Angular

Angular is an open source javascript library used for building front end client side web applications maintained by google

Angular is based on typescript which is developed by microsoft.

Features of Angular:

1.used to create complex and scalable applications.

2.single page applications.

3.asynchronous http requests.

4.URL routing to navigate around an application.

5.components are building blocks -used to have clean way of organizing UI Code into self contained,reusable chunks

6.business logic,structure and styling can be composed into the component code

7.In Angular 7, the CLI prompts have been updated to v7.0.2 with new features. prompt users when typing commands like @angular/material, ng-new, and ng-add.

8.The new drag-drop module basically provides a better way to easily create drag & drop interfaces

9.new Virtual Scrolling in Angular 7 basically loads and unloads items from the DOM depending upon visible parts of lists, resulting into a much faster experiences.

Typescript

It is a superset version of javascript.

allows to use latest javascript features that can run angular applications.

Angular CLI added the package for Typescript compiler which is set up to generate js files automatically when a change in ts file is detected.

Features:

1.support for type annotations-applies type checking when the code is compiled to reduce common errors.

Classes

classes are templates used to create objects that have identical functionality.

used to define types that can be instantiated with new keyword to create objects that have data and behaviour.

'class' keyword is used to declare a class followed by the name of the class.

constructor function is invoked when a new object is created and used to have intial setup that the class requires.

classes have properties and methods.

when we refer to one of the members of the class we prepend 'this' keyword. 'this' denotes that it’s a member access.

methods are functions without need to use function keyword

Everything in a class is public if not specified. Everything in a module is private unless export keyword is used.

Example:

class social media

{

constructor(name)

{

console.log(name+"facebook"+"\n"+"twitter"+"whatsapp");

}

link(name)

{

console.log("media is "+name);

}

face()

{

return "facebook";

}

}

let s=new socialmedia("medias are ");

s.link("linkedln");

console.log(s.face());

Inheritance

classes can inherit from another class by using 'extends' keyword.

'extends' keyword is used to declare the class that will be inherited from,known as the super class or base class.

We use the extends keyword to signal that this class inherits all the properties and methods from the parent class.

We can describe additional properties.

We use the 'super' function to call the constructor and methods of the parent class.

We can override member functions of the parent class with our own versions.

In member functions 'super' refers to the parent instance.

Example:

class socialmedia

{

constructor(n,l)

{

this.name=n;

this.likes=l;

document.write(this.name+"<br>");

}

disp()

{

return(this.name);

}

cal()

{

document.write((this.likes/1000)+"k");

}

}

var wp=new socialmedia("whatapp",56000);

document.write(wp.disp());

wp.cal();

var tw=new socialmedia("twitter",78000);

tw.disp();

tw.cal();

Interface

An interface is an abstract type, it does not contain any code as a class does.

It only defines the 'signature' or shape of an API.

During transpilation, an interface will not generate any code.

It is only used by Typescript for type checking during development.

Example

export interface data

{

ename:string;

rno:number;

sal:number;

today:boolean;

save();

}

export class emp implements data

{

today:boolean=true;

constructor()

{console.log(this.today);

}

}

let s=new emp();

Fat Arrow functions

One of the popular feature of ES6 which allows us to write concise functions.

-If the function only contains one expression we can drop the braces{}.

- if it is on one line, the statement gets returned automatically without the need to use the return keyword.

Example

var f1=function(st){

return st;

}

console.log(f1('fat arrow'));

let f=(s)=>s;

console.log(f('fat arrow function'));

let add=(a:number,b:number):number=>a+b;

console.log(add(5,6));

- if we use fat arrow functions the value of this inside a fat arrow function will be the same as the value of this outside the fat arrow function.

Example

let person={

name:"ojas",

disp:function()

{

console.log(this.name);

setTimeout(()=>

console.log("inside fn"+this.name)

,4000);

}

}

person.disp();

Variables declared by 'let' have their scope in the block for which they are defined, as well as in any contained sub-blocks

Variables declared by 'var' have their scope in the entire enclosing function

Example

function scope()

{

if(true)

{

var a=100; //let a=100; //const a=100;

a=a+1;

console.log(a);

}

sub();

function sub()

{

console.log("inside sub"+a);

}

console.log(a);

}

scope();

Redeclaring the same variable within the same function or block scope raises a SyntaxError.

can't redeclare variables with 'let' keyword whereas 'var' keyword allows redeclaration of variables.

function scope()

{

let a=1;

let a=100;

console.log(a);

}

scope();

var=function scoped variables

let const =block scoped

MODULES

Modules are a way to create a local scope in the file. So, all variables, classes, functions,etc. that are declared in a module are not accessible outside the module. A module can be created using the keyword export and a module can be used in another module using the keyword import.

Export

A module can be defined in a separate .ts file which can contain functions, variables, interfaces and classes. Use the prefix export with all the definitions you want to include in a module and want to access from other modules.

Import

A module can be used in another module using an import statement.

Syntax:

import { export name } from "file path without extension".

Requirements:

1.Node.js version 8.x or 10.x. To get Node.js, go to nodejs.org.

To check version, run node -v in a terminal/console window.

2.Node Package Manager

Angular apps depend on features and functionality provided by libraries that are available as npm packages

3.Install the Angular CLI

To install the CLI using npm, type command

npm install -g @angular/cli

4.Create a new workspace

ng new appname

The Angular CLI

-installs the necessary Angular npm packages and other dependencies.

-creates a new workspace, with a root folder named appname

5.Serve the application

Go to the current workspace folder

cd appname

Launch the server by using the CLI command

ng serve.

ANGULAR PROJECT STRUCTURE

Every Angular web application relies on an HTML file that is loaded by the browser and starts the application

The HTML document includes an **app element**,which is the placeholder

The application’s files are within the src/app folder

**Package.json:**

Refers to the dependencies the application is dependent on.

lets you keep track of project dependencies.

