**NODE.JS**

Since Javascript runs in browser , to help it to run outside the browser NODE.JS was created.

**ANGULAR CLI**

A utility used to run Angular commands to develop applications. It helps in generating the pre-working app with loaded essential components.

**ANGULAR CLI COMMANDS**

Command Prompt-

1. **node –v** = To check node version, You can download from nodejs site and also update.
2. **npm –v** = To check npm version

**npm update -g** = To update npm globally use

1. **ng –v** = To check angular version

**npm install -g @angular/cli** = To install angular globally

**ng update @angular/cli @angular/core** = update angular

1. **ng new <my-first-project>** = create new project
2. **cd <my-first-project>** = change directory to that project
3. **ng serve** = run that project
4. **ng g class <name>** = to generate class, pipe, service, component
5. **To create a server using Express**

**Create a folder and in its command terminal type**

**npm init -- yes**

**npm install --save express body-parser cors**

**node server** == To run the server

1. **npm install bootstrap –** to install bootstrap across application instead of copying links.

Add this in angular.json in architect/build/styles "./node\_modules/bootstrap/dist/css/bootstrap.min.css"

Add this in angular.json in architect/build/scripts "./node\_modules/bootstrap/dist/js/bootstrap.js"

1. **npm install jquery –** to install jquery

Add this in angular.json in architect/build/scripts before bootstrap.js

“./node\_modules/jquery/dist/jquery.js”

**Also install below Extensions which is in left side pane having four square circle logo**

Angular Language Service

Visual Studio Intellicode

Debugger for Chrome

**ANGULAR**

**Node.js:**Node.js is open-source, cross-platform JavaScript runtime environment that executes JavaScript code outside a browser. And it is developed by **Ryan Dahl** wrote in C, C++, and JavaScript. It is free to use and run on various platforms Windows, Linux, UNIX, Mac, etc. It can generate dynamic page content and can add, delete, and modify data in the database. Node.js file have extension **“.js”.**

**NPM**: NPM is a package manager in node.js packages. [**www.npmjs.com**](http://www.npmjs.com/) hosts millions of free packages to download and use. The NPM program is installed in our computer when we install node.js. A package in node.js contains the entire package we need for a module. Downloading a package is very compatible. Open the command-line interface and say NPM to download the packages which we want. If we want to download a package called **“upper-case”:**

**Angular CLI:**Angular CLI is known as Angular Command-line interface. An Angular CLI is used for creating a project. It can be used to create content, services, pipes, directives, and many more; also, it helps in building, serving, testing, etc. It makes Angular Development workflow much easier and faster.

**Git:**Git is a distributed version control system used for tracking changes in source code during software development. It is designed for spreading work among programmers. But it is used to track any set of change in files. **Linus Torvalds** created it in 2005 for developing the Linux kernel. Git is free and open-source software distribution system under the term of the GNU(General Public License) version 2.

**MVVM**- Model View View Model

**Building blocks are Modules.** Modules are just lines of code which can be imported/exported. Every angular module has atleast one module which is the **root module called as AppModule. Module is mechanism to group components,pipes,directives,and services.**

**Components**-Modules are made of components. Components control a portion of view. E.g. we can have a component for Navigation, main body,etc. Again as module it has atleast one component which is the **root component called AppComponent.** All other components will be nested inside the Root component and each component will have an HTML template to represent a view and a class that controls the logic of that particular view.

**Services** – Modules also has Services. They are basically classes which contain business logic of your application.

**Angular app – one or more modules**

**Module – one or more components or services**

**Components – HTML + Class**

**Services – Business logic**

Modules interact and render the view in the browser.

Angular 8 app files which are used in our project are below:

* **Src folder:** It is the folder which contains the main code files related to our angular application. It is the entry to our application. Inside it we have the app folder which has the ***app.module.ts*** file which is the root module of our application and the ***app.component.ts*** which is the root component of our application.
* **app folder:** It contains the files which we have created for app components.
* **app.component.css**: The file contains the**CSS**(cascading style sheets) code in our app component.
* **app.component.html:**The file contains the **HTML** file related to its app component. It is the template file which is specially used by angular to the data binding.
* **app.component.spec.ts:**This file is a unit testing file is related to the app component. This file is used across with more other unit tests. It is run from angular CLI by command ng test.
* **app.component.ts:**It is the essential typescript file which includes the view logic beyond the component.
* **app.module.ts:**It is also a typescript file which consists of all dependencies for the website. The data is used to define the needed modules has been imported, the components to be declared, and the main element to be bootstrapped.

