Angular Framework

Angular is used to develop single page application, it uses components to create reusable interfaces which can be added in any part of the page, it uses mainly 2 technologies

1. HTML

2. Typescript - super set of Javascript

ECMAScript new features

1. New types of keywords let, const, class, constructor, extends, super
2. Template Strings
3. Arrow functions
4. Object Destructuring
5. Exponential Operator
6. String padding
7. Optional Chain
8. Rest & Spread operators

Arrow functions:

It is to simplifying writing the callbacks

Callback functions are the functions which are executed later

() => “Hello” // returns String

() => console.log(‘hello”) // no return value

(x, y) => x + y; returns x + y

(x, y) => {   
 //statements  
 return (x + y);  
}

forEach(callback)

map(callback)

Rest & Spread operators

Rest operator accepts 0 or more arguments, it must be the last formal argument in the function

function sum(x, y, …z) { }

Spread operator is used to distribute the values to the multiple arguments of the function or variables

String padStart & padEnd

These are used to add some special characters or even regular characters before or after the strings

Suppose we have 45678 to that we need to add \*\*\*\*45678 (or) 45678\*\*\*\*

padStart() & padEnd() functions part of String

Static keyword

Static members you can access directly by using class-names

class X {   
 a = 10;  
 static b = 20;  
  
 static demo() {   
 // print some message  
 }  
 test() {   
   
 }  
}

let x1 = new X(); x1.test();  
X.demo();  
 x1.demo(); // this says undefined/not a function  
 // x1.b >> gives undefined  
// X.b >> works  
// X.a >> undefined  
// x1.a >> works  
// Access a & b outside the class

Optional Chain:

It is used to access properties with a condition whether its present or not.

let users = [ { firstname: “Raj”, email:”raj@gmail.com”}, { firstname:”Vijay”}, { firstname:}];

Callback: The action is initiated now but will be completed later, these are useful to perform asynchronous task, in Javascript there are many methods that take call-backs to perform asynchronous tasks like

setTimeout(callback, time), XMLHttpRequest.onreadstatechange

Promises: It is used to perform asynchronous task in a simplified way without using nested callbacks, a Promise can be successful or unsuccessful which are called as resolved / rejected, based on this Promise objects takes a callback with 2 arguments

Promise(callback(resolve, reject) { } )

When promise is resolved you can use the callback of .then() if its rejected then you can use callback of .catch()

Async and Await: These are keywords used in asynchronous operations, it waits for the result from the Promise.

Note: await must be used inside async function



Output:



Typescript:

It is a super set of Javascript which is more reliable than Javascript, it can identify early errors because it is first compiled.

TS -> compiled to -> JS

Installing typescript compiler

>> npm install typescript

[OR]

>> npm install -g typescript

How to compile typescript

>> tsc file.ts

>> tsc --target es6 file.ts

Problems with Javascript

* Javascript data is not reliable, means results are sometimes unpredictable, i.e., a + b can either add numbers or concatenate strings
* Javascript functions can return any kind of value there’s not restriction
* Javascript is not compiled, so you can’t find any errors until you execute

Typescript

* It supports all the features of Javascript
* It has additional features which are not supported by Javascript but it needs to be converted to Javascript to make browser to understand the code.
* It provides following features
  + Types to Variables
  + Types to Return types
  + Tuples, Union, enum types
  + Access modifiers like private, protected & public
  + Short-cut constructor initialization
  + Readonly properties

In Javascript

function add(x, y) { return x + y; }

add(2, 3) [or] add(“hello”, 123)

