**Angular**

Angular is a component-based framework for building scalable web applications.

**CLI:**

CLI stands for **command line interface**. It is used to create components services, modules pipes etc in an easy way.

**Files and Folders of an angular project (Structure of angular project):**

1. **E2e folder (End to End) folder:**

This folder consists of files that are used to perform end – to – end tests. In these files we write files which perform end to end tests. These are automated tests that simulate a real user.

1. **Node modules folder:**

This folder consists of all the third-party libraries on which our application depends on. When we compile our application parts of these libraries are bundled together and deployed to our application. We are not going to deploy node modules folder to the server.

1. **Src (Source) folder:**

This folder consists of all the source files of our application. In this folder we have modules and components. Every application has at least one component and one module.

1. **Assets folder:**

In this folder we are going to store all our static files for the application.

Example: Image files, text files or icons etc.

1. **Environments folder:**

In this folder we store all the configuration settings for different environments. We have one file for the production environment and one file for the development environment.

1. **Favicons file:**

These are the icons displayed on our browser.

1. **Index.html:**

This file contain our angular application. It does not have any references to any styles or scripts. These references are inserted dynamically inserted into the file.

1. **Main.ts:**

This is a typescript file. This file is the starting point of our application.

1. **Polyfills.ts:**

This file imports all the scripts that are required to run angular. Because angular framework uses features of javascript that are not available in the current version of the javascript supported by most browsers out there. So this polyfills.ts file fills the gap between the features of js that angular needs and the features supported by the current browser

1. **Styles.css:**

This is where we add global styles for our application. Each page and each component can have its own styles.

1. **Test.ts:**

Used to set up the testing environment.

**Other files includes:**

Outside the source folder, we have

1. Configuration file for angular cli
2. Editor config file
3. Git file
4. Karma files that config the js. Karma is a test runner for js code.
5. Package.json:
   1. This is a standard file that every node project has.
   2. It has dependencies : The libraries that our application is dependent upon. Third party libraries that we use are also listed in the dependencies
   3. Dev.dependencise: We need these files to develop the application.
6. Protractor.conf.js: It is a tool for running end to end tests for angular.
7. Ts.config: This file consists of settings for our typescript compiler
8. Tslint.json: This includes settings for tslints. Tslint is a static analysis tool for typescript code. It checks typescript code for readability, maintainability and functionality errors.

Angular cli uses a tool called webpack, which is a build automation tool. It gets all our scripts and style sheets, combines and puts them in a bundle and minifies that bundle. This is for optimisation.

**Hot module replacement (or) Hot module reloading:**

This is a feature of the webpack which:

Whenever one of the source files is modified webpack automatically refreshes the browser.

**Angular Versions:**

1. AngularJS was the first version of angular and it was introduced in 2010. But the framework was not designed

**Features of Angular:**

1. Application state is stored in the component. Every application contains application state i.e., the data presented to the user or data entered by the user. It is stored in the form of properties or arrays as a part of components.
2. Component is a class that contains programmatic reflection of a user interface.

**Building blocks of angular:**

1. **Components:**

Components is a class that contains application data and event handler methods. For every screen you have to create a component. Components contains application data in the form of properties and events that are needed to be displayed in the view.

1. **Templates:**

The template contains design logic i.e., HTML logic to present output to the user templates can access all the properties and methods present in the component class by using Data Binding concept.

1. **Data Binding:**

Data bindings are special syntaxes used in the templates to create connections between HTML elements and component properties (or) methods.

1. **Modules:**

Modules are used to group related components. A module can import components **to (or) from** other modules.

1. **Services:**

Service is a TypeScript class which contains client side business logic which is involved in loading the data from the servers, making AJAX calls to REST API servers and also process the received data by performing custom and complex validations, calculations and also adding additional essential data to the data-structures.

Service doesn’t contain any view specific logic such as preventing keys that can’t be pressed in the text box (or) controlling the text box tab index etc

Business logic always be reusable and accessible among multiple components.

1. **Dependency Injection :**

It is a concept of creating service objects at runtime automatically and loading those service objects into the constructor of the components as per the necessity of the component.

1. **Directives:**

In general it is not recommended to write DOM manipulation code directly inside angular, it will be performed automatically when the application data is changed. But in order to make some plugins such as draggable, collapsible, model popups etc. You may need to interact with the attributes of HTML tags directly.

So it is allowed to write such DOM manipulations only in the directives. But not in other such as components, services etc.

**Angular Packages:**

1. **@angular/core:**

Provides essential pre-defined decorators, classes, interfaces and modules that are needed to run every angular application.

Ex: @Component, @NgModule, @Pipe, @Directive, @Injectable, @Injec, NgZone, OnChanges, OnInit, ApplicationModule etc.

1. **@angular/common:**

Provides build-in directives that are useful for most of the real-time applications.

Ex: ngIf, ngSwitch, ngClass, ngFor etc.

1. **@angular/compiler:**

It compiles “ templates “ ( html code ) into “ JavaScript code “, which produces DOM directly at runtime in the browser.

But angular developer never invokes this directive directly. It is always invoked automatically by Angular Cli package ( or ) by Platform Browser Dynamic package.

1. **@angular/platform-browser-dynamic:**

It invokes the angular compiler ( JIT compilation ) and specifies the start-up module and also start executing he application.

1. **@angular/platform-browser:**

Provides a set of pre-defined classes that are related to DOM ( manipulations ) and browser interaction.

Ex: BrowserModule.

1. **@angular/forms:**

Provides necessary pre-defined classes that are related to ( or ) needed to create and execute angular forms.

Ex: FormsModule, ReactiveFormsModule, Validators, ngModel, ngForm etc.

1. **@angular/router:**

Provides necessary pre-defined classes that are needed to create and execute angular routes.

Ex: RouterModule, Routes, ActivatedRoute, CanActivate, routerLink etc.

1. **@angular/animations:**

It provides necessary pre-defined classes hat are needed to create and execute angular animations.

Ex: BrowserAnimatinosModule, animate, state, style, transition etc.

1. **@angular/cli:**

Provides necessary pre-defined commands hat are needed to create, compile, build, add items in angular applications.

Ex: ng new, ng serve, ng build, ng test etc.

1. **rxjs:**

It provides necessary pre-defined classes for creating Observables, which are needed to represent the response of REST-API calls of AJAX.

Ex: Observable, Observer, Subject etc.

1. **zone.js:**

Provides necessary pre-defined classes for executing “ change detection processes “, while executing angular app.

**Dependency Injection :**

Dependency Injection (DI) create singleton service objects and serves them to components whenever needed.

The purpose of Dependency Injection is to load service objects into components.

**To create a new Angular workspace:**

Creating project:

ng new < project\_name > --style = scss –routing

**--style=scss –** is used to specify which type of CSS to be used.

**--routing -** is used to enable the routing.

**To compile and run the project:**

**ng serve –open**

If we user – open then the file will be compiled and opened in the browser window with url:

localhost:4200

if we don’t use - -open then we need to enter the url manually.

*We can specify our own port number by:*

*ng serve - -open - -port = <port number>*

**(for windows the port number varies between 1024 to 65535.)**

**Flow of Execution:**

**Index.html 🡪 main.ts 🡪 app.module.ts 🡪 app.component.ts**

**app.component.html**

**-------- freeCodeCamp.org --------**

**To create a new empty workspace (without source folder):**

Ng new <workspace name> --create-application = false

**To create a new angular application:**

ng g app <application name>

**Refer for difference between workspace, library, application, module:**

https://www.stackchief.com/blog/Angular%20Module%20vs.%20Library%20vs.%20Application%20vs.%20Workspace

**Workspace:**

Angular 6 introduced the concept of a workspace. A workspace is really just a collection of different projects. The ng new command creates a new workspace...

ng new myworkspace

This will create a new workspace named myworkspace.

**Library:**

A library is a collection of components, services, directives etc. that can be shared across different Angular projects.

You generate a new library just like you would a component, service or anything else:

ng g library mylibrary

**Library vs Application:**

Unlike an application, a library doesn't generate index.html, styles.css, polyfills.ts, main.ts. These files are specific to deploying an application in the browser. Since a library is simply a collection of reusable components, it doesn't need these files.

Additionally, a library generates a few files that an application doesn't.

**Module:**

You'll probably notice that both libraries and applications generate a @NgModule class defined in .module.ts file.

So what's the difference? Are both considered modules? What IS a module in Angular exactly?

A module is simply a way of organizing pieces of functionality together. While libraries and apps have their differences, they both represent collections of components, services, directives. A module is simply a way of defining how these elements work both internally and externally.

Modules both import and export functionality.

**Files in angular workspace:**

**Tsconfig.spec.json:**

It generates all the unit test files named as spec.ts This tsconfig.spec.json file will be used to compile . It extends tsconfig.json file.

**Tsconfig.app.json:**

It compiles all the type script files that are created inside the application. It extends tsconfig.json file.

**Package.json:**

This file contains the information about all the package files that we are using.

It contains dependencies and devdependencies:

**Dependencies:** all the files that are going to production purpose will be on dependencies

**Devdependencies:** all the files that are only for developmental purpose will be on devdependencies.

**Lock.files:**

Lock files is used to keep all the libraries used for the project under same version. If someone uses other version files, then they have to commit them, and lock files need to be updated as well.

**Karma.config.js:**

It is config file or a test runner which comes default in Angular. It is for unit testing.