**Other Important Files in Angular 8**

* **package.json**: It is the npm configuration file. It includes details of our website’s and package dependencies(libraries and modules) that are required for your angular application to run. These packages will be installed when you run the *ng new project* command. These packages will then get installed inside ***node\_modules*** folder
* **package-lock.json:** This is an auto-generated and transforms file that gets updated when npm does an operation related to the node\_modules or package.json file.
* **angular.json:**It is a necessary configuration file related to our angular application. It defines the structure of our app and includes any setting to accomplish with the application.
* **.gitignore**: The record is related to the source code git.
* **.editorconfig:** This is a standard file which is used to maintain consistency in code editors to organizing some basics. such as indentation and whitespaces.
* **Assets folder:**This folder is a placeholder for the resource files which are used in the application such as images, locales, translations, etc.
* **Environment folder:**The environment folder is used to grasp the environment configuration constants that help when we are building the angular application.
* **Browser list:** This file specifies a small icon that appears next to the browser tab of a website.
* **favicon.ico:**It defines a small image that appears next to the browser tab of any website.
* **Index.html:**It is the entry file which holds the high-level container for the angular application.
* **karma.config.js:**It specifies the config file in the karma Test runner, Karma has been developed by the AngularJS team and can run tests for AngularJS and Angular 2+.
* **main.ts**: This is the main ts file that will run; first,  It is mainly used to define the global configurations.
* **polyfills.ts**: The record is a set of code that can be used to provide compatibility support for older browsers. Angular 8 code is written in **ES6+** specifications**.**
* **test.ts:**It is the primary test file that the Angular CLI command ng test will apply to traverse all the unit tests within the application.
* **styles.css:** It is the angular application uses a global CSS.
* **tsconfig.json:**This is a typescript compiler of the configuration file.
* **tsconfig.app.json:**It is used to override the ts.config.json file with app-specific configurations.
* **tsconfig.spec.json:** It overrides the tsconfig.json file with the app-specific unit test cases.

When we use the ***ng serve*** command which starts our application, the execution comes to ***main.ts*** file where it kickstarts the app module. In app module it kickstarts the app component. This has app component has two things the HTML and the class to control logic.

**COMPONENTS**

The component is the **basic building block of Angular**. It has 3 parts -

**Template – View – HTML**

**Class – Code – TypeScript – Data & Methods(Logic)**

**Metadata – Information needed by angular if the particular class is angular component or a regular class- metadata is defined using a decorator which is a feature in Typescript (decorator is function that provides information about the class attached to it)**

Components are defined using a **@component** decorator and a tree of the angular component. It makes our complex application in reusable parts which we can reuse easily.

A component is an independent block of an extensive system that communicates with the other building blocks of the systems using inputs and outputs. It has an associated view, data, and behavior and has parent and child components.

**Default Code:**

**Import { Component } from ‘@angular/core’;**

**@Component ({**

Selector: ‘app-root’,

templateUrl: ‘./app.component.html’ ,

styleUrls: [‘./app.component.css’]

**})**

**export class AppComponent {**

title = ‘myfirstapp’;

**}**

**AppComponent** is a class. And **title** is property and not any methods.

To this class we have the **metadata** attached in the form of decorator to be specific **component decorator.**

**@Component({}).** The component decorator is basically a function that attaches to the class right below it. And it tells angular that it is not a plain class but a component. It contains metadata in form of –

* **selector:** It specifies the tag that can be used to call this component in HTML templates just like the standard HTML tags. A selector instructs Angular to instantiate this component wherever it finds the corresponding tag in template HTML.
* **templateUrl:** It indicates the path of the HTML template that will be used to display this component.
* **styleUrls:** It specifies an array of URLs for CSS style-sheet in the component.

For our case selector is ‘app-root’ which is a custom HTML tag used in index.html file. So angular renders the app.component.html in <app-root> tag in index.html file.But for what to render a **templateUrl** tag is used which specifies which template to render in selector. And similarly a **styleUrls** which provides CSS styles to the template.

To create a new component use command - **ng g c {component\_name}**

This will create a new component in app folder along with the existing app.component . This also includes similar class file, HTML file and CSS file.

Angular CLI also creates a entry in app.module. It automatically imports the component and adds an entry in **declarations** **array** in the **NgModule** **decorator** which is in **app.module.ts** file. This **declarations array** contains the list of all components used by angular application.

Now to include this in HTML simply create a HTML tag in component HTML having same name as selector in test class. From now on every HTML tag should be created in component HTML (root component).