JSON format where key-name of the package

Value-version of the package

As the application evolves add packages to package.json file

**Package-lock.json**

the package-lock specifies a version, location and integrity hash for every module and each of its dependencies, the install it creates will be the same, every single time

**TSLint.json**

TSLint is a static code analysis tool used in software development for checking Typescript code quality , if TypeScript source code complies with coding rules. TSLint checks your TypeScript code for readability, maintainability, and functionality errors.

HOW ANGULAR WORKS

When the browser requests URL [http://localhost:4200,development](about:blank) server responds with the contents of the index.html file in the src folder

The server populates the body with script elements that tell browser to load the bundle files

ROOT MODULE

The Angular CLI generates the following basic app module when creating a new app.

Angular application needs a bootstrap file to start the application which is **main.ts**

Every application has a root module that it needs to start the application.

conventional name for the root module is **app.module.ts** and root component is the appcomponent i.e app.component.ts

INTERPOLATION

A TypeScript class to shape the model data for your component and display properties of that model.

To display a component property is to bind the property name through interpolation. With interpolation, you put the property name in the view template, enclosed in double curly braces:

<p>message={{msg}}</p>

<p> {{‘message=’+msg}}></p>

<img src=”{{url}}”>

Property Binding:

<p [innerHtml]=”msg”></p>

<img [src]=”urlpath”></p>

<button [disabled]=”stat”>click</button> [hidden]

Cannonical form:

<img bind-src=”urlpath”>

Attribute Binding:

we do not have a corresponding property in the DOM for colspan attribute. To fix this we have to use attribute-binding in Angular, which sets the colspan attribute. To tell angular framework that we are setting an attribute value we have to prefix the attribute name with attr and a DOT as shown below.

<th [attr.colspan]="columnSpan">

The same is true when using interpolation

<th attr.colspan="{{columnSpan}}">

DIRECTIVES

A directive is a custom **HTML** element that is used to extend the power of **HTML**.

There are three kinds of directives in Angular:

Components—directives with a template.

**Structural directives**—change the DOM layout by adding and removing DOM elements.

\*ngFor-Angular [ngFor](https://angular.io/api/common/NgForOf) directive in the template to display each item in the collection.

**Example:**

<ul>

<li \*[ngFor](https://angular.io/api/common/NgForOf)="let s of student">

{{ s }}

</li>

</ul>

This UI uses the HTML unordered list with <ul> and <li> tags. The \*[ngFor](https://angular.io/api/common/NgForOf) in the <li> element is the Angular "repeater" directive.

Angular duplicates the <li> for each item in the list, setting the variable to the item in the current iteration. Angular uses that variable as the context for the interpolation in the double curly braces.

\*ngIf-The Angular [ngIf](https://angular.io/api/common/NgIf) directive inserts or removes an element based on a *truthy/falsy*condition.

marks:number[]=[40,50,60];

<div \*ngFor="let m of marks">

<p \*ngIf="m>50">{{m}}</p>

</div>

**Attribute directives**—change the appearance or behavior of an element

**ng Style**

The NgStyle directive lets you set a given DOM elements style properties.

One way to set styles is by using the NgStyle directive and assigning it an object literal, like so:

<div [ngStyle]="{'background-color':'green'}"></<div>

This sets the background color of the div to green.

[ngStyle] becomes much more useful when the value is dynamic. The values in the object literal that we assign to ngStyle can be javascript expressions which are evaluated and the result of that expression is used as the value of the css property

**Example:**

<p \*ngFor="let m of marks" [ngStyle]="{backgroundColor:m>50?'green':'red',

color:m>50?’light green’:’pink’}">

{{m}}

</p>

**ngClass**

The NgClass directive allows you to set the CSS class dynamically for a DOM element.

[ngClass] directive is used to assign multiple CSS based on multiple conditions.

<div [ngClass]="{

'example-class': condition,

'other-class': !condition

}">

</div>

**Custom Directives:**

Ng g d mouse

Included in declarations:[**MouseDirective**]

@HostBinding lets you set properties on the element or component that hosts the directive, and @HostListener lets you listen for events on the host element or component.

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

@**Directive**({

selector: '[appMouse]'

})

export class **MouseDirective** {

constructor(private el:**ElementRef**) {

}

@**HostBinding**('style.backgroundColor')bg="lightgreen";

@**HostListener**('mouseover') **do**()

{

this.el.nativeElement.**setAttribute**('src', 'assets/pics/pic1.webp');

this.el.nativeElement.style.border="5px solid grey";

this.el.nativeElement.style.backgroundColor="lightpink";

}

@**HostListener**('mouseout') **dosome**()

{

this.el.nativeElement.style.backgroundColor="lightgreen";

this.el.nativeElement.**setAttribute**('src', 'assets/pics/pic2.webp');

this.el.nativeElement.style.border="none";

}

}

<img *width*="100px" *appMouse* *src*="assets/pics/flower.jpg">

<input *appMouse* *[value]*="vall" *(input)*="fn($event)">

<input *appMouse* *[ngModel]*="val" *(ngModelChange)*="val=$event">

{{val}}

In Angular, there are 3 types of bindings:

1. Property Binding
2. Event Binding
3. Two-way Binding

## 1. Property Binding

Property binding means we pass the data from the component class and set the value to the given element in the view. This is one way that data is passed from the component to the view.

The square brackets set the value from the component class to the HTML element in the view.

**Example**

export class AppComponent

{

picture:string=”./assets/pics/pic1/jpg”

}

<img [src]=”picture”>

Or using Interpolation {{}}

export class AppComponent

{

picture:string=”./assets/pics/pic1/jpg”

}

<img src=”{{picture}}”>

## 2.Event Binding

All the user interactions with the application happen through clicks, double clicks, or hovering or maybe some key actions such as keypress, key up, or key down.