In Typescript we have types like string, number, [], boolean, any

function add(x: number, y: number) : number {   
 return ( x + y )  
}

Now if you call

add(20, 30); // it works

add(“Hello”, 123); // it raises compilation error

List of datatypes

void

number

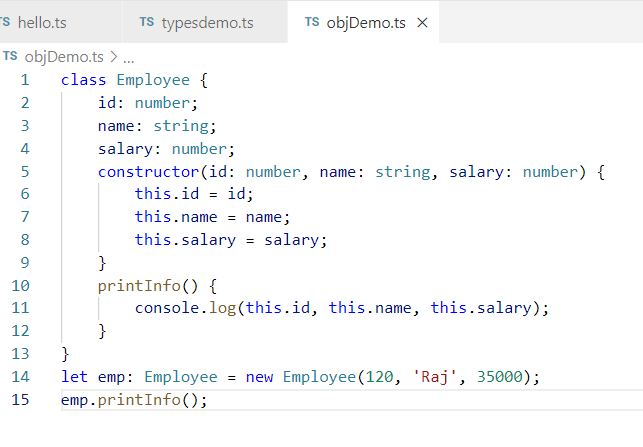
string

boolean

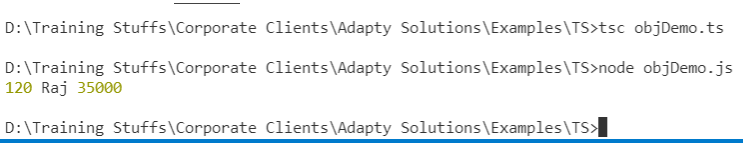
any

enum

objDemo.ts



Output:



Access Modifiers:

These are the keywords that determine the visibility of the members, there are 3 access modifiers

1. private
2. protected
3. public

private: Visible only within the class

public: Visible within the class & outside the class

protected: Visible within the class & only to the sub-class

class X {   
 private a: number;  
 protected b: number;  
 public c: number;  
   
 abc() {   
 can access this.a, this.b & this.c  
 }   
}   
  
class Y {  
 X x1 = new X();  
 can access only x1.c  
}  
  
class Z extends X {  
 // inherits b & c both

can access this.b & this.c  
}

Angular Framework

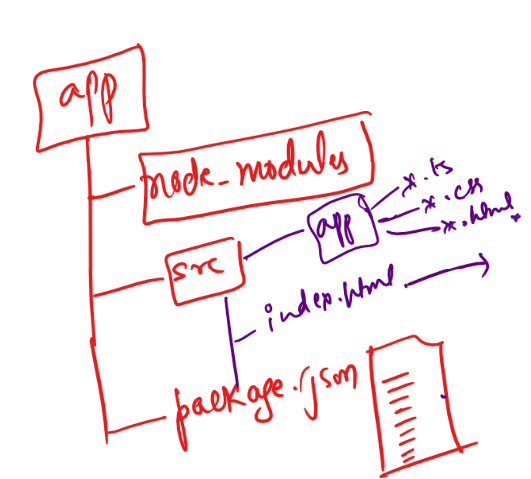
>> ng version : Shows the angular version

>> ng new app-name: Create angular project

Angular Framework is used to develop single page applications, Angular has provided an Angular CLI which is a toolkit which gives us below features

* Ready to run angular project
* Embedded Typescript compiler
* Embedded lite server
* Lived reloading features
* Commands that can quickly generate angular elements like components, routers, pipes, classes, services and so on

Angular Project Structure



index.html

This is the only file that will be loaded on the browser, all the components content will be part of this file

src: It will have all the application logic related components, services, routers and etc

package.json: This is a configuration file used by node, it will have entries about libraries, commands, project metadata

Components:

These are reusable templates which are part of the page, they are created with @Component({….}) mentioned on top of the class.