The selector tag can be specified with 3 types-

**1)** The normal way as above by creating the HTML tag with same name as selector name.

**2)** By creating an HTML element with . / # for class or id property and using that in selector with . / #

e.g. <div class=”app-test”></div> @Component({selector = ‘.app-test’})

**3)** By enclosing the selector name with [] and directly providing the HTML tag with same name attribute

e.g. <div app-test></div> @Component({selector = ‘[app-test]’})

The templateUrl can be replaced with inline HTML by using template metadata(Backticks should be used).

@Component({

template= `<div>Template Format Selector</div>`

})

**INTERPOLATION (**Output Data to HTML**)**

Interpolation refers to embedding expressions into marked up text. By default, interpolation uses as its delimiter the double curly braces, {{ and }}. You can do calculations, concatenate strings , use methods from class, javascripts functions etc. But you cannot perform assignments({{a = 1+1}}) or access Javascript global variables(They can't refer to window or document. Additionally, they can't call console.log() or Math.max()) with interpolation.

@Component({

  selector: 'app-test',

  template: `<h2>Welcome {{name}}</h2>

  <h2>{{2+2}}</h2>

  <h2>{{"Welcome " + name}}</h2>

  <h2>{{name.length}}</h2>

  <h2>{{name.toUpperCase()}}</h2>

  <h2>{{greetUser()}}</h2>`,

  styleUrls: ['./test.component.css']

})

export class TestComponent implements OnInit {

  public name = "Shivam Bagi"

  constructor() { }

greetUser(){

  return "Hello " + this.name;}

  ngOnInit(): void {

  }

}

**PROPERTY BINDING (**Output Data to HTML**)**

**To attach functions,data from class to attributes to HTML controls**

**HTML Attribute vs. DOM Property**

When we create a HTML input element textbox with pre-filled attribute value = shivam. <input id="{{myId}}" type="text" value="Shivam">

Now in the browser we get a textbox with prefilled value(shivam).

If we inspect in the console ($0 is current element)

$0.getAttribute(‘value’) – shivam

$0.value – shivam

Now if change the prefilled text(bagi) we get

$0.getAttribute(‘value’) – shivam

$0.value – bagi

The attribute does not change but values change. Attributes and properties do not change. Attributes define HTML while properties define DOM. Attributes initialize DOM properties and then they are done. Attribute values cannot change one they are initialized. Property values can change. HTML attribute value specifies the initial value while DOM specifies current value.

Interpolation has some drawbacks , it cannot be used set values which are non-string such as Boolean. When we want to set disabled attribute of textbox to false we cannot use theattribute directly or use interpolation.

e.g. <input disabled=”false” id="{{myId}}" type="text" value="Shivam"> - WRONG

<input disabled=”{{false}}” id="{{myId}}" type="text" value="Shivam"> - WRONG

<input [disabled]=”false” id="{{myId}}" type="text" [value]="Shivam"> - RIGHT

Class {

Public isDisabled = false

}

<input [disabled]=”isDisabled” id="{{myId}}" type="text" value="Shivam"> - RIGHT

**myId** is a property declared in class component.

To bind data to a property of any element, we use square braces[ ]. In property binding, we link property of a DOM element to a field. Angular internally converts string interpolation into property binding. Does not work through interpolation

**CLASS BINDING –** Used for binding a single or group of HTML attributes to a HTML element

**Binding to a single class**

To create a single class binding, use the prefix class followed by a dot and the name of the CSS class(or any other HTML attribute)—for example, [class.sale]="onSale". Angular adds the class when the bound expression, onSale is truthy, and it removes the class when the expression is falsy—with the exception of undefined. See [styling delegation](https://angular.io/guide/style-precedence#styling-delegation) for more information.

**Binding to multiple classes**

To bind to multiple classes, use [class] set to an expression—for example, [class]="classExpression". The expression can be a space-delimited string of class names, or an object with class names as the keys and truthy or falsy expressions as the values. With an object format, Angular adds a class only if its associated value is truthy.

@Component({

  selector: 'app-classbinding',

  template: `

<h2 [class]="successClass">Shivam Bagi</h2> - Single Binding

<h2 [class.text-danger]="hasError">Shivam Bagi</h2> - Single Binding

  <h2 [ngClass]="messageClass">Shivam Bagi</h2> - Multiple Binding

`,

  styles: [`.text-success{color: green;}

  .text-danger{color: red;}

  .text-special{font-style: italic;}`]