All test cases are written in jasmine and they are run using karma.

**Angular.json:**

This file contains information about the workspace being used and about each project inside that workspace.

**Node\_modules:**

All the packages installed will be under this folder.

**Src:**

We are going to write our project code inside the src folder.

**Polyfills.js**

This file makes sure that all the code is backward compatible with all the browsers. That means it makes sure that all the code that we are written in our project is supported by all the browsers.

It will always add some extra bundle (code) to your final code to make it compatible.

**Zone.js:**

Zone.js patches a lot of code and features which is not supported by browsers and use internally by angular.

**Main.ts:**

This is where the execution of our application starts

**Index.html:**

This is the main html page which will be served to the user. Because angular is an SPA (Single Page Application).

**Assets:**

Contains all the static files i.e., the images, pdfs, documents etc.

**App FOLDER:**

This is the folder where we are going to write our code.

**Introduction to mono-repo:**

1. Create and maintain multiple apps in the same repo.
2. Use libraries within the project.
3. Deploy multiple apps/libs from the same repo.
4. Easy to share code within the project.

**To bootstrap our application, we need at-least one ROOT module.**

*All the modules should be specified under the imports section and all the pipes, components, and directives should be mentioned under declarations section.*

**Components:**

Components are views which will be rendered to the end user. Like root-module we also have a root-component. And this root component is loaded on to the index.html file. In future if we want to load any other files, we can load them within the app component files.

**To create a new Component:**

Ng g c <component name>

(or)

Ng generate component <component name>

**Binding Syntax:**

There are three types of binding syntaxes to bind information in your ts file to html file. They are:

1. Interpolation
2. Property Binding
3. Event Binding

1. **Interpolation :**

You can use interpolation syntax with any basic data types such as date, string, number that information can be binded using interpolation. You can also work with objects by using interpolation.

To use interpolation we use {{ }} syntax. We write the variable name i inside those double curly brackets and place it inside our html. TO display that information.

Some class {

Name = “psh”

}

In html file:

<h1> Hello {{ Name }} <h1>

Output: Hello psh.

1. **Property Binding:**

Property binding refers to binding data in our ts file with any valid html tag attribute. This is called property binding. We use [ ] bracket to use property binding. Angular allows you to use native html attribute properties for binding as well.

Ex:

Some class

{

Number\_of\_rooms = 10;  
}

Some html:

<p>Number of rooms: </p>

<div [innerText] = “Number\_of\_rooms”></div>

Output:

Number of rooms:

10.

1. **Event Binding:**

To add events to our html tags we can use event binding. We use ( ) brackets to use event binding. And inside those brackets we write out event handlers. Ex: click etc.

Some class {

isHidden = false;

Number\_of\_rooms = 10;

Toggle()

{

this.isHidden = !this.isHidden;

}

}

Some html:

<div [hidden] = “isHidden”>

<p>Number of rooms: </p>

<div [innerText] = “Number\_of\_rooms”></div>

</div>

<button (click) = “toggle()”>Click to hide </button>

Outpu:

First it is visible. But when you click then the div becomes invisible. And for the third click it is visible.

**Directives:**

Directives are used to change the behaviour and appearance of DOM elements.

Directives can implement all lifecycle hooks.

Directives cannot have template.

**Null collision operator:**

Null collision operator is used to display some other information if the other information is null or not available.

We use interpolation to do null collision.

{{ rooms?.availableRooms ?? “no rooms” }}

<div \*ngif = “rooms?.availableRooms>

Rooms List

</div>

In the above example if available rooms are not there (or) the value is null then “no rooms” will be displayed. Else “no rooms” will not be displayed and then the other logic will work.

Here rooms?availableRooms is used with ? which means optional parameter.

**Types of Directives:**

1. **Structural Directives:**
2. **Attribute Directives**
3. **Structural Directives:**

Structural directives are always costlier because they are going to change the behaviour of the DOM which may give performance issues as well.

Any attributes which can modify the DOM are called structural directives. They have an asterisk ( \* ) in the starting of their name.

1. **Attribute Directives:**

Attribute directives does things such as adding attributes to the elements or removing attributes from the element or add some extra logic to your element based on some properties.

They can modify the DOM. But they cannot add or remove elements from the DOM like structural directives.

1. **Built-in Directives:**

All the structural directives have “ \* “ in the naming convention.

\*ngIF and \*ngFor are Structural directives. They can modify the DOM. Which is adding or removing the view elements.

* 1. **\*ngIF:**

It is just like a if condition. The code written inside it is performed when the condition becomes true.

Ex:

<div \*ngif = “rooms.availableRooms > 0>

Rooms List

</div>

The div will be displayed if above condition is true. Else the div will be completely removed from the DOM structure.

* 1. **\*ngFor:div>**

Just a normal for loop.

<div \*ngFor = “let room of roomList; let e = even; let o = odd; let i = index “>

<td>{{ i }}</td>

<td>{{ e ? “Even” : “Odd” }}</td>

<td>{{ room.roomNumber }} </td>

<td>{{ room.price }} </td>

<td>{{ room.amenities }} </td>

<td>{{ room.roomtype }} </td>

<td>{{ room.checkinTime }} </td>

</div>

Now the for loop will iterate through the roomList array of objects and print the value inside the properties of those objects.

If there is any change inside the \*ngFor data then it will re-render all the data present in it.

* 1. **\*ngSwitch:**

Just like normal switch case.

Ex:

<div [ngSwitch] = “role”>

<div \*ngSwitch = “ ‘user’ “>

Welcome User

</div>

<div \*ngSwitch = “ ‘admin’ “>

Welcome Admin

</div>

<div \*ngSwitchDefault>

You are not authorized to view this page!

</div>

</div>

* 1. **ngClass:**

ngClass is used to add CSS classes directly to the elements. It is a Attribute Directive.

Ex:

Some CSS:

.even

{

Background-color: red;

Color: white;

}

.odd

{

Background-color: green;

Color: white;

}

Some HTML:

<div [ngClass] = “e ? ‘even’ : odd” \*ngFor = “let room of roomList; let e = even; let o = odd; let i = index “>

<td>{{ i }}</td>

<td>{{ e ? “Even” : “Odd” }}</td>

<td>{{ room.roomNumber }} </td>

<td>{{ room.price }} </td>

<td>{{ room.amenities }} </td>

<td>{{ room.roomtype }} </td>

<td>{{ room.checkinTime }} </td>

</div>

The above classes in the CSS and HTML based on the e value. If odd Odd classes will be set and if even Even class will be set.

* 1. **ngStyle:**

By using ngStyle we can apply CSS directly to the elements.

<div [ngStyle] = “{ ‘color’ : rooms.availableRooms ? ‘green’ : ‘red’ }”

[ innerHTML ] = “numberofrooms”>

</div>

**PIPES:**

Pipes are used for data transformation. They change the data at the time of presentation to the user.

They don’t change actual objects.

It is represented by using “ | “ (pipe) symbol.

**Built-in Pipes:**

1. **DatePipe:**

<div [ngClass] = “e ? ‘even’ : odd” \*ngFor = “let room of roomList; let e = even; let o = odd; let i = index “>

<td>{{ i }}</td>

<td>{{ e ? “Even” : “Odd” }}</td>

<td>{{ room.roomNumber }} </td>

<td>{{ room.price }} </td>

<td>{{ room.amenities }} </td>

<td>{{ room.roomtype }} </td>

<td>{{ room.checkinTime | date : “short” }} </td>

</div>

Changes the date to DD/MM/YY by default. We can specify the particular format by using : “ <format>“.

1. **UpperCasePipe:**

Used to convert to all uppercase.

1. **LowerCasePipe:**

Used to convert to all lowercase.

1. **CurrencyPipe:**

Used to display currency symbol before the numbers. Default is dollars. You can specify format using <data> : <format>

1. **DecimalPipe:**

To show data in form of Decimal.

Syntax: <data> | number : ‘ 1.1-2’

<How many total numbers> . <how many digits before decimal> - <how many digits after decimal>

If: 1.0-0 Then the decimal number is rounded to its nearest integer value.

We can also specify the currency based on the country like:

<data> | number : ‘ 1.1-2’ : ‘en-us’ // US numbers

1. **PercentPipe:**

To show data in form of percentage.

1. **JsonPipe:**

Used to show the JSON data inside a JSON file.

Syntax: <data> | json

It can be used for array, and it can be used for object files.

It is recommended to use this in debugging purpose not in production purpose.

1. **SlicePipe:**

Like normal slice. You can specify the index values up to which you want to show the data.

Syntax: <data> | slice : <start index> : <end index>

1. **AsyncPipe:**

**Life Cycle Hooks:**

Component instance has lifecycle hooks which can help you to hook into different events on components.

Lifecycle ends when component is destroyed.

In angular for every lifecycle hook there is an interface.

Make sure that you implement the interface whenever you are trying to access some of the lifecycle hooks of your component.

There is also a constructor which is available. Before your ngOnInit() event gets called because your component needs to be initialized. When page is being rendered on to the view a that time the constructor is called. Just after the constructor is finished your ngOnInit() lifecycle hook will be called. Constructor should be used only when you want to inject some services and Lifecycle hook is where you should write your code. Constructor should not have any blocking code. That code should be in your lifecycle hook.