Modules:

It is a reusable element that represent either your entire app or part of the app, it can have all your components, services, routers and etc and can be loaded as an application

Generate components

We have a command to generate components

>> ng generate component componentName

[OR]

>> ng g c componentName

Angular Building Blocks

1. Components
2. Modules
3. Data binding
4. Pipes
5. Services
6. Routers
7. Directives

Directives:

NgFor: It is for iterating the arrays to generate the DOM elements

NgIf: It is for applying conditions to hide/show DOM

NgSwitch: It for applying multiple conditions using NgSwitchCase

These directives are used with \* in the beginning

<div \*ngFor = “expressions”>

<div \*ngIf = “expressions”>

<div \*ngSwitch = “expressions”>

arrayItems = [“Java”, “Python”, “Javascript”]

<div \*ngFor = “let item of arrayItems”>  
 {{item}}  
</div>

NgIf: Can be used for boolean type values as well as for properties that has value or undefined

Using bootstrap libraries

>> npm

install bootstrap --save

Pipes:  
These are used to format the output using | character, we have inbuilt pipes like

json, uppercase, lowercase, date, currency,

{{property | json}}

{{property | date}} >> {{user.birthday | date}}

{{property | date : ‘dd-MM-yyyy’}} >> {{user.birthday | date: ‘dd-MM-yyyy’}}

{{property | currency : ‘INR’}} >> {{ user.balance | currency : ‘INR’ }}

Custom Pipes

It is used to create our own pipes that can transform the data, you need to use @Pipe() decorator with a name i.e., @Pipe(name = “demoPipe”) and the class having @Pipe needs to implement PipeTransform and implement the transform(..) method

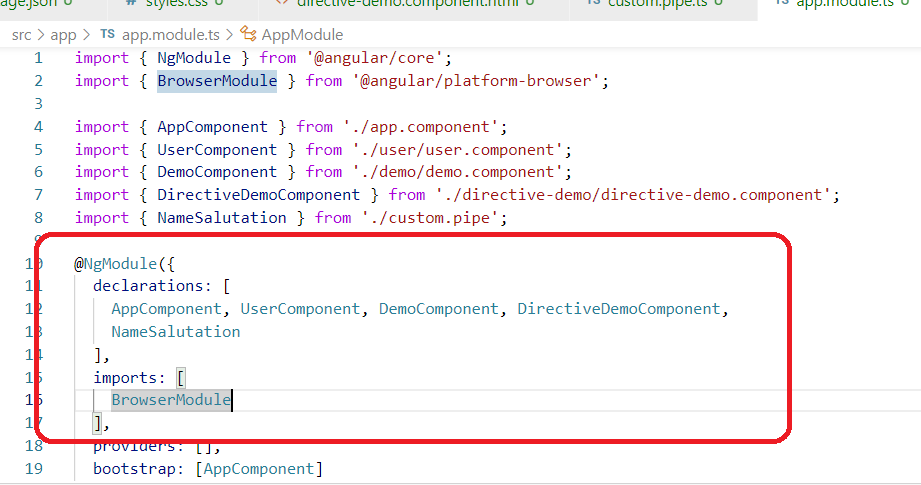
The transform(..) is automatically called when the pipe name is used

@Pipe(name = “demoPipe”)  
class Demo implements PipeTransform {  
 transform(input, arg) {   
 return ouput;  
 }  
}

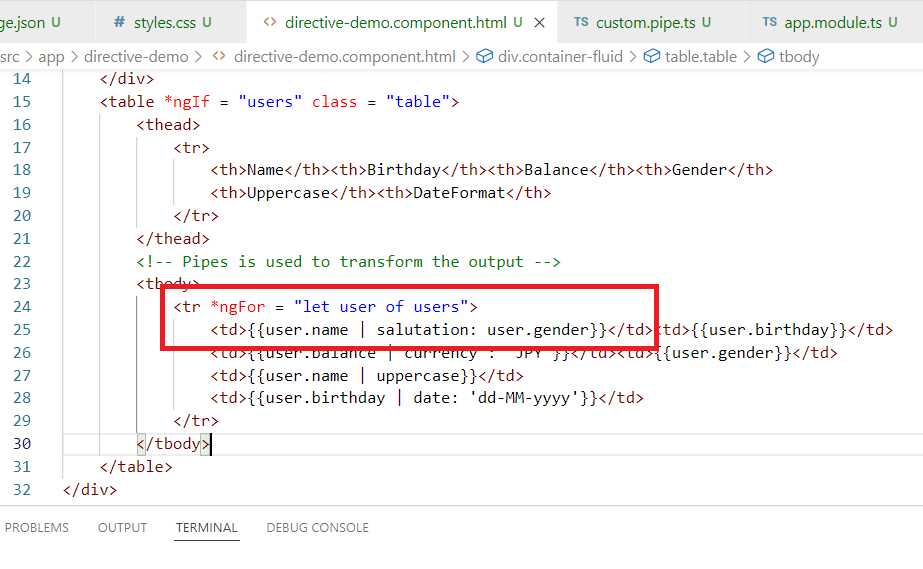
Note: Pipe you create must be part of the module hence you need to register in the AppModule i.e., app.module.ts