})

export class ClassbindingComponent implements OnInit {

public successClass = "text-success";

public isSpecial = true;

public hasError = true;

public messageClass = { - Multiple binding class having many attributes

  "text-success" : !this.hasError,

  "text-danger" : this.hasError,

  "text-special" : this.isSpecial

}

  constructor() { }

  ngOnInit(): void {

  }

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Binding Type** | **Syntax** | **Input Type** | **Example Input Values** |
| Single class binding | [class.sale]="onSale" | boolean | undefined | null | true, false |
| Multi-class binding | [class]="classExpression" | string | "my-class-1 my-class-2 my-class-3" |
| {[key: string]: boolean | undefined | null} | {foo: true, bar: false} |
| Array<string> | ['foo', 'bar'] |

**STYLE BINDING**

|  |  |  |  |
| --- | --- | --- | --- |
| **Binding Type** | **Syntax** | **Input Type** | **Example Input Values** |
| Single style binding | [style.width]="width" | string | undefined | null  <h2 [style.color]="’orange’">Shivam Bagi</h2>  <li [ngStyle]="{'color':i === count ? 'orange' : 'red' }"></li> | "100px" |
| Single style binding with units | [style.width.px]="width" | number | undefined | null  <h2 [style.width.px]="width">Shivam Bagi</h2>  public width = 100; | 100 |
| Multi-style binding | [[style](https://angular.io/api/animations/style)]="styleExpression" | String (Similar to Class Binding)  <h2 [ngStyle]=" styleClass">Shivam Bagi</h2> | "width: 100px; height: 100px" |
| {[key: string]: string | undefined | null} | {width: '100px', height:100px'} |

**EVENT BINDING**

Event binding is a data-binding from an element to a component. User actions such as keystrokes, mouse movement, clicks, and touches can be bound to the component property using event binding. In the event binding, the target will be an event name. It is one-way data binding and sends information from the view to the component class. It is opposite of property binding.

<button (click)="onClick($event)">Button</button>

export class EventbindingComponent implements OnInit {

  onClick(event){

    console.log(event);

    this.greeting = "Learn Angular";

  }

}

To pass data to Class from event use $event.

**TEMPLATE REFERENCE VARIABLES**

Template variables help you use data from one part of template to another part of the same template. It can be used to get value from textbox or any other such inputs and use them or bind them to other views or components. You use the hash symbol, #, to declare a template variable. It can be then used or passed in HTML actions to functions.

<input #inputelement type="text">

<button (click)="onClick(inputelement.value)">Log</button>

**TWO WAY BINDING**

For Two-Way binding to work, you need to enable the ngModel directive. This is done by adding the FormsModule to imports array in the AppModule.

Two-way binding combines property binding with event binding.

Angular's two-way binding syntax is a combination of square brackets and parentheses, [()]. The [()] syntax combines the brackets of property binding, [], with the parentheses of event binding, ().

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

{{name}}

export class TwowaybindingComponent implements OnInit {

public name = "";

}

ngModel is not a known property of input elements hence it needs to be imported from angular/forms in app.module.ts

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

And add FormsModule in imports array of @NgModule class

In the above example,It will simultaneously bind the typed value in textbox to name variable in the class and simultaneously display the value in the view(i.e. in the {{name}} area).

**DIRECTIVES**

Directives are instructions in the **DOM** (Document Object Model). It specifies how to place our business logic in Angular. The directive is markers on a DOM element that tell Angular to attach a specified behavior to that DOM element or even transform the DOM element and its children. Mostly directives in Angular starts with ng- where **ng** stands for **Angular,**and it extends the HTML. “ \* ” is required before structural directives as it specifies angular it is part of angular directives and not custom directives.

**STRUCTURAL DIRECTIVES**

Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, or manipulating elements.

Structural directives are easy to recognize. An asterisk (\*) precedes the directive attribute name as in this example.

**ngIf**

<div \*[ngIf](https://angular.io/api/common/NgIf)="true">Welcome</div>

For Else create a new ng-template tag and assign a #name to it and use it in else after Boolean value in ngIf

<div \*[ngIf](https://angular.io/api/common/NgIf)="true; else ElseBlock">Welcome</div>

<ng-template #ElseBlock>Else Welcome</ng-template>

Two ng-template can also be used for If-else statements.

<div \*ngIf="value; then thenblock1; else elseblock1"></div>

<ng-template #thenblock1>

    <h2>Learn Angular 1</h2>

</ng-template>

<ng-template #elseblock1>

    <h2>Learn Angular Fast 1</h2>

</ng-template>

Using Only ng-template(Show is in ts file)

<ng-template [ngIf]="show==’red’” >

    <h2>Red Block</h2>

</ng-template>

<ng-template [ngIf]="show==’green’” >

    <h2>Green Block</h2>

</ng-template>

**ngSwitch**

The Angular NgSwitch is actually a set of cooperating directives: [NgSwitch](https://angular.io/api/common/NgSwitch), [NgSwitchCase](https://angular.io/api/common/NgSwitchCase), and [NgSwitchDefault](https://angular.io/api/common/NgSwitchDefault).