Event binding is the one-way data binding which sends the value from the view to the component, which is in contrast to the property binding where we used to send the data from component to the view.

export class AppComponent{

logevent()

{

alert("Hi");

}

<button (click)="logevent()">Log</button>

### 3.Two-Way Binding

There might be some cases where a user enters in some values and the UI will be updated accordingly. Two-way binding is the combination of property binding [] and event binding (), so two-way binding can be used by combining the two, such as: ([……]).

<input [(ngModel)]="username">

<p>Hello {{username}}!</p>

<input [value]="username" (input)="username = $event.target.value">

<p>Hello {{username}}!</p>

<input [ngModel]="username" (ngModelChange)="username = $event">

<p>Hello {{username}}!</p>

The property binding [ngModel] takes care of updating the input element.

The event binding (ngModelChange) notifies the outside world when there was a change in the input element.

Creating Components

Angular 4 components are simply classes that are designated as a component with the help of a component decorator.

Every component has a defined template which can communicate with the code defined in the component class.

Within the component, you have a variety of configuration properties that help define this given component.

* **selector**: This is the name of the tag that the component is applied to. For instance: **<app-root>Loading...</app-root>** within index.html.
* **templateUrl** & **styleUrls**: These define the HTML template and stylesheets associated with this component. You can also use **template** and **styles** properties to define inline HTML and CSS.

Working with multiple components

Breaking up an application into multiple logical components makes it easier to:

• Architect an application as it grows in complexity.

• Re-use common components in multiple places.

**Nested Components:**

To implement one component within another component :

In "app.module.ts" file we need to do 2 things.

1.Import firstComponent

2.Add firstComponent to the declarations array

Angular cli to create a new component

**Ng generate component home**

//app.module.ts

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

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

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

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

@NgModule({

imports: [ BrowserModule ],

declarations: [ AppComponent ,HomeComponent],

bootstrap: [AppComponent ]

})

export class AppModule {}

//app.component.ts

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

@Component({

selector:'app-root',

template:`<h1>this is angular</h1>

<h2>this is app (outer) component</h2>

**<app-home></app-home>** `

})

export class AppComponent{

}

//home.component.ts

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

@Component({

selector:'app-home',

template:`<h2>this is home (inner) component</h2>`

})

export class HomeComponent{}

Custom property Binding:(sharing data from app component(root) to home component(child)

@Input is used to make data flow from appcomponent (parent) to (child) homecomponent used for custom property binding

syntax:

< [customprop]="value">

@Input() customprop;

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

.

.

.

class HomeComponent {

@Input() human;

}

This tells Angular that the human property is an *input* property and therefore in HTML we can bind to it using the [] input property binding syntax.

<app-home [human]="student"></app-home>

App.comp.html

<main *class*="container">

<section *class*="row">

<div *class*="col-6 bg-primary">

<h1>this is app (parent) component</h1>

{{text}}

<input *[(ngModel)]*="data">

<button *(click)*="sendata=data">send data</button>

</div>

<article *class*="col-6 bg-warning">

<h1>this is child component</h1>

<app-structuraldir *[human]*="text" *[sen]*="sendata"></app-structuraldir>

</article>

</section>

</main>

App.comp.ts

export class **AppComponent** {

text:string="this is from app";

arr:number[]=[12,45,89,90,12,78];

sendata;

Strdir.comp.html

<h1 *class*="bg-danger text-white">sen={{sen}} {{human}}</h1>

Strdir.comp.ts

export class **StructuraldirComponent** implements **OnInit** {

@**Input**() human;

@**Input**() sen;

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

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

@Component({

selector: 'my-app',

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

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

})

export class AppComponent {

student:any=

{

name:"aaa",

id:1001,

food:"briyani"

};

customprop="this is @Input for custom property";

secustom:string[]=["w","e","l"];

app.component.html

<my-home [human]="student" [myprop]="custom prop" [sprop]="secustom"></my-home>

home.component.ts

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

@Component({

selector:'my-home',

templateUrl:'./home.component.html'

})

export class homeComponent

{

@Input() human;

@Input(‘myprop’) name; //property data comes from parent in ‘myprop’ but assign the value in name property in child component

@Input() sprop;

}

home.component.html

<p>this is from home component</p>

<h2>{{[human.name](http://human.name/)}}</h2>

<h2>{{myprop}}</h2>

<h2>{{sprop[1]}}</h2>

Custom Events :(sharing data from side-bar component(child) to app component(root/parent)

To create a custom output event on our component we need to do two things:

1. Create an EventEmitter property on the SideBarComponent class.

2. Similar to when we created a custom input property binding, we need to annotate that property

with the @Output decorator.

An EventEmitter is a helper class which we can use to emit events when something

happens, other components can then bind and react to these events.

App.module.ts

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

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

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

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

import { SideBarComponent } from './side-bar/side-bar.component';

@NgModule({

imports: [ BrowserModule, FormsModule ],

declarations: [ AppComponent, SideBarComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

app.component.html

<app-side-bar (fire)="trgfn($event)"></app-side-bar>

App.component.ts

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

@Component({

selector: 'my-app',

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

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

})

export class AppComponent {

trgfn(e)

{

alert("custom ev i m triggered");

console.log(e);

}

}

Side-bar.component.html

<button type="button" (click)="evfn($event)">click to trigger</button>

Side-bar.component.ts

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

@Component({

selector: 'app-side-bar',

templateUrl: './side-bar.component.html',

styleUrls: ['./side-bar.component.css']

})

export class SideBarComponent {

@Output() fire=new EventEmitter();

evfn(ev)

{

alert("evfn triggers");

this.fire.emit(ev);

console.log(ev);

}

}

**Template Driven Forms**

In template driven we use directives to create the model.