1. **ngOnChanges:**

ngOnChanges() is related to some part of component communication.

ngOnChanges() can be applied to a component which has @Input property. We cannot apply ngOnChanges() anywhere else.

You can only apply on a component or a directive which has the @Input property.

And It only works if the data inside is being updated with a new value everytime.

Export class RoomsListComponent implements OnInit, OnChanges

{

@Input() rooms : RoomList[ ] = [ ];

@Output() selectedRoom = new EventEmitter<RoomList>();

Constructor();

ngOnChnages(changes: SimpleChanges):void

{

}

// Here SimpleChanges is an interface. It has a property name and

Some meta data. It will give us some information like previous value, current value, and whether it is the first change to the component or not.

ngOnInit(): void  
 {

}

selectedRoom (room : RoomList)

{

This.selectedRoom.emit(room);

}

}

1. **ngOnInit:**

ngOnInit() contains some code which should work, just after initializing your component.

That part of code should be written inside ngOnInit() lifecycle hook.

If you load some data from your API and display that data on your view, you should write that part of your code in your ngOnInit() lifecycle hook. The data will be rendered just after the page gets loaded.

1. **ngDoCheck: [Not recommended to use commonly]**

**(refer component communication in below section)**

This onChanges event is something which is executed every time we raise any event irrespective of where the component is implemented or available.

Incase it is active it will listen to any change that has happened inside your entire application.

Implementing this lifecycle hook is very costly (may be: Because it checks and runs for every event.

You should not implement ngOnChanges and doCheck together on the same component

**The Reason is :** Both are going to do the same work.

1. **ngAfterContentInit:**
2. **ngAfterContentChecked:**
3. **ngAfterViewInit:**

**(refer view child on below sections)**

1. **ngAfterViewChecked:**
2. **ngOnDestroy:**

**Change Detection in angular:**

Whenever we do an action the view needs to be updated. Angular does it via change detection.Change detection is internally handled by angular, but there are multiple ways you can actually change the change detection for your component.

To apply specific change detection properties we have define it in the

@component class in the ts file.

@Component

{

……

changeDetection: changeDetectionStrategy.OnPush;

}

Before you go ahead and use OnPush there are certain requirements that have to be met.

OnPush change detection strategy can only be applied in case we’re not modifying some data internally in this component.

We can achieve this by using:

1. @Input and @Output
2. Incase we use some state management technique

Ex: ngx, or any other state management library. In that can we can also move it to onpush.

1. The property that are passing should be **immutable.**

So we cannot directly push the data. It won’t work. We have to return a new instance.

Ex: this.roomList = [ …roomlist, room ];

Inside ngOnInit:

selectRoom(room: RoomList)

{

This.selectedRoom.emit(room);

}

**Component Communication:**

The scenario where two or more components need to interact is known as component communication.

There are multiple ways to achieve component communication.

**Ways for Component Communication:**

1. Using @Input and @Output.
2. Using @ViewChild and @ContentChild
3. Using services
4. **@Input and @Output:**

These are used to communicate between two components.

@Input() rooms : RoomList [] = [];

This takes input of RoomList and assign the value to rooms variable.

To display the data in RoomList component we need to use our own tag related to that component name.

<hinv-rooms-list [rooms] = “roomList”></hinv-rooms-list>

@Output() selectedRooms = new EventEmitter<RoomList>();

If you want to send data from child component to child component then we should use @Output.

This is an event, so we should use EventEmitter class which is available in angular core. Then we have to tell what type of data we are going to send to the parent inside < > brackets.

In Child:

<hinv-rooms-list [rooms] = “roomList” (selectedRoom) = “selectRoom($event)”></hinv-rooms-list

*(dollar event)*

<td>

<button (click) = “selectRoom(room)”>Select

</td>

In parent:

selectRoom(room : RoomList)

{

This.selectedRoom.emit(room);

}

**ViewChild:**

It is used to access component inside another component which does not have any @Input and @Output property.

**AfterViewInit:**

AfterViewInit lifecycle hook is called after the initialization of the view. That is being called inside the AfterViewInit. If you are called your header inside the rooms component it should complete its initialization first.

So if we write header calling inside the AfterViewInit inside the rooms.ts file. Then the AfterViewInit makes sure that the rooms component calls the header after the initialization of the header. So, the header will be available to access for the rooms component.

**AfterViewChecked:**

A callback method that is invoked immediately after the default change detector has completed one change check cycle for a component’s view.

**ng-template:**

ng-template is a tag which never render anything. But it can help us to render some other template (or) component. ( It can be used to load components dynamically ).

**---------------------------------------------------**

**RSK Helpline:**

**Template reference Variables:**

Template reference variables are used to capture the values of an element. We write template reference variables using “ # “ symbol followed by the name of the variable.

Ex:

<input #msg type = “text” >

<button (click) = “message(msg.value)”></button>

Here when the button is clicked it calls the message function. And passes input tag’s value as its parameter.

**Two-way data binding:**

In two way data binding the data from the class is sent to view and also the data from the view is sent to the class.

We can achieve two-way binding by using ngModel directive. To use ngModel directive we have to import it from the forms module. ‘@angularForms’.

[ ] ---- > for property binding

( ) ---- > for event binding

Two-way binding = property binding + event binding

= [ ] + ( )

= [ ( ) ] This symbol is used for two-way binding.

Ex: in html:

<input [(ngModel)] = data type = “text”>

<button type = “button”>Click me</button>

{{ data }}

In Class:

public (not necessary to use public) data : string = “”;

Output: When you click the button the value inside the input is sent to the class and it is updated to the variable. Then the value inside the data variable is shown in the html using interpolation.

Note: to use ngModel import the forms module in app.module.ts and include the name in imports section.

**Structural Directives:**

1. **ngIf:**

Like normal if conditional statement. It works when its true.

Ex: <h1 \*ngIf = “status; else elseBlock”> Welcome Mr> {{ name }} </h1>

<ng-template #elseBlock>

<h1> You are not authorised! </h1>

</ng-template>

In class:

status : boolean = false;

In the the code if the status is true then the **welcome** is displayed, but if its false then immediately else block will be executed. And it shows you are not authorized. Else block should be declared using the ng-template component with template – variable.

**(or)**

<h1 \*ngIf = “status; then trueBlock else elseBlock”> Welcome Mr> {{ name }} </h1>

<ng-template #trueBlock>

<h1> Welcome {{ name }} </h1>

<ng-template>

<ng-template #elseBlock>

<h1> You are not authorised! </h1>

</ng-template>

In the above code we have two blocks they are true block and also the else block. We can use **then** to display the desired block if the statement is true by using the template variables.

**ngSwitch:**

Just like a normal switch case.

<div [ngSwitch] = “mychoice”>

<div class = “switchCase”>

<div \*ngSwitchCase = “ ‘ one ‘ ”> First block<div>

<div \*ngSwitchCase = “ ‘ two ‘ ”> Second block<div>

<div \*ngSwitchDefault> Invalid choice<div>

</div>

</div>

In the above code the respective div will be displayed based on the value in the switch case. The comparison value in the \*ngSwitchCase = “ ‘ ‘ “. Should be given within the single quotes. If all cases are not matched then the default case will run.

**\*ngFor:**

It is just like a normal for loop.

<div \*ngFor = “ let x of subects ; index as i“>

<h1> {{ i }} ---- > {{ x }} </h2>

</div>

Here “ i “ is used as index. Index always starts from 0.

<div \*ngFor = “ let x of subects ; first as f“>

<h1> {{ f }} ---- > {{ x }} </h2>

</div>

In the above code first as f means for first element of the loop it is displayed as true and for all the remaining elements it is displayed as false.

Also:

Even as e is used to display true for even indexes and false for odd.

Odd as o is used to display true for odd indexes and false for even indexes.

**Component Interaction:**

Component interaction is done using @Input() and @Output().

@Input() is used to transfer data from Parent/App component.

@Output() is used to transfer data from Child/Test component.

**@Input:**

The properties that are defined in the parent class can be sent to the child class using @Input().

Parent class:

public parentMsg = “this is parent message”;

In parent html file:

<child-component-tag [fromParent] = “parentMsg”></child-component-tag>

Child class:

@Input() fromParent : any;

In child html file:

<h1> {{ fromParent }} </h1>

Output:

This is parent message.

**@Output:**

This is used to emit event that contains information from child component to the parent component.

In child html :

<button (click) = “childMsgFunc()”>Click me</button>

In child ts:

@Output childMsg = new EventEmitter();

childMsgFunc()

{

this.childMsg.emit(“This message is from child”);

}

In parent html:

<child-component-tag (childMsg) = “msg=$event” [fromParent] = “parentMsg”></child-component-tag>

In parent ts:

msg : any;

In the above code, in the parent html file the child tag from where the event is coming from, we have to write an event whose name is the same name of child event emitter variable name. Then in double quotes we have to declare a variable to store the data after that name without any spaces we have to use “ = “ symbol and after that without any space we have to specify “ $event “. The $event is used to capture the data emitted by the child.

**Pipes:**

Pipes are used to transform incoming stream of data into required format before displaying it in the view.

In ts file:

msg : string = “Message”;

person : object = {

“firstName” : “Prashanh”.