[NgSwitch](https://angular.io/api/common/NgSwitch) itself is not a structural directive. It's an attribute directive that controls the behavior of the other two switch directives. That's why you write [[ngSwitch](https://angular.io/api/common/NgSwitch)], never \*[ngSwitch](https://angular.io/api/common/NgSwitch).

[NgSwitchCase](https://angular.io/api/common/NgSwitchCase) and [NgSwitchDefault](https://angular.io/api/common/NgSwitchDefault) are structural directives. You attach them to elements using the asterisk (\*) prefix notation.

    <div [ngSwitch]="color"> // color is property in ts file

<div \*ngSwitchCase="'red'">Red Color</div>

<div \*ngSwitchCase="'orange'">Orange Color</div>

<div \*ngSwitchCase="'green'">Green Color</div>

<div \*ngSwitchDefault>Pick Again</div>

    </div>

**ngFor**

<div \*ngFor="let color of colors" >

    <h1>{{color}}</h1>

</div>

export class NgForComponent implements OnInit {

public colors = ["red","orange","green","yellow"];

}

<ul \*ngFor="let user of arr;let i = index">

    <!-- <li \*ngIf="i >= count">{{user}}</li>  -->

    <li [ngClass]="{error: i == count}">{{user}}</li>

</ul>

**COMPONENT INTERACTION**

Communication between components i.e. passing data between parent component(app.component) and child component(other components).

**Pass data from parent to child with input binding**

* Create a variable in parent class.

public name = “shivam”;

* To send it to child class include it in the **selector HTML tag** by class binding in app.component.html file.

<app-test [parentData]=”name”></app-test>

* To receive it in child component, declare a variable with the same name of the variable passed in HTML element tag. Also to differentiate between normal variable and Input variable use the **@Input()** decorator on the declared variable. You have to import it from @angular/core, simply add the **Input** keyword after OnInit in child component import line. You can also declare variable with different name and pass the same name variable in parenthesis of @Input in single inverted commas.

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

@Input() public 'parentData'; // @Input('parentData') public name;

* Now use the variable name in HTML tag , and you have your parent component data in your child component.

<h2>{{"Hello " + name}}</h2>

**Pass data from child to parent with EventEmitter**

Since you have a selector in child component you can get the data from parent through the HTML selector tag to the child class. But such is not possible vice-versa.

* Similar to Input import above we have to import EventEmitter for this purpose.

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

* Create a instance of **EventEmitter** class

public childevent = new EventEmitter;

* To send the childevent to parent we use the **output** decorator, which also needs to be imported.

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

@Output() public childevent = new EventEmitter;

* To use this event create a button or any other way and create a function in child class. And use the function in button tag created in child HTML.

fireEvent(){

   this.childevent.emit('Child Component');

}

* To listen to the emitted event, in the app.component.html similar to binding click event, we can bind to custom events. So within the HTML selector tag capture the childEvent and assign a property name and to it the $event. This $event variable will get the value from the child class.

<app-componentinteraction (childevent)="childevent($event)" ></app-componentinteraction>

* Now declare the same property declared in the custom event(in our case message) in the parent class. And now can be used in the parent HTML.

export class AppComponent {

public name = "Parent Component";

public message = "";

title = 'binding';

childevent(data)

{

this.message=data;

}

}

**ViewChild**

There is another way getting access of HTML elements in typescript component, similar to Local Reference Variable. In Local Reference Variable we get access to HTML element after a method is called, but sometimes we need to get hold of the element before a method is called. We use a decorator called **ViewChild** in component. It is present in “angular/core”.

<input #inputelement type="text">

<button (click)="onClick(inputelement.value)">Log</button> **//You can pass value attribute or**

// <button (click)="onClick(inputelement)">Log</button> **HTML element directly(but use value attribute in component)**

If you pass the element directly you will get the whole element when you try to log.

Create a ViewChild decorator having the name of Local ref var. in parenthesis passed to it.

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

export class Component {

@ViewChild(‘inputelement’) inputelement; //@ViewChild(‘inputelement’) inputelement:ElementRef;

onClick(inputelement){ // onClick(inputName: HTMLInputElement){

    console.log(inputelement); //console.log(inputelement.nativeElement.value);

   }

}

**ContentChild**

In ViewChild you can only access the values form that component html only. It is similar to ViewChild but only differs in data sharing from component to component.Since it is not available in view. This is not initialized until ngAfterContentInit lifecycle hook.