In model driven we create a model on the component and then use directives to

map elements in the template to our form model.

'

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

One of the directives pulled in via the FormsModule is called NgForm.

This directive has a selector which matches the <form> tag.

So just by adding FormsModule to our NgModule imports our template form is already associated wth

an instance of the NgForm directive.

This instance of ngForm is hidden but we can expose it with a local template reference variable

attached to the form element, like so:

<form #f="ngForm"> ... </form>

Now we can use the variable f in our template and it will point to our instance of the ngForm

directive.

app.component.ts

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

@Component({

selector: 'my-app',

templateUrl:'./app.component.html'

,

styles:[`

.er

{

border:3px dotted red;

}

div

{

float:left;

}

.ok

{

border:3px solid green;

}

`]

})

export class AppComponent {

ag:number=12;

min:number=0;

max:number=20;

svfn(frm)

{

console.log(frm.value);

}

}

console.log("form submitted");

}

**Reactive Forms**

With reactive forms, the form model is explicitly defined in the component class.

We use the formGroup property in the <form> tag to bind the form with our userfm form group and we use the formControlName property to bind the <input> tag to individual form controls.

• The template driven approach makes use of built-in directives to build forms such as ngModel, ngModelGroup, and ngForm available from the FormsModule module.

• The model driven approach of creating forms in Angular 6 makes use of FormControl, FormGroupand FormBuilder available from the ReactiveFormsModule module.

• create an example HTML form. We then create a form model in the application component using the FormGroup and FormControl classes.

• Finally, we'll use the formGroup, formControlName and formGroupName directives to bind our form model to our HTML form.

Step 1: Registering the reactive forms module

import { [ReactiveFormsModule](https://angular.io/api/forms/ReactiveFormsModule) } from '@angular/forms';

Step 2:Generating and importing a new form control

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

export class AppComponent {

userfm:FormGroup;

constructor()

{

this.userfm=new FormGroup({

name:new FormControl('',[Validators.required,Validators.minLength(4),Validators.maxLength(7)]),

telno:new FormControl('',[Validators.required,Validators.pattern('[0-9]{3}')]),

mail:new FormControl('',[Validators.required,Validators.pattern('[a-z0-9.\_%+-]+@[a-z0-9.-]+\.[a-z]{2,4}$')]),

edu:new FormGroup({

bachelor:new FormControl('',[Validators.required]),

master:new FormControl('',[Validators.required])

})

Step 3: Registering the control in the template

<form [formGroup]="userfm">

<div class="form-group">

enter name: <input formControlName="name"

[ngClass]="{'is-invalid':userfm.get('name').invalid}" class="form-control">

<div \*ngIf="userfm.get('name').invalid" class="invalid-feedback">

<div \*ngIf="userfm.get('name').errors.required">pls enter name</div>

<div \*ngIf="userfm.get('name').errors.minlength">pls enter 4 chars</div>

<div \*ngIf="userfm.get('name').errors.maxlength">pls enter 7 chars</div>

</div>

</div>

[Reactive Forms](https://amzn.to/2Jov9wx) composed of primarily three Class types. First FormGroups which typically represents a single form. A FormGroup is typically made of many FormControls. A FormControl usually represents a single input in a form. Lastly is the FormArray. The FormArray is used to represent a collection of FormControls that are interrelated.

Input fields have the following states:

• untouched The field has not been touched yet

• touched The field has been touched

• pristine The field has not been modified yet

• dirty The field has been modified

• invalid The field content is not valid

• valid The field content is valid

reactive forms are better when there needs to be some real-time processing of the form as the user types in content. Handling model driven forms with submit handlers is better when there needs to be a discrete action applied when the user presses a button.

Reactive forms come as the best choice to build complex forms. Template driven forms are built with [ng-model](https://docs.angularjs.org/api/ng/directive/ngModel), we write the logic part, validation part, and controls in the template. But in the reactive forms, the logic and validation part is in the controller, that is we define the model of the form in the component. this gives us more control over the form values and validations when compared to template-driven forms.

Reactive Forms are also known as “model-driven” forms because they use the reactive model-driven technique to handle the form data which explicitly manages the form data between component and template. It manages the form data flow between the non-UI data model and UI-oriented form model and returns the states and values of HTML form controls. And it provides reactive patterns and validation to handle forms.

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

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

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

@Component({

selector:'my-app',

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

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

})

export class AppComponent{

myfm=new FormGroup({

user:new FormControl('',[Validators.required,Validators.minLength(6)]),

email:new FormControl(' ',[Validators.required,Validators.email]),

ag:new FormControl('',[Validators.required,Validators.min(30)]),

edu:new FormGroup({

pg:new FormControl('',[Validators.required,Validators.minLength(2),Validators.maxLength(8)]),

ug:new FormControl("",[Validators.required,Validators.minLength(2),Validators.maxLength(8)])

})

});

svfn()

{

console.log(this.myfm.value);

}

}

·  [FormControl](https://angular.io/api/forms/FormControl) tracks the value and validation status of an individual form control.

·  [FormGroup](https://angular.io/api/forms/FormGroup) tracks the same values and status for a collection of form controls.

·  [FormArray](https://angular.io/api/forms/FormArray) tracks the same values and status for an array of form controls.

With reactive forms, the form model is explicitly defined in the component class.