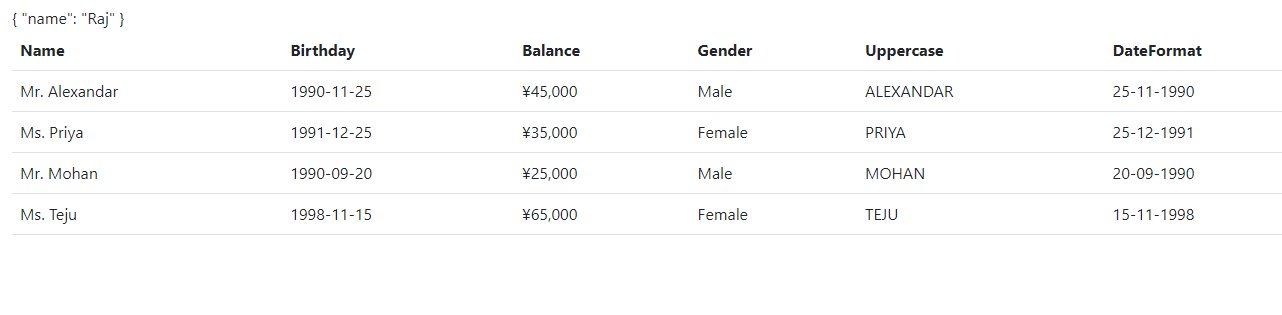
Add this pipe in the app.module



Using the pipe in the component template



Output:



Databinding:

It helps to share the data between the view template & the component class, there are mainly 4 types

1. Interpolation {{expr}}: From Component class to View - One way databinding
2. Event Binding (eventName): From View to Component class - One way databinding
3. Property Binding [propertyName]: From Component class to View - One way databinding
4. Two way databinding [(ngModel)]: In both directions we can share the data

Apart from these we have

* @Input & @Output: Data sharing between the components ie.., parent to child or child to parent
* Style binding: Type of property binding where you will have CSS styles
* Class binding: Type of property binding where you will have CSS class-name

Two way databinding is achieved using [(ngModel)] directive, it is by default not part of the angular module, hence we need to import ‘FormsModule’ in the AppModule

<input [(ngModel)] =”property”>

@Component({})  
class ComponentName{  
 …  
 isDisabled = false;  
 …  
 background = {“background-color”:”yellow”}  
   
   
}

HTML

<div [ngStyle] = “background”>content</div>

Note: ngStyle is not part of FormsModule

CSS

.redColor { color : ‘red’}  
  
TS

redColor = true;

HTML

<div [ngClass] = ‘redColor’>…</div>

@Input & @Output:

These decorators are used to share the data between the components from parent to child & child to parent.

@Input: It is for parent to child, here the child component property must have @Input()

@Output: It is for child to parent, here the child component property must have @Output(), through EventEmitter child can push the data to the parent.