“lastName” : “Bocha”

}

In html file:

The various pipe operations include:

1. lowercase
2. uppercase
3. titlecase
4. slice -------- <start Index (including)> : <end index (excluding)>
5. json
6. number
   1. ‘ <max number of integers> . <min number of digits after decimal> - <max number of digits after decimal>,
7. Percent
8. Currency ----- use colon and then, currency code of country ex: | currency: ‘ INR ‘
9. Date: In general it gives full format gmt time date etc but using this
   1. Short ------- DD/MM/YY, Time AM/PM
   2. ShortDate ------- DD/MM/YY
   3. shortTime ------- Time AM/PM
   4. medium --------- Oct 11, 2020, 11:19:29 AM
   5. mediumDate ------- Oct 11, 2020
   6. long -------- August 21, 2001 at 11:20:21 AM GMT+5
   7. longDate ------ October 11, 2020
   8. longTIme ------ 11:11:29 AM GMT+5

Ex:

{{ msg | lowercase }} ------- Output: message

{{ msg | uppercase }} ­------- Output: MESSAGE

{{ msg | titlecase }} ­------- Output: Message

{{ msg | slice : 3:6 }} ­------- Output: sag

{{ person | json }} ­------- Output: represented in json format

{{ 5.678 | number : ‘ 1.2-3 ’ }} ­------- Output: 5.678

{{ 5.678 | number : ‘ 3.4-5 ’ }} ­------- Output: 0005.6780

{{ 5.678 | number : ‘ 13.1-2 ’ }} ­------- Output: 005.68

**Services:**

Services are used to:

1. Share data
2. Implement application logic
3. External interaction

Naming convention is : xxxx.service.ts

**Dependency Injection:**

1. Without dependency injection
2. Dependency injection as design pattern
3. Dependency injection as a framework (provided by angular).
4. **Let us consider a class that doesn’t have dependency injection:**

Class Engine

{

Constructor() { }

}

Class Tires

{

Constructor() { }

}

Class Car

{

Engine;

Tires;

Constructor()

{

This.engine = new Engine();

This.tires = new tires();

}

}

In the above code we used the engine class and tires class in car class. And to use them we had to create new instances of them.

But if the engine class and tires class constructors need a parameter then we must change their implementation in the car class also.

So even if the car class doesn’t need those parameter we must pass them in order to implement the engine and the tires classes.

Our code is not flexible. Means that any time dependencies change -> Car class need to be changed as well.

Without dependency injection the testing process is also difficult. That means whenever we test the constructors with various types of parameters we also have to change the car class.

1. **Dependency injection as Design Pattern:**

Dependency injection is a coding pattern in which a class receives its dependencies from external source rather than creating them itself.

Class Car

{

Engine;

Tires;

Constructor(engine, tires)

{  
 this.engine = engine;

This.tires = tires;

}

}

Here, Car class doesn’t create any dependencies. It just uses them. The creation of those dependencies is external to this class.

Now to create a car, we are going to have code, something as follows,

Var myEngine = new Engine();

Var myTires = new tires();

Var myCar = new Car(myEngine, myTires);

Var myEngine = new Engine(new\_parameter);

Var myTires = new tires();

Var myCar = new Car(myEngine, myTires);

Var myEngine = new Engine();

Var myTires = new tires(new\_parameters);

Var myCar = new Car(myEngine, myTires);

There is also a problem with this type of implementation.

In the above example, Car class has only two dependencies (Engine and Tires). Suppose Car has 10 or 20 dependencies or more the above type of code is not recommended.

In this case it becomes difficulty for the developer.

**Here, angular dependency injection as a framework comes into picture.**

1. **Dependency Injection as Framework:**

Injector must register all dependencies. That means injector is like a container for all dependencies.

Engine ------------------ Service A

Tires -------------------- Service B

DepA ------------------- Service C

DepZ ------------------- Service Z

**Using a Service:**

**Dependency Injection as a framework:**

1. Define the studentService Class
2. Register with injector
3. Declare as dependency in Student-list and Student-details component.

We can create a service using: **ng g s <service name>**

Now this generated service class should provide student details to required components.

All the code that is reused by components should be written in this file.

After creating the service file we have to register the service with angular injector otherwise it is treated as a regular class according to angular.

* We can register the service in any component.
* Only the children of that component can access these services.
* If we register In app component file then all its children can access the services.
* But, in future if we create any other modules then they won’t be able to use those services.
* **So, it is best practice to register services in the app module.**
* **We always declare the services in the providers decorator (array).**
* **Then we have to import it on top of the app.module file.**

Now to use the service in a component we have to specify it in the constructor parameters of that component.

Constructor( Private \_testService : TestService )

{

}

Underscore is just a convention for private variables.

Now we have to assign the service to the variable inside that component. This code is written inside the constructor.

ngOnInit()

{

This.students = this.\_testService.getStudents.

}

**HTTP and Observables:**

In the above code example we wrote hard coded data inside the service. In general we should not do that. We should request data from the HTTP server.So, we use the HTTP requests.

**Observable:**

1. An observable is a sequence of items that arrive asynchronously overtime.
2. HTTP call (request) < - > Single item <- >HTTP response.
3. In angular, an observable is HTTP response which arrives asynchronously.
4. But, that incoming HTTP response(observable) is not ready to use in your application.
5. So, after receiving the observable we need to convert the incoming response into an array.
6. After conversion into an ARRAY only it is ready to use in our application.

Steps to follow in our example:

1. HTTP get request from testService
2. Receive the observable and cast it to the students array
3. Subscribe the observable from test and test1
4. Assign the student array to local variable.

Note:

**RxJS (Reactive Extensions for JS) is an external library used in angular to work with observables.**

**Fetching Data using HTTP :**

1. Import HTTP client moduel in app.module.ts file.

Import { HttpClientModule } from ‘@angular/common/http’

1. Then we have to declare this module In the imports array.
2. To use the service we should declare the HTTP client as dependency in the constructor.

Now we can send the http request.

1. Now inside the service getStudents method

This.\_http.get( <http url >)

url = < some url >

Constructor( private \_client : HttpClient) { };

getStudents( )

{

This.\_client.get(this.url);

}

1. Now we get an observable from the server. We cannot user observable directly in our application. So we have to cast that observable.
2. So, we create one type script file. And create one interface in it.

In that ts file:

Export interface StudentDetails

{  
 id : number;

Name : string;

Course : string;

}

So the code in getStudents becomes:

Import { StudentDetails } from ‘<file name>’

getStudents( ) : Observable <StudentDetails[ ]>

{

Return This.\_client.get<StudentDetails[ ]>(this.url);

}

Now the code inside the subscribed components will be changed to :

ngOnInit()

{

This.\_testService.getStudents()

.subscribe(data => this.students = data);

}

**HTTP Error Handling:**

For error handling we have to import **catchErrror** from **rxjs/operators.**

Then we have to .pipe that error using catchError.

Then the line in get students become

getStudents() : Observable<studentInterface[ ]>

{  
 return this.\_http.get<studentInterface[ ]>(this.\_url)

.pipe(catchError(this.errorHandler);

}

// In the above line of code errorHandler is a user defined function which is used to throw error.

errorHandler(error : HttpErrorResponse)

{

Return throwError(error.message || “unknown server error”);

}

In that throwError the message initially contains **null** value so we have to change it to a custom message. So we used “ || “ symbol.

Then we have to subscribe to that error it whatever component we have subscribed to that http request. Then the code becomes

errMsg !: string;

errMsg !: string;

ngOnInit(): void {

    this.\_stdService.getStudents()

    .subscribe(data => this.studentsList = data,

                error => this.errMsg = error);

  }

No we have to declare the errMsg variable in those subscribed components using interpolation then the message will be displayed whenever the server is facing an issue.

**Routing and Navigation:**

* Any angular application is built with multiple components.
* Each component has its own view
* Based on the user action, it may necessary to navigate among the views.
* For this purpose we use Angular Routers.
* We define all the routes in routes array in the app-routing.module.ts file.
* Each route is taken as an object.
* Every object has two parameters. They are:
  + Path :

It contains the path which will be displayed on the url of the page for to routing. It should be wrote within ‘ singlequotes ‘.

* + component :

It contains the component name, to which we have to route upon the path selection.

When we give the component names they will be imported to the app routing file.

Note:

whenever you have more routes you can declare them inside a constant array, and import that variable in the app module file and then mention it in the declarations. This will reduce the no of import lines to one.

Now to show the routes we have to declare <router-outlet> tag in side the app-component.html file.

We can use anchor tags for routings. To make this possible we use two special directives from the router package. They are:

1. routerLink directive :

*This specifies the path that we want to navigate to. We specify the the path using this directive.*

1. routerLinkActive directive:

*This directive let us specify one or more CSS classes that will be applied when the corresponding router link is active.*

Ex: <a routerLink = “/departments” routerLinkActive = “< CSS class >“>

<a routerLink = “/students” routerLinkActive = “ <CSS class> “>

**Wild Card Route:**

When the user tries to navigate to a route that has not been configured then the browser will throw an error in the console. To handle invalid urls we use **wild card route.**

By using a wild card route we can navigate the user to a 404 not found component if the url doesn’t match any of the configured routes.

In this example we are going to create a component with name page-not-found. And specify that component in the routing module routes array.

In Routes array in the route module specify the route as below:

{ path : “ \*\* “, component : PageNotFoundComponent }

The path value should be in “ double quotes “

The wild card route should be written at the bottom of the routes array.

The route will check every path from top to bottom to get the matching path. If the path is not matching with the specified path in the routes then the wild card route will get activated.