We use the formGroup property in the <form> tag to bind the form with our userfm form group and we use the formControlName property to bind the <input> tag to individual form controls.

export class AppComponent {

username;

edu;

myfm;

ngOnInit()

{

this.myfm=new FormGroup({

username:new FormGroup({

firstname:new FormControl(),

lastname:new FormControl()

}),

edu:new FormGroup({

bachelor:new FormControl(),

master:new FormControl()

})

});

}

}

<form [formGroup]="myfm">

<div formGroupName="username">

<input type="text" formControlName="firstname"><br>

<input type="text" formControlName="lastname">

</div>

<br>

<div formGroupName="edu">

<input type="text" formControlName="bachelor"><br>

<input type="text" formControlName="master">

</div>

<pre>{{myfm.value|json}}</pre>

</form>

The type of the first parameter is AbstractControl, because it is a base class of FormControl, FormArray, and FormGroup, and it allows you to read the value of the control passed to the custom validator function. The custom validator returns either of the following:

1. If the validation fails, it returns an object, which contains a key-value pair. **Key** is the name of the error and the value is always **Booleantrue**.
2. If the validation does not fail, it returns **null**.

function ageRangeValidator(control: AbstractControl): { [key: string]: boolean } | null {

if (control.value !== undefined && (isNaN(control.value) || control.value < 18 || control.value > 45)) {

return { 'ageRange': true };

}

return null;

}

**Pipes**

A pipe takes in data as input and transforms it to a desired output.

It takes integers, strings, arrays, and date as input separated with | to be converted in the format as required and display the same in the browser.

slice:1:3 means return the items from the 1st to the 3rd index inclusive (indexes start at 0).

slice:2 means return the items from the 2nd index to the end of the array.

slice:2:-1 means return the items from the 2nd index to one from the end of the array.

We can use slice inside for loops to only loop over a subset of the array items.

<h1>pipes</h1>

<form>

<input type="text" name="un" [(ngModel)]="unm">

</form>

<h2>{{unm|uppercase}}</h2>

<h2>{{name|uppercase}}</h2>

<h2>{{n2|lowercase}}</h2>

<h2>{{no|number:'3.1-2'}}</h2>

<h2>{{today}}</h2>

<h2>{{today|date:'h:mm:ss a'}}</h2>

<h2>{{today|date:'MMM'|uppercase}}</h2>

<h2>{{ namesarr|slice:2:4}}</h2>

<pre>{{stu|json}}</pre>

<h2>{{45.000|percent}}</h2>

<h2>{{3456.5612456|currency:'INR'}}</h2>

<h2>{{3456.566|currency:'GBP'}}</h2>

<h2>{{3456.566|currency:'USD'}}</h2>

<h2>{{"peers tech"|titlecase}}</h2>

app.component.ts

export class AppComponent{

unm:string="emp";

name:string="peers";

n2:string="TECH";

no:number=123.566;

today=new Date();

stu={id:101,

name:"aaa",

perc:23,

loc:"hyd"

};

namesarr:string[]=["w","e","l","c","o"];

}

Custom Pipes

Ng generate pipe pipename

Eg:ng generate pipe mul

Two files ll b created

Src/app/mul.pipe.spec.ts

Src/app/mul.pipe. ts

Pipes, PipeTransform, "transform" method

To create a custom pipe, we have to import Pipe and Pipe Transform from Angular/core. In the @Pipe directive, we have to give the name to our pipe, which will be used in our .html file. Since, we are creating the mul pipe, we will name it mul.

As we proceed further, we have to create the class and the class name is **MulPipe**. This class will implement the **PipeTransform**.

The transform method defined in the class will take argument as the number and will return the number after taking the multiplication.

Since we have created a new file, we need to add the same in **app.module.ts.**

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

@Pipe({

name: 'mul'

})

export class MulPipe implements PipeTransform {

transform(value:number): number {

let r=value\*value\*value;

return r ;

}

}

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

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

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

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

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

import { MulPipe } from './mul.pipe';

@NgModule({

declarations: [

AppComponent,

MulPipe

],

imports: [

BrowserModule,FormsModule,BrowserAnimationsModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Invoking Custom Pipes

<form #myfm="ngForm">

<input type="number" name="ip" [(ngModel)]="ip">

</form>

<p>{{ip|mul}}</p>

Mul is custom pipe .which finds the cube of number.

HTTP:

* How to generate a fake and complete working REST API,

To work with HttpClient we need a REST API server, you can either use an external API service, create a real Rest API server or create a fake API using json-server

1.npm install -g json-server

2.define your data in a db.json file

{“details”:[{"id": 1,

"name": "Product001"

},{"id": 2,

"name": "Product002"

}]}

3.run a REST server using the following command:

json-server --watch db.json

* How to create Angular services,
* How to subscribe to Observables,
* How to use the async pipe in templates to iterate over Observable data.

HttpClient module in your Angular 6 application,

you need to add it to the imports array in the application main module:

Start by importing the HttpClientModule module from the @angular/common/http package:

import { HttpClientModule } from ‘@angular/common/http’;

Next, add the HttpClientModule module to the imports array of the main module

use the new HTTPClient API to send get, post, put and delete requests to a REST server.

inject HttpClient via the component's constructor

constructor(private httpClient: HttpClient){}

The HTTP PUT method is used to completely replace a resource on the API server. We can use

the HttpClientmodule to send a PUT request to an API server using the the put() method.

we can send an HTTP DELETE request to delete a resource from the API server

using delete() method provided by the HttpClient

The HTTP POST method has many uses but mostly used when we need to add new data on the

server

We are calling the post() method from the injected instance of HttpClient. The first parameter is

the API endpoint and the second parameter is the data object. We also subscribe to the

observable returned by the post() method.

What happens if the request fails on the server, or if a poor network connection prevents it from even reaching the server? [HttpClient](https://angular.io/api/common/http/HttpClient) will return an *error* object instead of a successful response.

.subscribe(

(data: Config) => this.config = { ...data }, // success path

error => this.error = error // error path

);

What are Services

Services are a great way to share information among classes that *don't know each other*. You'll create a myservService and inject it in two places:

1. in myservService which uses the service to send a message.

2. in HomeComponent which displays that message.