ParentComponent

parentPropery = undefined

ParentComponentTempalte

<app-child (childProperty) = “parentProperty = $event” />

[or]

<app-child (childProperty) = “someFun($event)” />

Directives:

Directives are the functions which can perform DOM manipulations, we have predefined directives like

* Structural Directives: \*ngFor, \*ngIf, \*ngSwitch
* Attribute Directives: [ngStyle], [ngClass], [ngForm] and so on
* Component Directives: a class with @Component

@Directive() is the decorator used for creating custom directives, you can create directive attributes or tags using @Directive

For Tag: @Directive({selector: ”tag-name”})

For Attribute: @Directive({selector: ”[attr-name]”})

If you want these directives to work you need to use ElementRef to add some behaviour to the DOM

>> we have a command to create directive “ng g directive directive-name“

Angular Forms

Angular provides 2 ways to handle form data

1. Template Driven Form
2. Model Driven Form / React Form

Template Driven Forms:

It handles the form in the HTML template, it uses 2 way data binding to share the data between view & the components, you can perform validations in the HTML itself

Template Driven Forms uses FormModule to handle the form data, it uses a directive called [ngForm] which helps to manage entire form

Form Directives: These are provided by FormsModule to easily handle the form data, like [(ngModel)], [ngForm], The FormsModule should be part of the AppModule

<form #user = “ngForm”>  
 <input name = “fn” ngModel = “fn” />  
 <input name = “ln” ngModel = “ln” />  
</form>

Form Validations in the TDF

You can use inbuilt html attributes for validations like

required, minLength, maxLength, pattern,

<input required #tf1 = “ngModel”>

Here #tf1 takes a control object that will have properties like .value, .valid, .invalid, .pristine, .dirty and so on

Reactive Forms / Model Driven Forms

It is another type of Forms, it will have more control over the form controls because the controls are created in the typescript code, it provides classes like FormControl, FormGroup, FormBuilder, all these are part of @angular/forms library, it doesn’t need Form Directives like ngForm, ngModel

Creating a simple form control using reactive form

<input [fromControl] = ‘username’>

username = new FormControl(‘defaultValue’, validators);

FormGroup

It is used to group the from controls to make them part of the form

profile = new FromGroup({  
 firstname : new FormControl(‘’),  
 lastname: new FormControl(‘’),   
 dob : new FormControl(‘’)  
})

FormBuilder: It is a service that helps to create FormGroups in a simple way, this object is automatically created for your components through Dependency Injection

constructor(private \_builder : FormBuilder) { }

profile = this.\_builder.group({  
 firstname: [“”, Validators],  
 lastname: [“”, Validators],  
})

Dependency Injection:

Process where the dependencies are initialized by the framework instead of initializing in the code, the framework supplies the object to the dependencies

How to use Validators in the Form Builder

profileForm = this.\_builder.group({  
 firstname: [“”, Validators.compose(Validators,….)],  
 lastname ….   
})

Show error message for each controls when there’s a validation error & disable the submit button if form is invalid

Today’s agenda

* Routers
* Services & Dependency Injection
* Modules
* Calling Backend API’s using JSON-server
* Observables
* Lifecycle hooks: ngOnInit(), ngAfterViewInit()

Routers: To navigate from one component to another component we need routers

<router-outlet>: It is a placeholder for components that needs to be loaded dynamically

Routes: It provides path & component information, like for which path what component must be loaded

routes: Routes = [

{path: ‘login’, component: LoginComponent },

{path: ‘register’, component: RegisterComponent } ]

RouterModule: It is used to take care of performing routing

routerLink: It gives the URL for the routes

<a routerLink = “/login”>Login</a>

Router: It is used to navigate from one component to another programmatically just like routerLink, Router provides the path to the RouterModule, there’s a navigate() method in the Router to define the paths.

Router object will be supplied to the components using Dependency Injection feature, we need to provide the constructor with Router argument

constructor(private \_router : Router) { }

this.\_router.navigate([“success”]) : path is /success

this.\_router.navigate([“login”]): path is /login

Path Parameters or Route Parameters

These are dynamic paths which are passed in the URL that could be extracted using some API

ex: success/1, success/2, success/3 [or] success/Ajay, success/Brad

Values like 1, 2, 3, Ajay & Brad are dynamic values given to the same success, but the success must have a path parameter success/:name

{:name}: It is a place holder that can accept any value, to extract that value we need to use that name

success/Ajay

obj.subscribe(p => p.name)

ActivatedRoute: It is an object that is used while routing from one component to another component to subscribe to the paths

constructor(private \_activated: ActivatedRoute) { }

Angular Services:

Services are reusable classes with business logics like access the backend service it is created with @Injectable({}) which means the objects of service we don’t have to create because angular framework auto-creates the objects, which is supplied to their dependencies like components.