If we write wild card route at the start of the array then only it will be matched with the link in the address bar. And always the page not found component will work.

**But,** if you follow the above process you get page not found even in the initial state of the website which is not desired. So to avoid this we have to specify another path in the routes array.

This path is only contain a **space.** SO this path states that in the initial start of the page which component needs to be displayed.

**And** this path needs to be written at the start of the route array.

The code in route array will look like:

{path : ‘ ’ , component : ‘<component name that should be displayed>’ }

**But the above code is not correct process:**

Instead of component we need to use redirectTo for redirection.

The code will look like :

{path : ‘’, redirectTo : ‘/student’, pathMatch : ‘<match name>’ }

Here, pathMatch specifies if the path matches to certain then use this path.

The following can be used in pathMatch:

1. **prefix:** it matches the first character of the path in addressbar
2. **full :** it matches the full path in the addressbar

**Route parameters ( how to pass and read ):**

**To pass parameter to routing we use Router Module:**

First we are writing code for the address in searchbar to change based on the component that we select.

So, create another path to which we have to redirect

{path : ‘student/:id’, component: <the component to which we redirect>}

Now we have to create a click event in the html file of the component on which we clicked. Then in it’s ts file Import Router module and then

Declare a private variable inside its constructor of type Router.

Then use this variable inside the click even method using:

onSelect(student : any)

{

**this.\_router.navigate([‘/test1,student.id ]);**

}

Here student.id is just the parameter of the function. From which we are taking id data.

the first parameter in the array is the path that we want to navigate to.

The second parameter is the route parameter to which the sub path is shown.

Ex: localhost:4200/test1/1

Here, test1 is the component that we redirected and 1 is the student.id value that we clicked.

**This is how we pass parameters while routing.**

**To show the details in the view:**

**To read parameter we use ActivatedRoute Service:**

For this we use ActivatedRoute service in that component’s ts file.

And then in the ngOnInit:

Let id = parseInt(this.\_route.snapshot.paramMap.get(‘id’));

( parseInt is not working. It’s saying null cannot be converted to int )

And then assign the value inside id to a class variable. And then show it in the html page using interpolation.

**paramMap observables:**

In the above codes we used snapshot of the route parameter. But this approach has a drawback.

In our application we navigate to course details component from the course list component.

The snapshot method only get the value at that particular instance until the component is changed it’s value does not change.

So we have declared the code in ngOnInit method so at the time of initialisation the code will work but when we create two buttons to navigate to different Id’s then the value at the address bar won’t change.

So If we use the param observable and subscribe to that event then we can access the value whenever there is a change to it. Even if the component is not recreated.

The paramMap code is written using the ActivatedRoute service. The code in ngOnInit will be as follows:

This.activatedRoute.paramMap.subscribe((params : ParamMap) => {

Let id = params.get(‘id’);

This.courseId = id;

})

  constructor(private \_activatedRoute : ActivatedRoute, private \_router : Router) { }

  ngOnInit(): void {

    // this.courseId = (this.\_activatedRoute.snapshot.paramMap.get('id'));

    this.\_activatedRoute.paramMap.subscribe((params : ParamMap) =>

    {

      let id = params.get('id');

      this.courseId = id;

    })

  }

  goPrevious()

  {

    let previousId = parseInt(this.courseId) - 1;

    this.\_router.navigate(['/course', previousId]);

  }

  goNext()

  {

    let nextId = parseInt(this.courseId) + 1;

    this.\_router.navigate(['/course', nextId]);

  }

By following the above procedure we can navigate from the course list component to the course details component, and whatever the id that we have selected in the course list component will be displayed in the course details component.

**Optional Route Parameters:**

Optional route parameters can be used to perform certain operations on the routes based on which route is selected.

In the below code we extracted the id of the element which is selected and used to apply specific CSS for that element.

constructor(private \_router : Router, private \_route : ActivatedRoute) { }

  ngOnInit(): void {

    this.\_route.paramMap.subscribe((params: ParamMap) =>

    {

      let id = params.get('id');

      this.selectedId = id;

      this.selectedId = parseInt(this.selectedId);

      console.log(this.selectedId)

    })

  }

  onSelect(data : any)

  {

    this.\_router.navigate(['/course', data.id]);

  }

  isSelected(data : any)

  {

    return data.id === this.selectedId;

  }

In html: we included property binding on to add a CSS class called selected to the element whenever we click on it. Based on the optional route parameters. This optional parameter doesn’t need a container like the route parameters (we used id as a container for the route parameter)

In the above code we have subscribed to that paramMap value. So that every time an event happens to that element then this paramMap will give us an id of that element.

<div>

    <ul>

        <li (click) = "onSelect(sample)" [class.selected] = "isSelected(sample)"\*ngFor = "let sample of sampleDetails">

            <span class="student\_name"><span class="student\_id">{{sample.id}}</span> {{sample.name}} </span>

        </li>

    </ul>

</div>

**Relative Navigation:**

Absolute path begins with a forward slash “ / “. When we use absolute paths there is lack of flexibility in our paths.

If we want to change the previous section’s routing code from “ course “ to something like “ course-list “ then we have to change all the occurrences of the absolute path and this is quite difficult when we are working on a large project.

The following is the code In :

Course-list component :

  onSelect(data : any)

  {

    // this.\_router.navigate(['/course', data.id])      // absolute path

    this.\_router.navigate([data.id], {relativeTo : this.\_route}); // relative path

  }

Course-details component :

  gotoCourseList()

  {

    let selectedId = this.courseId ? this.courseId : null;

    // this.\_router.navigate(['/course', {id : selectedId}]);   // absolute path

    this.\_router.navigate(['../', {id : selectedId}], {relativeTo: this.\_activatedRoute});

  }

**Brief:**

In relative navigation we use routing in the initial component and then navigate to some other url. And to get back from that url we use the activated route and use the router package to again navigate to the url.

**Child routes:**

In our angular application some routes may be viewed only in other routes. In such a scenario we create child routes.

We specify child routes in angular by using children property and set it to an array of routes.

Then we go to the component where we want to create our child routes. Create two buttons and assign click events.

In the click events we specify the navigate using the router package and navigate to it using the url and activated route service.

  { path : 'course/:id',

    component : CourseDetailsComponent,

    children : [

      {path : 'overview', component : StudentOverviewComponent},

      {path : 'contact', component : StudentContactComponent}

    ]

  }

<p>

    <button (click) = "showOverview()">Overview</button>

    <button (click) = "showContact()">Contact</button>

</p>

  showOverview()

  {

    this.\_router.navigate(['overview'], {relativeTo : this.\_activatedRoute});

  }

  showContact()

  {

    this.\_router.navigate(['contact'], {relativeTo : this.\_activatedRoute});

  }

**Lazy Loading:**

In angular by default, modules are eagerly loaded. Means, as soon as the app is loaded all the modules are loaded without checking whether they are immediate required or not?

Lazy loading is a design pattern that loads required modules only.

Lazy loading helps to keep initial bundle size smaller, which helps to decrease the load time and increase the performance.

Two steps to create a lazy loading feature:

1. Create a feature module
2. Configure load children in appRouting.

**Syntax:**

Ng g m faculty - - route faculty - - module app.module.ts (maybe its app-routing.module.ts)

* Generate a module with the name “ faculty “
* Create a lazy lading feature in Routing file by the path as “ faculty “
* Update in app.module.ts (app-routing.module.ts)

Then a module with name faculty will be created and it will be imported to the app-routing.module.ts file with lazy loading. Then the code will be as follows in the Routes array:

  { path: 'faculty', loadChildren: () => import('./faculty/faculty.module').then(m => m.FacultyModule) },

**Route Guards :**

* Route guards are used to prevent users from navigating without having authorization rights.
* Route Guards are used to secure the route paths.
* We can generate any number of guards based on our application requirement.
* Whenever we implement a route guard, it will give Boolean value (True or False).
* Based on this Boolean value, Angular router decide if user should access the route or not?

**Syntax:**

ng g guard < guard name >

Inject this guard in our module under providers array.

**Types of Guards:**

There are various types of route guards available:

* **CanActivate –** Checks to see if a user can visit a Route.
* **CanActivateChild –** Checks to see if a user can visit a route children
* **CanDeactivate –** Checks to see if a user can exit a route
* **Resolve –** Performs route data retrieval before route activation
* **CanLoad –** Checks to see if a user can route to a module that lazy loading

**Note :** Route guard resolve to TRUE or FALSE based on Custom Logic and Functionality.

While creating the route we get an option to select the type of route that we want to use. Then we can select that particular type of guard.

Then after going into the route file we can see that type of guard in the class file.

Here the RouterStateSnapshot specifies the return value for the guard when it gets activated. We can change the return value type of the guard there.

Then go into the app-routing.module.ts file and there specify a path in he routes array:

Ex: { path : ‘admin’ ,

component : AdminHomeComponent,

canActivate : [ AdminAccessGuard ]

}

And declare it in the app.module.ts file under the providers section.

In admin-access file:

export class AdminAccessGuard implements CanActivate {

  canActivate(

    route: ActivatedRouteSnapshot,

    // state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {

    state: RouterStateSnapshot) : boolean{  // We only need a boolean value so I changed the return value type to boolean

    return false  ;

  }

}

In app-routing.module.ts file in Routes array:

  { path : 'admin',

    component: AdminHomeComponent,

    canActivate: [AdminAccessGuard]},

In app.module.ts :

  providers: [TestService, AdminAccessGuard],

Based on the return value of the canActivate function the AdminHomeComponent page will be available to access. Else it will be redirected to the localhost:4200 even if we want to access localhost:4200/admin without the access.