3. a situation where we need some code to be used everywhere on the page. It can be for data connection that needs to be shared across components, etc. Services help us achieve that. With services, we can access methods and properties across other components in the entire project.

Creating Services

Ng generate service myserv

CREATE src/app/myserv.service.spec.ts (374 bytes)

CREATE src/app/myserv.service.ts (135 bytes)

Injectable and Dependency Injection

Here, the Injectable module is imported from the **@angular/core**. It contains the **@Injectable** method and a class called **MyserviceService**. We will create our service function in this class.

Myserv.services.ts

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

@Injectable({

providedIn: 'root'

})

export class MyservService {

student=[

{name:"kar",id:101},

{name:"deena",id:543}

]

constructor() { }

showmsg()

{

return "this is from service";

}

}

Invoking Services

App.module.ts

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

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

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

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

import {MyservService} from './myserv.service'; //import service

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

@NgModule({

declarations: [

AppComponent,HomeComponent

],

imports: [

BrowserModule,FormsModule

],

providers: [MyservService], //import same class in providers also

bootstrap: [AppComponent]

})

export class AppModule { }

App.component.ts

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

import {MyservService} from './myserv.service';

@Component({

selector: 'app-root',

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

styles: [``

]

})

export class AppComponent {

constructor(private s:MyservService)

{

}

ngOnInit()

{

this.s.student[0].name="siv"; //value changed

console.log(this.s.showmsg());

console.log(this.s.student);

}

}

The **ngOnInit** function gets called by default in any component created. The data is fetched from the service as shown above.

App.component.html

<app-home></app-home>

Sharing Data among Components using Services

A class constructor in Angular is mostly used to inject dependencies.

Angular DI inspects the constructor parameters and when Angular DI creates a new instance of a class (service, component, directive, pipe), it looks up matching providers to be passed to the constructor.

Therefore,injection only works for classed instantiated by DI

only constructor parameters are considered for injection

when Angular calls ngOnInit() it has finished intailazation thru constructors

[Home.component.ts](http://home.component.ts/)

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

import {MyservService} from './../myserv.service';

@Component({

selector: 'app-home',

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

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

})

export class HomeComponent implements OnInit {

aaj

constructor(private s:MyservService) { }

ngOnInit() {

console.log(this.s.showmsg());

console.log(this.s.student);

}

}

[Home.component.html](http://home.component.html/)

<p>

home component is also changed

</p>

**Introduction to Routing**

Routing basically means navigating between pages. You have seen many sites with links that direct you to a new page. This can be achieved using routing. Here the pages that we are referring to will be in the form of components.

A routed Angular application has one singleton instance of the Router service. When the browser's URL changes, that router looks for a corresponding Route from which it can determine the component to display.

creates route definitions, configures the router via the RouterModule.forRoot method, and adds the result to the AppModule's imports array.

Creating Routes

Each Route maps a URL path to a component. There are no leading slashes in the path.

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

let links:Routes= [

{

path:'home',

component:HomeComponent

},

{

path:'gallery',

component:GalleryComponent

}

]

@NgModule({

imports: [ BrowserModule,RouterModule.forRoot(links) ],

Routes is a typescript type of Route[], an array of individual Route instances.

The Routes type is an array of routes that defines the routing for the application. This is where we can set up the expected paths, the components we want to use and what we want our application to understand them as.

Each route can have different attributes; some of the common attributes are:

· *path* - URL to be shown in the browser when application is on the specific route

· *component* - component to be rendered when the application is on the specific route

· *redirectTo* - redirect route if needed; each route can have either component or redirect attribute defined in the route.

|  |
| --- |
| The redirectTo property describes the path we want to redirect this user to if they navigate to this  URL. |

· Now if the user visits the root (empty) URL they are redirected to /home instead.

·

· *pathMatch* - optional property that defaults to 'prefix'; determines whether to match full URLs or just the beginning. When defining a route with empty path string set pathMatch to 'full', otherwise it will match all paths.

· *children* - array of route definitions objects representing the child routes of this route

The path property describes the URL this route will handle.

· The component property is the name of the component we want to display when the URL in the browser matches this path.

const myrout:Routes=[

{

path:'',

redirectTo:'home',

pathMatch:'full'

},

{ path:'home',

component:HomeComponent

},

{

path:'gallery/:id',

component:GalleryComponent

},

{

path:"gallery/:name/:id",

component:GalleryComponent

},

{

path:'\*\*',

component:MsgComponent

}]

The **empty path** in the route represents the default path for the application, the place to go when the path in the URL is empty, as it typically is at the start. This default route redirects to the route for the /home URL and, therefore, will display the HomeComponent.

The \*\* path in the last route is a **wildcard**. The router will select this route if the requested URL doesn't match any paths for routes defined earlier in the configuration. This is useful for displaying a "404 - Not Found" page or redirecting to another route.

**The order of the routes in the configuration matters** and this is by design. The router uses a **first-match wins** strategy when matching routes, so more specific routes should be placed above less specific routes

We can also add a *catch all* route by using the path \*\*, if the URL doesn’t match *any* of the other routes it will match this route.`

We need to add a directive called router-outlet somewhere in our template HTML. This directive tells Angular *where* it should insert each of those components in the route

We place <router-outlet> where we want the component inserted.

Router Link

· The routerLink directive tells the router where to navigate when the user clicks the link. Add links to routes using the RouterLink directive.

Alternatively, you can navigate to a route by calling the navigate function on the router:

· this.router.navigate(['/component-one']);

·

Route Parameters

To create a route with parameter include a FORWARD SLASH, a COLON and a place holder for the parameter. The example below, creates a route with parameter id.

{

path:'gallery/:id',

component:GalleryComponent

}

You can have more than one parameter in a route. The following route definition creates a route with 2 parameters - id and name.