Command to create a service

ng g service service-name

@ViewChild & @ViewChildren

These are the decorators used to reference the child element(s) having some template reference.

<p #tf1>…..</p>

@ViewChild(“tf1”)  
childElement: ElementRef

<input #tf2 type = ‘radio” >   
<input #tf2 type = ‘radio’>

@ViewChildren(“tf2”)  
 childList : QueryList<ElementRef>

ngAfterViewInit() { // this is called when view is loaded to the DOM }

You can access the elements of @ViewChild or @ViewChildren inside ngAfterViewInit() or later

<child-app #cmp></child-app>

@ViewChild(“cmp”)  
childElement : ChildComponent;

Here you can get a reference of child component and can access child component members in the parent component

>> ng g c parent  
>> ng g c child

Calling parent component function from parent component



Child Routes

These are the routes which are part of the another routes

<router-outlet> Loads a component if that component also has <router-outlet> then it is called as child routes/nested routes

app.component.html

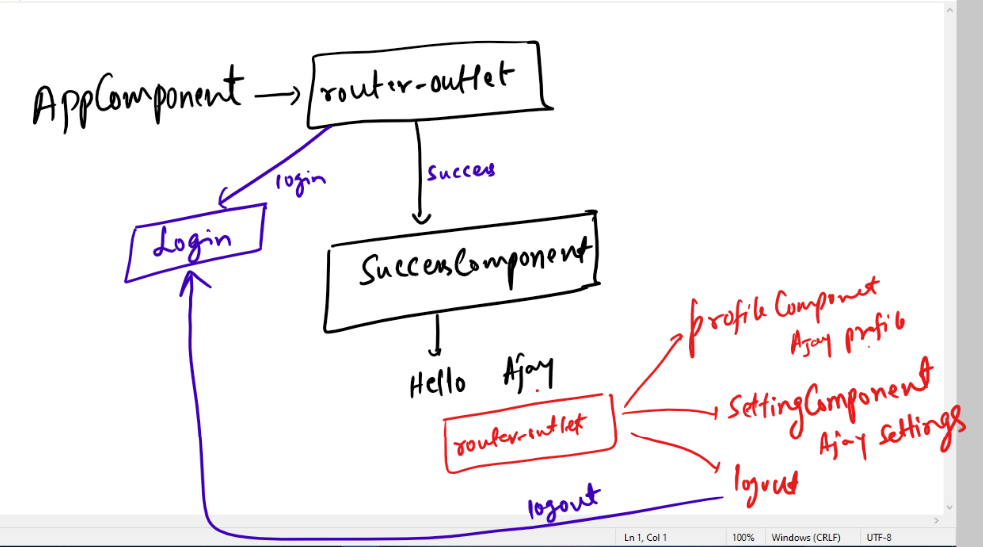
routerLink = login/register/success:user

<router-outlet>

success.component.html

routerLink = profile/home/logout

On profile you must show profile component within the success component only it must be loaded, similarly on home you load home component, on logout go back to login component.



Route Guards: These are the guards which will have some functions which are executed when you route to any components, these are also called as Auth Guards, you need to implement an interface called CanActivate which will have a method canActivate() that will be executed when you route to any components having guards,

You must configure guards in the routes

app-routing.module.ts

routes = [

{path : ‘success/:user’, component: SuccessComponent, canActivate: [CanActivateImpl], …. }  
]

/success/Ajay

Generate auth-guards using

>> ng generate guard authdemo