<childcomponent>

<p #othcomppara></p>

</childcomponent>

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

export class ChildComponent {

@ContentChild(‘othcomppara) othcomppara:ElementRef;

ngOnInit{    //only first time the value will not be logged since content is not initailized

console.log(othcomppara.nativeElement.value);

   }

ngAfterContentInit{     //the value will be logged since content is initailized

console.log(othcomppara.nativeElement.value);

   }

}

**Content Projection (ng-content)**

You use the **<ng-content></ng-content>** tag as a placeholder for that dynamic content, then when the template is parsed Angular will replace that placeholder tag with your content. Think of it like curly brace interpolation, but on a bigger scale. The technical term for this is “content projection" because you are *projecting* content from the parent component into the designated child component.

Below the content that passed form the parent component into the child component element will be rendered into the <ng-content><ng-content>. This eliminates the need for passing data from component to component using property binding @Input,@Output decorators.

<child-component>some content</child-component>

<div>

<ng-content></ng-content>

</div>

**PIPES**

Use [**pipes**](https://angular.io/guide/glossary#pipe) to transform strings, currency amounts, dates, and other data for display. Pipes are simple functions you can use in [template expressions](https://angular.io/guide/glossary#template-expression) to accept an input value and return a transformed value. The | symbol is used after variable in interpolation.

{{title | uppercase}}

{{ birthday | [date](https://angular.io/api/common/DatePipe) | [uppercase](https://angular.io/api/common/UpperCasePipe)}}

There are two categories of pipes:

1. Pure
2. Impure

By default, pipes of angular are pure. Every pipe we have seen are pure and built-in pipes. We can make the pipe impure by setting the pure flag into false.

1. **Pure pipes**

Angular executes the pure pipe only when if it detects the perfect  change in the input value. The real difference is either in the shift to the primitive input value (Number, Boolean, Symbol, String) or a changed object reference (Array, Function, Object, Date).

1. **Impure pipes**

Angular executes the corrupted pipe during every component change detection cycle. The impure pipe is called often every keystroke or mouse-move.

* To create a custom pipe, we create a new ts file and use the code according to work, and we have to import Pipe, Pipe Transform from Angular/Core.
* Import this Appmodule also and add in declarations array. Now it can be used. Also you need to create new directive of NewCmpComponent and changeTextDirective

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

@Pipe ({

name: ‘sqrt’

})

Export class SqrtPipe implements PipeTransform {

transform (val: number) : number{

Return Math.sqrt(val);

}

}

**DEPENDENCY INJECTION**

Dependency injection, or DI, is a design pattern in which a class requests dependencies from external sources rather than creating them.

Angular's DI framework provides dependencies to a class upon instantiation. You can use Angular DI to increase flexibility and modularity in your applications.

<ul \*ngFor="let emp of employees">

<li>{{emp.name}}</li>

</ul>

export class EmployeeComponent implements OnInit {

public employees = [

  {"id":1,"name":"shivam","age":23},

  {"id":2,"name":"pranav","age":18},

  {"id":3,"name":"suresh","age":51},

  {"id":4,"name":"abhilasha","age":44}

];

}

In the above example we can generate the list from array in a view, supposedly we want to create a new array of only age but in a different view(html). It will be possible only if we create the same array in that component also, that is not desirable. To overcome this we can use services.

* Create a new service

To create a new service **ng g s *servicename***

After creating the service, create the same array inside the return of a new created method in that service class. Also create empty arrays in both components.

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

@Injectable({ **@Injectable()decorator** is a must if you

  providedIn: 'root' want to inject a service into another service

})

export class EmployeeService {

  getEmployees() {

    return [

    {"id":1,"name":"shivam","age":23},

    {"id":2,"name":"pranav","age":18},

    {"id":3,"name":"suresh","age":51},

    {"id":4,"name":"abhilasha","age":44}

    ];

  }

  constructor() { }

}

In the above case since the EmployeeService has no dependency on any other service it is not needed. But if it has it is needed. So it is recommended to always include the decorator irrespective of dependency. The @Injectable() decorator distinguishes the class into a service and a normal component.

* Register service with Injector

If we don’t register the service, it will be just another class. Angular DI follows a hierarchical pattern so if we register a service inside a component , it will be only accessible to its child elements and not to other components. Hence it is important to register a service in AppModule so that it can be accessible in all application.

To register a service we use the **providers metadata i.e.in providers array** in appmodule. Also it necessary to import the service.