**ANGULAR FORMS INTRODUCTION:**

Forms are very important for any business application.

Forms are used to create an experience that guides an user efficiently and effectively through the workflow.

As a developer we have to do,

* Data Binding
* Change Tracking
* Validation
* Visual feedback
* Displaying error message
* Form submission

Diagram

Description automatically generated

Two approaches for forms:

1. **Template Driven Forms :**

Most of the code is written in the HTML file.

1. **Reactive forms / Model driven forms :**

Most of the code is written in the Component class.

**Template Driven Forms ( TDF ):**

* Template driven forms are forms where we write logic, validations, controls etc, in the template part of the code (HTML file).
* The template is responsible for setting up the form, the validation, control, group etc.
* Template driven forms are suitable for simple scenarios, uses two way data binding using the [( NgModel )] syntax.
* But unit testing might be a challenge.
* TDF forms are easy to use and similar to Angular JS forms.
* We don’t have to keep track of the input field values and react to change in the input field values. Angular takes care that with the ngModel directive.
* As a result of that we have Bulky HTML code and minimal component code.
* In TDF Angular automatically tracks the form and form elements state and validity.
* **The form validation logic cannot be unit tested. The only way to test the logic is to run an end to end (e2e) test with a browser.**
* Readability of a form decreases with increase in form validations.
* **Suitable for simple scenarios.**
* **When we have to create a simple form for which unit testing can be handled with the browser go with the TDF approach.**
* **For more complex forms with more complex validations and where unit testing is absolutely necessary go with reactive forms.**

To use angular forms import the FormsModule from @angular/core.

Include that module name in the imports section.

We can get a hold of this FormsModule using a **template reference variable**. And set it to “ ngForm “. ngForm exports itself as the string ngForm and by assigning it to a reference variable we have a reference to the directive itself.

The directive gives us access to the values of the Form controls by using the **value** property.

<div class = "container-fluid">

    <h1>Bootcamp Enrollment Form</h1>

    <form #userForm = "ngForm">

        {{userForm.value | json}}

        <div class = "form-group">

            <label>Name</label>

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

        </div>

Form continues

But the value of that json interpolation will be nothing. Because by default angular does not track every single form control in the page. We can specify which of the form control need to be tracked by using the **ngModel** directive.

        <div class = "form-group">

            <label>Name</label>

            <input type = "text" class = "form-control" ngModel>

        </div>

If you run the above code you will get an error

ERROR Error: If ngModel is used within a form tag, either the name attribute must be set or the form

control must be defined as 'standalone' in ngModelOptions.

Example 1: <input [(ngModel)]="person.firstName" name="first">

Example 2: <input [(ngModel)]="person.firstName" [ngModelOptions]="{standalone: true}">

Because if we use ngModel directive inside a form we have to either set the **name** attribute or **formcontrol** must be defined as **standalone.**

After specifying the name attribute for all the input fields then the json output on the screen will be as follow:

{ "userName": "Prashanth Bocha", "email": "bocha.prasanth", "phone": "", "topic": "", "timePreference": "", "subscribe": "" }

The value changes by the corresponding input change.

* The input values correspond to the text in the field.
* Radio button values correspond to the value attribute.
* Select tag corresponds to value attribute ( if present ) else it corresponds to the text that’s present.
* Checkbox corresponds to either **true** or **false.**

In addition to the ngModel angular also provides **ngModelGroup** directive.

**ngModelGroup directive:**

We use the ngModelGroup directive if we like to group together or create a sub group within a form.

Ex: Consider address. Address can have street, city, country and postal code. We can group all of those fields into an address object by using the ngModelGroup directive.

    <form #userForm = "ngForm">

        {{userForm.value | json}}

        <div ngModelGroup = "address">

            <div class="form-group">

                <label>Street</label>

                <input type="text" class = "form-control" name = "street" ngModel>

            </div>

            <div class="form-group">

                <label>City</label>

                <input type="text" class = "form-control" name = "city" ngModel>

            </div>

            <div class="form-group">

                <label>State</label>

                <input type="text" class = "form-control" name = "state" ngModel>

            </div>

            <div class="form-group">

                <label>Postal Code</label>

                <input type="text" class = "form-control" name = "countryCode" ngModel>

            </div>

        </div>

The json output in the html page will be as follow:

{ "address": { "street": "", "city": "", "state": "", "countryCode": "" }, "userName": "", "email": "", "phone": "", "topic": "", "timePreference": "", "subscribe": "" }

As you can see in the above output. All the address fields are grouped into a single address object. Which is done by using the ngModelGroup directive.

When it comes to data binding in template driven forms, we have three directive. They are:

1. ngForm
2. ngModel
3. ngModelGroup

By using these directive we can bind the data inside the form. And by using templateReferenceVariable.value.

Ex: userForm.value

And send it to the server. Also this works completely fine, but a better approach is to bind the data to a user defined model, and send that model data to the server.

**Binding Data to a Model:**

In this we bind the form data to a model. As users enter the data we capture the changes and update the instance of the model that can later be sent to the server.

The first step is to generate a model class.

**ng generate class <name of the class>**

If we run the above command then we get a ts file and a spec.ts file with the class name.

Then in the constructor of the class in ts file. Write the different properties of the class.

export class User {

    constructor (

        public name : string,

        public email : string,

        public phone : number,

        public topic : string,

        public timePreference : string,

        public subscribe : boolean

    ) {};

}

Then open app.component.ts file and create a new property inside the class with the instance as the “ User “ class.

Import { User } from “./user”

  userModel = new User('Prashanth', 'psh@gmail.com', 1234567890, '', 'morning', true);

By having this instance of the model, it is now possible to bind the userModel data to our enrolment form.

A Simple use case for would be updating or editing data that is already saved.

Ex:

If you are on an e-commerce site and you have a shipping address if you click on the edit button a form appears prefilled with the shipping address data.

So the shipping address is bound to the form.

As you can see even though angular creates its own object of the form values there are definitely use cases where it is necessary to create our own model and bind that model data to our form.

We are going to use interpolation to bind the userModel and display it in the view. This will help us see how the data binding works.

        {{userModel | json}}

Binding the userModel to the form is really simple, we bind the properties of the model to the ngModel directive. And for property binding we make use of the square brackets.

userModel has a property of name. And we bind that property to the ngModel directive inside our HTML page. So we use square brackets.

            <input class = "form-check-input" type = "checkbox" name = "subscribe" [ngModel] = "userModel.subscribe">

* If we do this to all the ngModel inside our HTML file. And see the output the json values will be same as the angular form values.
* But when we change the value inside our input in the view, the angular ngModel form values change but the ngModel which has the property binding of the userModel does not change.
* This is because the property binding is a one-way binding. That means data from the class is sent to the view but the data from the view is not sent to the class.
* So, we need to use the two-way data binding using **“ [ ( ) ] “** syntax.

Now the userModel is binded with data and is ready to be sent to the server. But before doing so we need do some client-side validations to the values. And give some visual feedback to the user.

**Tracking state and Validity:**

Graphical user interface, application

Description automatically generated

At any point in time angular applies three classes to the form control based on its state. They are shown in the above picture.

* First classes are dependent whether the user has visited the form control or not.
  + When you load a form for the very first time, you have not yet visited a form field. So angular applies a class called **ng-untouched.**
  + If you do visit the form control either by clicking, tabbing or navigating away from the form control angular applies a class called **ng-touched.**
  + **The key-point here is that the class will change only onblur. You have to navigate away from the form control.**
* When you load a form for the very first time the value has not yet changed. Then angular applies the **ng-pristine** class to the form.
* If you do change the value of the form control, angular applies a class called **ng-dirty.**

Ex: If you type a value in the input field of the form, the value changes from ng-pristine to ng-dirty.

* The last pair of classes are considered for the validity of the form control. If the form-control’s value is valid then **ng-valid** class is applied. If not **ng-invalid** class is applied.

With the help of these classes we can provide visual feedback to the user.

For each of the classes Angular also provides associated property on the ngModel directive. The property is same as the class without the “ ng- “ in it as shown in the below table:

|  |  |
| --- | --- |
| Class | Property |
| ng-untouched | untouched |
| ng-touched | touched |
| ng-pristine | pristine |
| ng-dirty | dirty |
| ng-valid | valid |
| ng-invalid | Invalid |

To use the above ngModel properties, first we have to create a reference to the ngModel directive:

            <input type = "text" class = "form-control" #name name = "userName" [(ngModel)] = "userModel.name" required>

In the above code the #name is a template reference variable for the input element in the DOM.

By assigning it a value of ngModel, now the template reference variable points to the ngModel of that particular form control.

now we can access the properties of the ngModel of the element.

By setting #name = “ngModel”

Then {{ name.pristine }} ----- Output is False or true; based on the state of the element. The output of these properties is Boolean.

**Input field required :**

            <input type = "text" class = "form-control" [class.is-invalid] = "name.invalid" #name = "ngModel" name = "userName" [(ngModel)] = "userModel.name" required>

Here we are applying a bootstrap class is-invalid to the element if its ngModel state is invalid.