{

path:"gallery/:name/:id",

component:GalleryComponent

}

Activating the route with parameters : One way to activate the route is by using the routerLink directive.

<h1>routing</h1>

<a [routerLink]='["home"]'>Home</a>

<a [routerLink]='["gallery",1]'>gallery</a>

<a [routerLink]='["gallery","john",101]'>name</a>

<router-outlet></router-outlet>

we are binding routerLink directive to an array. This array is called link parameters array. The first element in the array is the path of the route to the destination component (gallery). The second element in the array is the route parameter, in our case the Id.

Introduction to Animations

The difference with Angular 4 is that animation is no more a part of the **@angular/core** library, but is a separate package that needs to be imported in **app.module.ts**.

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

The BrowserAnimationsModule needs to be added to the import array in app.module.ts

Trigger, state, transition

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

Here we have imported trigger, state, style, transition, and animate from @angular/animations.

we will add the animations property to the @Component () decorator

Trigger defines the start of the animation. The first param to it is the name of the animation to be given to the html tag to which the animation needs to be applied. The second param are the functions we have imported - state, transition, etc.

The **state** function involves the animation steps, which the element will transition between. Right now we have defined two states, first and large .For first state, we have given the style transform:'translateX(45px) scale(1)'and transform:'scale(4) translateX(70px)

Transition function adds animation to the html element. The first argument takes the states, i.e., start and end; the second argument accepts the animate function. The animate function allows you to define the length, delay, and easing of a transition.

Unit testing

Introduction to Unit Testing

When creating Angular projects using using the Angular CLI it defaults to creating and running unit tests using Jasmine and Karma.

Whenever we create files using the CLI as well as creating the main code file(home.component.ts) it also creates simple jasmine spec file named the same as the main code file but ending in .spec.ts, home.component.spec.ts

Creating Test Cases

describe('myeg test',()=>

{

it('should have values same',()=>

{

expect('hello').toEqual('hello');

});

it('should have value greater than',()=>

{

expect(45).toBeGreaterThan(45);

});

})

Myeg test-test suite which consists of individual test specs

'should have values same'-test specification

Expect-used with prebuilt matcher given by jasmine

Executing Test Cases using Karma

To run all the tests in our application we simply type ng test in our project root.

This runs all the tests in our project in Jasmine via Karma.

It watches for changes to our development files, bundles all the developer files together and re-runs the tests automatically.

Jasmine is a javascript testing framework that supports a software development practice called [Behaviour Driven Development](https://en.wikipedia.org/wiki/Behavior-driven_development),attempts to describe tests in a human readable format

Jasmine tests by refreshing a browser tab repeatedly in different browsers every-time we edit some code can become tiresome.

Karma is a tool which lets us spawn browsers and run jasmine tests inside of them all from the command line. The results of the tests are also displayed on the command line.

Jasmine comes with a few pre-built matchers like so:

expect(array).toContain(member);

expect(fn).toThrow(string);

expect(fn).toThrowError(string);

expect(instance).toBe(instance);

expect(mixed).toBeDefined();

expect(mixed).toBeFalsy();

expect(mixed).toBeNull();

expect(mixed).toBeTruthy();

expect(mixed).toBeUndefined();

expect(mixed).toEqual(mixed);

expect(mixed).toMatch(pattern);

expect(number).toBeCloseTo(number, decimalPlaces);

expect(number).toBeGreaterThan(number);

expect(number).toBeLessThan(number);

expect(number).toBeNaN();

expect(spy).toHaveBeenCalled();

expect(spy).toHaveBeenCalledTimes(number);

expect(spy).toHaveBeenCalledWith(...arguments);

The ***constructor*** is a method in JavaScript and is considered as a feature of the class in es6 .When the class is instantiated it immediately runs the constructor whether it is used in Angular framework or not.So it is called by JavaScript engine and Angular has no control on that.

That is why there is ***ngOnInit*** lifecycle hook in Angular.ngOnInit renders when Angular has finished initialising the component.

constructor calls by JavaScript engine immediately when the class is instantiated (before calling ngOnInit by Angular), so typescript helps us to get the type of the dependencies are defined in the constructor and finally tells Angular what type of dependencies we want to use in that specific component.

**constructor()** is the default method in the Component life cycle and is used for dependency injection. Constructor is a Typescript Feature.

**ngOnInit()** is called after the constructor and ngOnInit is called after the first ngOnChanges.

i.e. Constructor()->ngOnChanges()->ngOnInit()

as mentioned above ngOnChanges() is called when an input or output binding value changes.

**ngOnInit** is purely there to give us a signal that Angular has finished initialising the component.

This phase includes the first pass at Change Detection against the properties that we may bind to the component itself - such as using an @Input() decorator.

Due to this, the @Input() properties are available inside ngOnInit, however are undefined inside the constructor, by design

The@Input communication mechanism is processed as part of following [change detection](https://blog.angularindepth.com/angulars-digest-is-reborn-in-the-newer-version-of-angular-718a961ebd3e) phase so input bindings are not available in constructor.

Behaviour subject has an intial value to be emitted when subscribed to

Rxjs offers different types of Subjects:

1.Behaviour Subject

2.Replay Subject

3.Async Subject

When many-to-one or one-to-many or many-to-many relationship communication between components is difficult

Using @Input or @Output.

So subject is used.

Observables allows to only subscribe

.whereas subjects (type of Observable) allows to publish as well as subscribe

Subjects can be used both as observable and observer at the same time

Subjects can push data and subject’s subscribers can receive that pushed data

Next() method to send messages to observable which are sent to all angular components who are all subscribers will get them

Behaviour subject is type of subject which is used to communicate back and forth between large number of components.

