handler';

  providers: [  {provide: ErrorHandler ,useClass:AppErrorHandler}],

app.html

getposts(){

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

     this.postList=response;

  }, )

}

onkeyUp(input)

{

  this.manageRolesErrorMessage="loading";

  let post={title:input.value}

  this.service.postdata(post).subscribe(response =>{

    console.log(response);

     post["id"]=response;

     this.postList.splice(0,0,post)

     this.manageRolesErrorMessage="";

  },(error:AppError) =>{

    if(error instanceof BadInput){

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

      // this.form.setErrors(error.originalError)

    }

    else throw error;

    })

 }

updatepost(post)

  {

    this.service.updatePost(post).subscribe(response=>{

     console.log(response);

   } ) ;

  }

deletepost(post)

{

 this.service.deletepost(post).subscribe(response =>{

   console.log(response);

  let i= this.postList.indexOf(post);

  this.postList.splice(i,1)

 },(error:AppError) =>{

   if(error instanceof NotFoundError)

   {

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

   }

   else throw error;

  })

}

}

Extractin a reusable error handling method:

private handleError(error:Response)  {

  if(error.status === 404)

   return Observable.throw(new NotFoundError())

 return  Observable.throw(new AppError(error))

}

updatePost(post)

{

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

.catch(this.handleError)

}

deletepost(post){

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

 .catch(this.handleError )

}

postdata(obj)

{

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

.catch((error:Response)=>{

// if(error.status === 404)

//  return Observable.throw(new BadInput(error.json()));

 return Observable.throw(new AppError(error.json()));

})

}

private handleError(error:Response)  {

  if(error.status === 404)

  return Observable.throw(new BadInput(error.json()));

  if(error.status === 404)

   return Observable.throw(new NotFoundError())

 return  Observable.throw(new AppError(error))

}

postdata(obj)

{

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

.catch((this.handleError))

}

Overall service.ts

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

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

import {Observable} from 'rxjs/Observable';

import  'rxjs/add/operator/catch';

 import { AppError } from './app.error';

import { NotFoundError } from './not-found-error';

import { BadInput } from './bad-input';

import 'rxjs/add/observable/throw';

@Injectable({

  providedIn: 'root'

})

export class PostService {

  private url="http://jsonplaceholder.typicode.com/posts"

  constructor(private http:HttpClient) { }

getPosts()

{

  return this.http.get(this.url)

}

postdata(obj)

{

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

.catch((this.handleError))

}

updatePost(post)

{

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

.catch(this.handleError)

}

deletepost(post){

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

 .catch(this.handleError )

}

private handleError(error:Response)  {

  if(error.status === 404)

  return Observable.throw(new BadInput(error.json()));

  if(error.status === 404)

   return Observable.throw(new NotFoundError())

 return  Observable.throw(new AppError(error))

}

}

**Extracting reusable data service:-**

**Crate data.service.ts**

import { BadInput } from './bad-input';

import { NotFoundError } from './not-found-error';

import { AppError } from './app.error';

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

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

import { Observable } from 'rxjs/Observable';

import 'rxjs/add/operator/catch';

import 'rxjs/add/operator/map';

import 'rxjs/add/operator/toPromise';

import 'rxjs/add/observable/throw';

@Injectable()

export class DataService {

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

  getPosts()

{

  return this.http.get(this.url)

}

postdata(obj)

{

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

.catch((this.handleError))

}

updatePost(post)

{

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

.catch(this.handleError)

}

deletepost(post){

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

 .catch(this.handleError )

}

private handleError(error:Response)  {

  if(error.status === 404)

  return Observable.throw(new BadInput(error.json()));

  if(error.status === 404)

   return Observable.throw(new NotFoundError())

 return  Observable.throw(new AppError(error))

}

}

**Service.ts**

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

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

import { DataService } from './data.service';

@Injectable({

  providedIn: 'root'

})

export class PostService  extends DataService{

  // private url="http://jsonplaceholder.typicode.com/posts"

  constructor(http: HttpClient) {

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

   }

 }

**Authentication and Authorization:-**

authentication. In order to implement authentication, on the client we should build a login page, and on the server we should build an APIN point, to validate the user. When theuser clicks on the login button, our Angular app is going to call this APIendpoint, and pass the username and password.Now on the server, you're going to validate these credentials, and if they arevalid, we're going to return what we call a json web tokenor JWT. This token is basicallya json object that includes certain attributes about thelogin user. And we use it to indentify the user on the clientand also on the server.