Then if we don’t fill the filed the input field will be displayed in red color.   
**But** the problem with the above approach is that, even in the initial condition when there is no value inside the input field, it will be in red color.

**So,** the above code can be changed to :

            <input type = "text" class = "form-control" [class.is-invalid] = "name.invalid && name.touched" #name = "ngModel" name = "userName" [(ngModel)] = "userModel.name" required>

Here, we’re taking use of touched property of the ngModel. If both are true then the is-invalid class will be applied to the element.

**Pattern Matching :**

            <input type = "tel" #phone = "ngModel" pattern = "^\d{10}$" class = "form-control" [class.is-invalid] = "phone.invalid && phone.touched" name = "phone" [(ngModel)] = "userModel.phone" required>

The above code will apply the is-invalid class to the element if the number of digits inside the input element is not equal to 10.

**Displaying Error Messages when field’s are invalid:**

        <div class = "form-group">

            <label>phone</label>

            <input type = "tel" #phone = "ngModel" pattern = "^\d{10}$" class = "form-control" [class.is-invalid] = "phone.invalid && phone.touched" name = "phone" [(ngModel)] = "userModel.phone" required>

            <small class="text-danger" [class.d-none] = "phone.valid || phone.untouched" >Phone number is required and must be 10 digits</small>

        </div>

The above code will display an error message when there is no phone number or the number of digits are not equal to 10.

We can display custom error message based on the specific type of error by using the **errors** property

        <div class = "form-group">

            <label>phone</label>

            <input type = "tel" #phone = "ngModel" pattern = "^\d{10}$" class = "form-control" [class.is-invalid] = "phone.invalid && phone.touched" name = "phone" [(ngModel)] = "userModel.phone" required>

            <!-- <small class="text-danger" [class.d-none] = "phone.valid || phone.untouched" >Phone number is required and must be 10 digits</small> -->

            <div \*ngIf = "phone.errors && (phone.invalid || phone.touched)">

                <small class="text-danger" \*ngIf = "phone.errors.required">Phone number is required</small>

                <small class="text-danger" \*ngIf = "phone.errors.pattern">Phone number should be 10 digits</small>

            </div>

        </div>

With the help of above code we only display the error based on the validation that is being applied.   
Ex:

If there is no number then the required validation will be applied and the error will be displayed as ------ “ Phone number is required “.

If the number of digits is not equals to 10 then the pattern validation is applied and the error will be displayed as ----- “ Phone number should be 10 digits “.

**Select Control Validation:**

In this example we set a custom validation for the select tag in our HTML page.

We can use the required attribute like we did in the above case, but doing that won’t be helpful because the value of the initial “ I am interested in “ option will be selected by default and it does not show any error.

So we rewrite the form validation so that if the default message is displayed and the user is trying to submit the form, then there will be error raised and displayed. The code will be as follows :

In HTML :

        <div class = "form-group">

            <select (blur) = "validateTopic(topic.value)" (change) = "validateTopic(topic.value)" class = "form-select" #topic = "ngModel" name = "topic" [(ngModel)] = "userModel.topic" [class.is-invalid] = "topicHasError && topic.touched">

                <option value = "default" selected>I am interested in</option>

                <option \*ngFor = "let topic of topics">{{ topic }} </option>

            </select>

            <small class="text-danger" [class.d-none] = "!topicHasError || topic.untouched">Please choose a topic</small>

        </div>

In ts file :

topicHasError = true;

validateTopic(value : any){

if(value === "default"){

this.topicHasError = true;

}else{

this.topicHasError = false;

}

}

Now when the default value is selected then the error will be raised saying “ Please choose a topic “. The error will be gone when we select any other value from the given options.

**Form Validation:**

In this section we see how angular helps us with the form level validation:

The ngForm will also contain some properties like the ngModel.

ngForm has a valid property in it which gives us a Boolean value based on the state of the entire form.

If every form input element’s validation is **“true”**, then the entire form valid state will be true. Otherwise it will be **“false“**.

        <button [disabled]="userForm.form.invalid" class = "btn btn-primary" type = "submit">Submit form</button>

The above code will check only for the default validation, but not for any custom validations that we made for the form. Ex: we made a custom validation for the select button in the form.

We can check that custom validation in two ways:

1. When the submit button is clicked then an event is fired. And we use that event to check for the value of the select button. If the select button value is “default” then the form should not be submitted.
2. Or We can add another condition to the disabled property.

        <button [disabled]="userForm.form.invalid || topicHasError" class = "btn btn-primary" type = "submit">Submit form</button>

In the above code topicHasError is a property of the ts file. We have used this property to validate the select input’s value.

**Submitting Form Data:**

* The first step is to add the **novalidate** attribute on the form tag. This will prevent browser validation from happening when we submit the form.
* The next step is to bind it to the ngSubmit event which gets emitted when the submit button is clicked. So on the ngForm bind to ngSubmit and a handler. Ex: (ngSubmit) = “ onSubmit() “

Then we create a service called enrolment

ng g s <service name>

Ex: ng g s enrolment

Then we import HTTP client module, inject it to the constructor and create a service and send e request to the url

In enrolment.ts file :

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

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

import { User } from './user';

@Injectable({

  providedIn: 'root'

})

export class EnrollmentService {

  \_url = ''

  constructor(private \_http : HttpClient) { }

  enroll(user : User){

    return this.\_http.post<any>(this.\_url, user);

  }

}

Then subscribe to that service in the app.component.ts file:

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

import {User} from './user'

import { EnrollmentService } from './enrollment.service';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  topics : string[] = ["Angular", "React", "Vue"];

  topicHasError = true;

  userModel = new User('Prashanth', 'psh@gmail.com', 1234567890, '', 'morning', true);

  constructor(private \_enrollmentService : EnrollmentService){}

  validateTopic(value : any){

    if(value === "default"){

      this.topicHasError = true;

    }else{

      this.topicHasError = false;

    }

  }

  onSubmit(){

    this.\_enrollmentService.enroll(this.userModel)

    .subscribe(

      data => console.log('Success!', data),

      error => console.log('Error!', error)

    )

  }

}

**Express server to receive form data:**

In this section we are going to use the express service as server and send the data to the server.

To generate the express server:

* Create another folder
* npm init - -yes
* npm install - -save express body-parser cors
* Then create a file with name server.js

Here,

**Express :** is our webserver

**body-parser :** is the middle ware to handle the form data

**cors :** is a package to make requests across different ports

Code inside server.js :

const express = require('express');

const bodyParser = require('body-parser');

const cors = require('cors');

const PORT = 4300;

const app = express();

app.use(bodyParser.json());

app.use(cors());

app.get('/', function(req, res){

    res.send("hello from server");

});

app.post('/enroll', function(req, res){

    console.log(req.body);

    res.status(200).send({"message" : "Data received"});

})

app.listen(PORT, function(){

    console.log("Server running on localhost: " + PORT);

});

To run the server enter: **node server** in cmd

Then in enrollment.service.ts file :

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

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

import { User } from './user';

@Injectable({

  providedIn: 'root'

})

export class EnrollmentService {

  \_url = 'http://localhost:4300/enroll';

  constructor(private \_http : HttpClient) { }

  enroll(user : User){

    return this.\_http.post<any>(this.\_url, user);

  }

}

Inside app.component.ts :

  onSubmit(){

    this.\_enrollmentService.enroll(this.userModel)

    .subscribe(

      data => console.log('Success!', data),

      error => console.log('Error!', error)

    )

    this.submitted = true;

  }

So when we submit the form we get an response from the server with in our console. And the data gets submitted to the server.

**Error Handling:**

Error handling means showing the user what error had occurred in the system. When the server doesn’t work properly we may get some error and the user should know that there is an error in the server, in such cases with the help of error handling we can make our webpage more user friendly.

To use error handling we are piping the response of the enroll method inside the enrollment.service.ts file:

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

import { HttpClient, HttpErrorResponse } from '@angular/common/http'

import { User } from './user';

import { catchError } from 'rxjs/operators';

import { throwError } from 'rxjs';

@Injectable({

  providedIn: 'root'

})

export class EnrollmentService {

  \_url = 'http://localhost:4300/enroll';  // request to the express local server

  constructor(private \_http : HttpClient) { }

  enroll(user : User){

    return this.\_http.post<any>(this.\_url, user)

      .pipe(catchError(this.errorHandler))

  }

  errorHandler(error: HttpErrorResponse){

    return throwError(error);

  }

}

Then change the error handling in the app.component.ts file to:

  onSubmit(){

    this.\_enrollmentService.enroll(this.userModel)

    .subscribe(

      data => console.log('Success!', data),

      error => this.errorMsg = error.statusText

    )

    this.submitted = true;

  }

And then using interpolation we display the error as :

    <div class="alert alert-danger" \*ngIf = "errorMsg">

        {{errorMsg}}

    </div>

**REACTIVE FORMS:**

* Code and logic resides in the component class.
* No two-way binding
* Well suited for complex scenarios
* Dynamic form fields
* Custom validation
* Dynamic validation
* Unit test
* Reactive forms require more coding

As far as HTML code goes there is not difference between template driven forms ( TDF ) and reactive forms.

**Creating the form model:**

In angular to work with reactive forms we have to import ReactiveFormsModule from @angular/forms in app.module.ts file and add it to the imports array.