Behaviour Subject stores the current value so directly get the last emitted value from the Behaviour Subject

2 main methods:

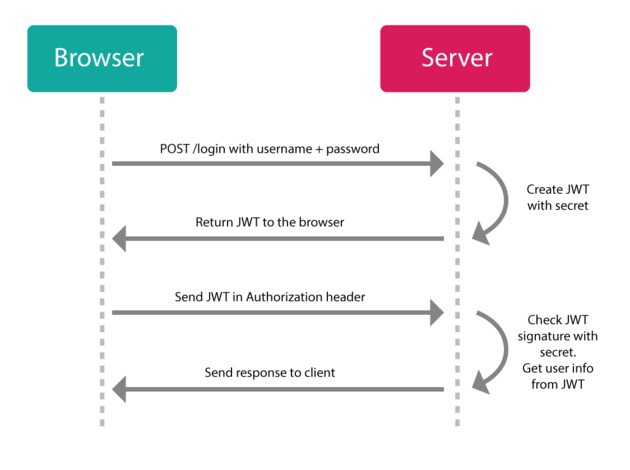
1.subscribe ()-for listening to new values.broadcasts out the value whenever there is a change

2.next()-for setting new values-passes a new value.

With the advent of Single Page Applications(SPA) and microservices, there is a need to look beyond the sessions. Any token based authentication serves that purpose. JWT is a type of token-based authentication. For every single request from a client to the server, a token is passed for authentication

* More trustworthy than cookie and sessions.

*A JSON Web Token (JWT) is a* [*JSON object*](http://www.w3schools.com/json/) *that is defined in* [*RFC 7519*](https://tools.ietf.org/html/rfc7519) *as a safe way to represent a set of information between two parties. The token is composed of a header, a payload, and a signature.*

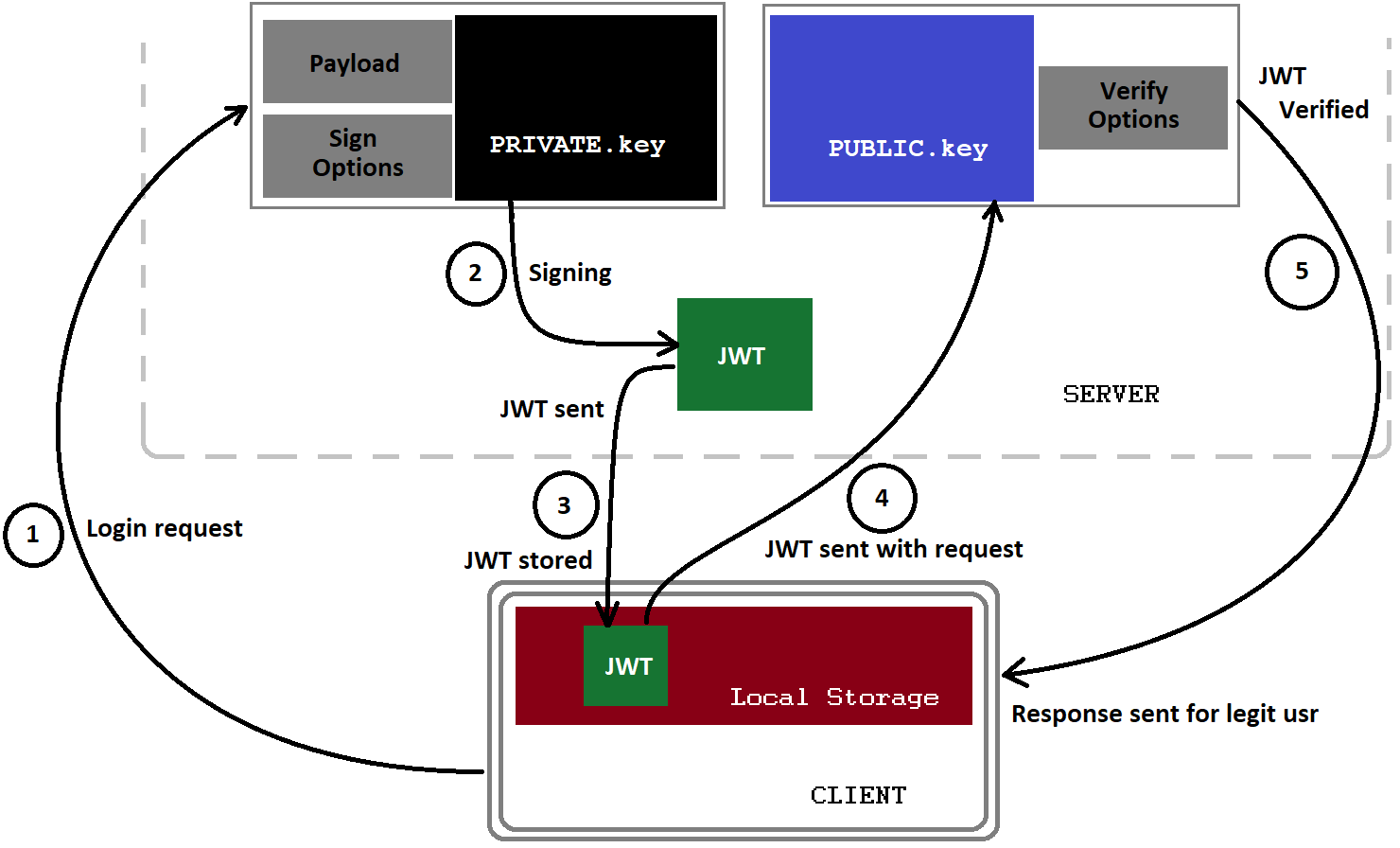


Token generation

jwt.sign(payload, secretOrPrivateKey, [options, callback])

payload is an object literal of key-value pairs that you would like to encode within your token.

we encode the user.id so that when we receive the token again on the back end for authentication, we know which user we are dealing with.



Refre in drive jwt node with angular