Now we get this json web token on the client, and then we

need to store it somewhere persistent so it can exist

across session restarts, so if the user closes their browser

but then opens it again, the token should still be there.

We use the local storage for that. So almost all modern

browsers have a simple storage per website, and they also

provide an API for storing key value pairs into the

storage. So we're going to use this native API to

store our json web token inside the browser's local storage.

Now on the client, we can use this token to identify the user.

For example, we can display their name on the navigation bar,

we can show or hide parts of a page, or prevent access

to certain routes if they don't have a valid token. This is

how we work with these tokens on the client. Now let's say the user

wants to get the list of orders from the server and this list is

only available to authenticated users. So on the server,

we have an API end point, like /api/orders.

In order to get the list of orders, on the client we

should include the JWT in the request header.

And then on the server, we should extract this token,

validate it, and if it's valid we'll return the list of orders.

Otherwise, we'll return an unauthorized response,

that's the response with the status code 401. So this

is the big picture. Next we're going to look at these json web tokens

in more detail.



**Starter Code:**

**download the attached file:-**

go to fake-backend.ts:-

**implement Login:-**

**Angular Animations:-**

.stretch

{

animation-name:stretch;

animation-duration:1.5s

}

**we have a library called animate.css**

**Implementing Fade-in animation**

So it's a library that gives us a bunch of predefined classes for various kinds of animations.

in angular we have three kinds of states.(void,default,custom).You have to avoid state the default state and custom States.

apmodule.ts

void state represents the state of an element that is not part of the dom.

appmodule.ts

1)import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

 imports: [

    BrowserAnimationsModule

  ],

Polyfill

code that allows to use modern javascript features in older browsers.

polyphills.ts:

uncomment  import 'web-animations-js';  // Run `npm install --save web-animations-js`. and install

npm install --save web-animations-js

implementing fade-in animation:-

we called a trigger function which comes in angular animations library.

todos.ts:

import {trigger} from '@angular/animations';

We need to register all the states and the transitions for this animation.So we often have calls to two functions.One is the state function and the other is transition.

**todo.ts**

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

import {trigger,animate, style,transition} from '@angular/animations';

// import {  } from '@angular/core/src/animation/dsl';

@Component({

  selector: 'todos',

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

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

  animations:[

    trigger('fade',[

                transition('void => \*',[

                  style({backgroundColor:'red',opacity:0 }),

                     animate(2000,style({backgroundColor:'white' , opacity:1}))

                ])

  ])

]

})

export class TodosComponent {

  items: any[] = [

    'Wash the dishes',

    'Call the accountant',

    'Apply for a car insurance'];

  addItem(input: HTMLInputElement) {

    this.items.splice(0, 0, input.value);

    input.value = '';

  }

  removeItem(item) {

    let index = this.items.indexOf(item);

    this.items.splice(index, 1);

  }

}

**todo.html**

<h1>Todos</h1>

<input #itemInput

  class="form-control"

  (keyup.enter)="addItem(itemInput)">

<div \*ngIf="items" class="list-group" >

  <button type="button"

     @fade

      \*ngFor="let item of items"

      (click)="removeItem(item)"

      class="list-group-item"

      >{{ item }}</button>

</div>

If you have a call to the animate function with only a timing value without any styles this functionis going to undo all the previous styles applied over this period of time.

animations:[

   trigger('fade',[

               transition('void => \*',[

                 style({backgroundColor:'red',opacity:0 }),

                    animate(2000)

               ])

 ])

]]

Initially we want to set its opacity to zero which will make this element invisible and then over a

period of time in this case two seconds you want to change that opacity to one which will make the element appear on the view.

animations:[

   trigger('fade',[

               transition('void => \*',[

                 style({opacity:0 }),

                    animate(2000)

               ])

 ])

]

Implementing Fadeout Animation:

-------------------------------

when creating animation is happening but while deleting no animation happened.

we have a transition from the void state to the default state.