The ReactiveFormsModule gives us access to a bunch of classes and directives that are necessary to build reactive forms. Out of those two classes made up the building blocks of reactive forms. They are:

1. formGroup
2. formControl

In reactive forms the form is represented by model in the component class, and to be able to create that model we make use of formGroup and formControl classes.

The entire form is represented as formGroup and each input field in the form is represented as formControl.

Now, create a new instance of FormGroup in the app.component.ts file and inside the FormGroup constructor pass an object with FormControl values as instances.

export class AppComponent {

  registrationForm = new FormGroup({

    userName : new FormControl('Prashanth'), // If you want any input field to have a default value we can pass it to the formControl constructor.

    password : new FormControl(''),

    confirmPassword : new FormControl('')

  });

}

Now to bind the form to the view. On the form tag in HTML we use the formGroup directive and bind the registration form group.

Ex:

<form [ formGroup ] = “registrationForm”>

………………..

</form>

And for form control we bind the inputs of the form with the

formControlName = “<name of the field>”

Ex: formControlName = “userName"

By doing this we created a 1 : 1 association between the formGroup, formControls and their corresponding HTML elements. The entire form is associated with formGroup “ registrationForm “.

And each individual formControl from the model is associated with its corresponding HTML element userName, password, and confirmPassword in page.

formGroup.value will provide the corresponding value of the form, which will be helpful for sending the data to the server.

{{ registrationForm.value | json }}

**Nesting Form Group:**

export class AppComponent {

  registrationForm = new FormGroup({

    userName : new FormControl('Prashanth'), // If you want any input field to have a default value we can pass it to the formControl constructor.

    password : new FormControl(''),

    confirmPassword : new FormControl(''),

    address : new FormGroup({

      city : new FormControl(''),

      state : new FormControl(''),

      postalCode : new FormControl('')

    })

  });

}

Here address is another FormGroup nested inside registrationForm FormGroup.

    <div  formGroupName = "address">

      <div class="form-group">

        <label>City</label>

        <input type="text" class="form-control" formControlName = "city">

      </div>

      <div class="form-group">

        <label>State</label>

        <input type="text" class="form-control" formControlName = "state">

      </div>

      <div class="form-group">

        <label>Postal Code</label>

        <input type="text" class="form-control" formControlName = "postalCode">

      </div>

    </div>

**Managing control Values:**

Managing control values lets you to set formcontrol values without any user interaction, i.e., setting values programmatically.

Ex: We might have to retriever form data from a backend API or service and update the formControls to their new values.

We can do that by using the set value method provided by reactive forms.

The setValue method can be called on formControl class or the formGroup class. For our purpose we need to call the method on registrationForm group.

The setValue method accepts an object that matches the structure of the form group with the control names as keys.

We created a button in the html page and set it a click event to load the data from the api function

  loadApiData() {

    this.registrationForm.setValue({

      userName : 'Prashanth',

      password : 'test',

      confirmPassword : 'test',

      address : {

        city : 'City',

        state : 'State',

        postalCode : '123456'

      }

    });

Now, if we press the button the data from the function will be set into the fields inside our form as follows:

Graphical user interface, text, application, email

Description automatically generated

The setValue method is strict and If you don’t match the form structure in your setValue method’s passed object then it will throw an error.

**But,** if you want to fill only the specific fields of the form, then we can use pathValue method. And omit the fields that should not be filled from the form structure inside the object.

    this.registrationForm.patchValue({

      userName : 'Prashanth',

      password : 'test',

      confirmPassword : 'test',

    });

So, the setValue and patchValue methods are useful when you want to fill the form fields with data from an API or a service.

Set value is used when you set the values form all the formcontrols, and patch value is used when you want to set the values to few of the form controls.

**Form Builder Service:**

Creating multiple formControl instances manually can be repetitive. To avoid this angular provides the FormBuilder service which in turn provides methods to handle generating formControls.

To use FormBuilder:

* Import FromBuilder service from @angular/forms
* Inject it to the constructor

FormBuilder is a simpler alternative for building form groups and and formcontrols.

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

in class :

  constructor(private \_fb : FormBuilder){}

  registrationForm = this.\_fb.group({

    userName : ['Prashanth'],

    password : [''],

    confirmPassword : [''],

    address : this.\_fb.group({

      city : [''],

      state : [''],

      postalCode : ['']

    })

  });

The form builder service’s group function takes an object that defines the structure of the formgroup.

Within the structure we specify the form controls and the values of the form controls is specified in the form of an array.   
The first value at the zero index of that array indicates the default value of that form control.

**Simple validations in reactive forms:**

Reactive forms include a set of validator functions out of the box for common use case. Unlike template driven forms ( TDF ) in reactive forms validation rules are specified in the component class instead of the template.

1. **Apply validation rule to a form control:**

To make use of built in validations we have to use the validator class.

First you have to Validators class from @angular/forms.

**The** second element inside the form control array inside the formBuilder group function is where we specify the validations for the form control.

1. **Multiple Validations**

To specify a simple required validation we can write Validators.required in the array.

To specify more than one validations to the form using Validators class we can declare the validations as the form of an array.

    userName : ['Prasanth', [Validators.required, Validators.minLength(3), Validators.maxLength(5)]],

in html file we used the properties of the registrationForm to get the required errors.

    <div class="form-group">

      <label>Username</label>

      <input [class.is-invalid] = "registrationForm.get('userName')?.invalid && registrationForm.get('userName')?.touched" type="text" class = "form-control" formControlName = "userName">

      <div \*ngIf = "registrationForm.get('userName')?.invalid && registrationForm.get('userName')?.touched">

        <small class = "text-danger" \*ngIf = "registrationForm.get('userName')?.errors?.required">Username Required</small>

        <small class = "text-danger" \*ngIf = "registrationForm.get('userName')?.errors?.minlength">Username Should be more than 3 characters</small>

        <small class = "text-danger" \*ngIf = "registrationForm.get('userName')?.errors?.maxlength">Username Should be less than 5 characters</small>

      </div>

    </div>

Writing the above code makes your page more user friendly.

**But,** we can write the above come much simpler by using a getter that returns the form control.

Now the above code becomes :

In ts file:

  get getUserName() {   // getter method that returns the userName of the registration form which is displayed to show errors in the form to the User.

    return this.registrationForm.get('userName');

  }

In HTML file:

    <div class="form-group">

      <label>Username</label>

      <input [class.is-invalid] = "getUserName?.invalid && registrationForm.get('userName')?.touched" type="text" class = "form-control" formControlName = "userName">

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

        <small class = "text-danger" \*ngIf = "getUserName?.errors?.required">Username Required</small>

        <small class = "text-danger" \*ngIf = "getUserName?.errors?.minlength">Username must be at least 3 characters</small>

        <small class = "text-danger" \*ngIf = "getUserName?.errors?.maxlength">Username must be at most 5 characters</small>

      </div>

    </div>

**Custom Validations:**

The built in validators won’t always match the exact user case of your application. In such scenarios we can create our own custom validators. As follows:

Let’s create a custom validation for the userName field.

Ex:

If we enter the username as admin, we should display an error message saying “username as admin is not allowed”.

A lot of the time we might want to filter out the spam registrations, username or email fields usually contains a string to identify such spam registrations. Let’s assume the spam word is “admin” for our example.

A custom validator is nothing but a function. The function can be written right into the component file itself. **But,** validator functions are usually reused in several places in our application it is always a good idea to create a separate file and export them.

Let’s create:

* A file with name user-name.validator.ts and
* create a function with name forbiddenNameValidator.
* The function accepts one parameter which is the form control being validated
* The type of the control is AbstractControl, make sure to import it from @angular/forms.
* The validator function return either of two values.
* **When the validation fails:** it returns an object where the key is of type string and the value is of type any.
* **If the validation passed:** It return null
* **Now:** we create a flag with name forbidden and then use **test** method to check whether the control.value contains string admin
* **If** it contains the admin string we set the flag to **true** or else we set it to **false.**
* Now import this function in the app.component.ts file and then mention it in the validators array.

In user-name.validators.ts file:

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

export function forbiddenNameValidator(control : AbstractControl): {[key : string] : any} | null {

    const forbidden = /admin/.test(control.value);

    return forbidden ? {'forbiddenName' : {value : control.value}} : null;

}

In app.component.ts file:

    userName : ['Prasanth', [Validators.required, Validators.minLength(3), Validators.maxLength(5), forbiddenNameValidator]],

in HTML file:

        <small class = "text-danger" \*ngIf = "getUserName?.errors?.forbiddenName">Username {{getUserName?.errors?.forbiddenName.value}} is not allowed</small>

It is also possible to pass parameters to custom validators. Right now we are forbidding the string ‘ admin ‘ in the username. But there could be another field that forbids the string ‘ password ‘. So, we should be able to pass in the string we want to forbid as a parameter to our custom validator. We can do this by:

**The drawback** of a validator function is that it accept only one parameter which is the form control. So, we cannot simply pass in a second parameter.   
Instead what we have to do is to create a factory function, that accepts a string as a parameter and returns the validator function itself.

/\*

    The below function is a factory function or a function which returns a validator function

\*/

export function forbiddenNameValidator(forbiddenName : RegExp) : ValidatorFn {

    return (control : AbstractControl): {[key : string] : any} | null => {

        const forbidden = forbiddenName.test(control.value);

        return forbidden ? {'forbiddenName' : {value : control.value}} : null;

    };

}

**Crossfield Validation:**

] ] ] ] ]

]}}} }