@NgModule({

 providers: [EmployeeService],

})

* Declare as dependency in the created components

We need to declare the components as dependent on the service. To declare dependency, in the constructor(in parenthesis as an argument) create an private object of the service class and also needs to be imported in that component.

  constructor(private \_employeeService: EmployeeService) { }

To fetch the data , we need to assign the value to the instance of created empty array in that component from the object instance of the service in the ngOnInit.

ngOnInit gets called when the component gets initialized

ngOnInit(): void {

    this.employees = this.\_employeeService.getEmployees();

  }

**HTTP AND OBSERVABLES(Inside Services)**

Observable is simply a data source like (user Input)Events,HTTP Requests, Triggered in code.

**HTTPClientModule** is used to request data from servers.

* First you have to import the HTTPClientModule first in AppModule and add in the imports array.

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

imports: [

    BrowserModule,

    HttpClientModule

 ],

You importing the HTTPClientModule, we are also registering the HTTP Service with Angular injector. You don’t have register it explicitly by using the providers metadata

* Now to use HTTP Service in the service class we declare it as dependency in the constructor of service class. Also you have to import the **HTTPClient** and pass an object of **HTTPClient** in the constructor.

Now make a request in the getemployees using the HTTP object, since we are taking data from server.

This get request accepts a URL to get data. For this case we have just created a file and getting the data from there.

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

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

@Injectable({

  providedIn: 'root'

})

export class EmployeeService {

private url: string = "/assets/data/employees.json"

  getEmployees() {

    return this.http.get(this.url);

  }

  constructor(private http: HttpClient) { }

}

The get() method takes two arguments; the endpoint URL from which to fetch, and an *options* object that you can use to configure the request.

options: { headers?: [HttpHeaders](https://angular.io/api/common/http/HttpHeaders) | {[header: string]: string | string[]},

observe?: 'body' | 'events' | 'response',

params?: [HttpParams](https://angular.io/api/common/http/HttpParams)|{[param: string]: string | string[]}, reportProgress?: boolean, responseType?: 'arraybuffer'|'blob'|'json'|'text',

withCredentials?: boolean, }

* You can find out that the get request returns a observable when you hover over it. It is an object type, so you need to convert it to required format(e.g. string,array,etc). In our case, To convert into necessary format i.e. Array of employees we create a Interface having the fields of the array.

export interface IEmployee{

    id:number,

    name:string,

    age:number

}

In your service file import the interface and also observable namespace.

Typecast the argument parenthesis of get method to the interface type. And the getEmployees will return an observable of type employee array.

import {IEmployee} from './employee';

import {Observable} from 'rxjs;

private url: string = "/assets/data/employees.json"

  getEmployees(): Observable<IEmployee[]>  {

    return this.http.get<IEmployee[]>(this.url);

  }

* Now you have to subscribe to the observable where you want to call the getEmployees method. The subscribe method takes argument of assigning the data from the observable to the local array inside the respective component. The data variable is just an variable used to assign the values to the local array.

ngOnInit(): void {

    this.\_employeeService.getEmployees()

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

  }

**HTTP Error Handling**

You have to import the throwError(throw in old versions) and catchError(catch in old versions) in your service file since get request is also in that file.

import {Observable,throwError} from 'rxjs';

import {catchError} from 'rxjs/operators';

Use the pipe function of the get method and pass the catchError to it and the errorHandler method(created manually to return the error) to the catchError.

getEmployees(): Observable<IEmployee[]>  {

    return this.http.get<IEmployee[]>(this.url).pipe(catchError(this.errorHandler))

}

Use the throwError to retun the error message or default msg.

 errorHandler(error: HttpErrorResponse){

    return throwError(error.message||"Server Error");

  }

Also in the component files where the getEmployees is being called we need to pass the errpr using => function just like the data keyword was passed. And assign the error to a local variable. You can now pass the variable to the html to display error.

public errorMsg;

  ngOnInit(): void {

    this.\_employeeService.getEmployees()

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

      error => this.errorMsg = error);

  }

**ROUTING AND NAVIGATION**

Routing is optional in angular. It can be included in the project at the start, the Angular CLI command prompt ask to include routing or not(newer versions). In older versions it was default set to no. It can be included by using the CLI command ***ng new routing-app –routing.***

The <base href="/"> tag in the index.html is used to manipulate the URL while navigating.

The app-routing.module.ts file contains the routing module for the application and this is where routes are configured.

In app.module import the app-routing module and add it in the imports array.

In the app-routing, the const routes which is strongly type to Routes object takes the all routes. Each route in this array is a JavaScript object that contains two properties. The first property, path, defines the URL path for the route. The second property, component, defines the component Angular should use for the corresponding path.

const routes: Routes = [

  {path:'employeelist', component: EmployeeComponent},

  {path:'employeedetaillist', component: EmployeedetailComponent}

];

**The Router-outlet**

The Router-outlet is a directive accessible from the router library where the Router inserts the component and gets matched based on the current browser’s URL. We can add multiple outlets in our angular application, which enables us to implement advanced routing scenarios.