Now we want to define another transition from the default state to the void state because when we click an item it's going to be removed from the dump.

animations:[

   trigger('fade',[

               transition('void => \*',[

                 style({opacity:0 }),

                    animate(2000)

               ]),

               transition('\* => void',[

                    animate(2000, style({ opacity:0}))

               ])

 ])

]

we have something that is duplicated. That is this style object here.

Well if you pay close attention here this is the style of the element in the void state.So when the element is out of the dom it's opacity should be zero.

So to clean up this code we can define the style for the void state.

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

 animations:[

    trigger('fade',[

           state('void',style({opacity:0})),

                transition('void => \*',[

                     animate(2000)

                ]),

                transition('\* => void',[

                     animate(2000,)

                ])

  ])

]

we still have a little bit of duplication in our court.So look you have repeated this and you made two thousand milliseconds in two place

animations:[

    trigger('fade',[

           state('void',style({opacity:0})),

                transition('void => \*, \* =>void',[

                     animate(2000)

                ])

  ])

]

So instead of having to unidirectional directional state change expressions we can have a bi directional

state change expression.

So I can delete the second expression and make this bi directional.

animations:[

    trigger('fade',[

           state('void',style({opacity:0})),

                transition('void <=> \*',[

                     animate(2000)

                ])

  ])

]

These two expressions are very common so in angular we have an alias for these expressions.

So for the first one.

When the element transitions from the void to the default state are alias is Inter.

So you add color enter and for the second expression are aliases leave.

So that's when the element is leaving the dome like this so using this alias.

In my opinion is cleaner and more readable than a bi directional state change expression like this.

But if you proofread the other syntax all means go for it.

 trigger('fade',[

           state('void',style({opacity:0})),

                transition(':enter,:leave',[ // void <=>\*

                     animate(2000)

                ])

  ])

]

**Angular Material2:-**

angular material Is a library of reusable and high quality you components that are built with angular and typescript.

**Installing Angular Material:-**

npm i --save @angular/cdk @angular/material @angular/animations hammerjs

a library that allows you to build awesome components for the web but without adopting the material.

 hammer J.S. is a powerful library that allows you to add just your support to your pages.

 styles.css

 @import "~@angular/material/prebuilt-themes/indigo-pink.css";

 Appmodule

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

import {MatCheckboxModule} from '@angular/material/checkbox';

**checkbox:-**

 <mat-checkbox

 #showDetails

 value="2" [checked]="isChecked"  (change)="onChange($event)"

 > Show details</mat-checkbox>

<div \*ngIf="showDetails.checked">

    Show Detaails....

</div>

<!-- <input type="checkbox"  checked="checked" value="" (change)="valuechange()">  -->

isChecked="true";

  onChange($event)

  {

    console.log($event);

  }

**Radio Buttons:-**

<!-- <mat-radio-group aria-label="Select an option">

    <mat-radio-button value="1">Male</mat-radio-button>

    <mat-radio-button value="2"  [checked]="isChecked"  >Female</mat-radio-button>

  </mat-radio-group> -->

  <mat-radio-group  value="0">

    <mat-radio-button value="1">Male</mat-radio-button>

    <mat-radio-button value="0"    >Female</mat-radio-button>

  </mat-radio-group>

**Dropdown-list:-**

 <!-- <select  >

     <option

     \*ngFor="let color of colors"

     [value]="color.id">{{color.name}}

     </option>

 </select> -->

 <md-select [(ngModel)]="color" >

    <md-option

    \*ngFor="let color of colors"

    [value]="color.id">{{color.name}}

    </md-option>

</md-select>

 colors=[

     {id:1, name:'red'},

     {id:2, name:'green'},

     {id:3, name:'black'}

   ]

color=2;

**Input:-**

 <input

 ngModel

 #username="ngModel"

 type="text"

 required>

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

 The username filed is required

 </div>

**TextAreas:-**

**DatePicker:-**

**Icons:**

**Tooltips:-**