**<router-outlet></router-outlet>**

The html attribute for routing is-

<button **routerLink**="/employeelist">Employee List</button>

    <button **routerLink**="/employeedetaillist">Employee Detail List</button>

Where the routerlink is the path specified in the Routes object in app-routing file.

**Route Params**

To creating the routes with parameters is a common feature in web apps. The angular Router allows us to access parameters in different ways.

We can create a router parameter using the colon syntax.

{path:'departments/:id',component: DepartmentdetailsComponent}

Now to where you want to route to in that component file in the constructor pass the object of the Router.In our case we have created list of buttons and provided the click functionality. In that click function we are passing the department object. In The function we use the navigate function of the router object created. Pass the URL in the navigate function along with the department ID to construct the URL inside it.

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

import {Router} from '@angular/router'

@Component({

  selector: 'app-departments',

  template: `

  <h2>Department List</h2>

<ul class="items">

    <li \*ngFor="let department of departments">

    <button (click)="onClick(department)">{{department.name}}</button>

    </li>

</ul>

  `,

  styles: []

})

export class DepartmentsComponent implements OnInit {

departments = [

  {"id": 1, "name": "angular"},

  {"id": 2, "name": "react"},

  {"id": 3, "name": "c#"},

  {"id": 4, "name": "MVC"},

  {"id": 5, "name": "WebAPI"}

];

**constructor(private router: Router) { }**

  ngOnInit(): void {

  }

  onClick(department){

**this.router.navigate(['/departments',department.id])**

  }

}

To access the passed parameter in the URL in the view. We have to use the ActivatedRouteService. Import it in the second file where you want to navigate to and pass it as an argument in the in the constructor of the component. And in the ngOnInit method we get the value from the id.

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

export class DepartmentdetailsComponent implements OnInit {

public departmentid;

**constructor(private route: ActivatedRoute) { }**

**ngOnInit(): void {**

**let id = parseInt(this.route.snapshot.paramMap.get('id'));**

**this.departmentid = id;**

**}**

}

You can also pass the parameter and path in HTML element also using the routerLink attribute. It is similar to passing to the click function. The ngOnInit picks the id as well similar to the previous way of passing the parameter.

<button [routerLink]="['/departments', department.id]">{{department.name}}</button>

When you add route path to the routes object, you can also add a route that returns a page when an invalid URL is entered. This will redirect it to the Error page when the URL does not match the paths. It need to be entered at the last so that it checks for all specified paths and if not found then returns the specified error page.

{path: "\*\*", component: ErrorpageComponent}

You also encounter another problem when you do the above check. You will get the error page even when you do not enter the URL(localhost:4200/). In this case you can specify a default route when the URL is empty. This should be on the top.

{path:'',component:DepartmentsComponent}

OR

You can use the redirectTo property and also the pathMatch property inside the routes object after path property. The redirectTo property accepts the path URL and the pathMatch tells the router how to match the URL. There are 2 values - 1) Prefix 2) Full

**{path:'',redirectTo:'/departments', pathMatch: 'full'},**

**paramMap**

To get information from a route:

1. Import [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) and [ParamMap](https://angular.io/api/router/ParamMap) to your component.

import { [Router](https://angular.io/api/router/Router), [ActivatedRoute](https://angular.io/api/router/ActivatedRoute), [ParamMap](https://angular.io/api/router/ParamMap) } from '@angular/router';

These import statements add several important elements that your component needs. To learn more about each, see the following API pages:

* + [Router](https://angular.io/api/router)
  + [ActivatedRoute](https://angular.io/api/router/ActivatedRoute)
  + [ParamMap](https://angular.io/api/router/ParamMap)

1. Inject an instance of [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) and [ParamMap](https://angular.io/api/router/ActivatedRoute)  by adding it to your application's constructor:

constructor(private router: Router, private route: [ActivatedRoute](https://angular.io/api/router/ActivatedRoute)) { }

1. Update the ngOnInit() method to access the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) and track the id parameter:

ngOnInit() {

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

this.departmentid = parseInt(params.get(‘id’));

});

}

This is a better way of getting data from routes than the previous one as it allows to get data from URL’s changed from the code.

**For active tabs CSS property**

<ul class="nav navbar-nav">

     <li **routerLinkActive="active"**><a routerLink="/recipes">Recipes</a></li>

     <li **routerLinkActive="active"**><a routerLink="/shopping-list">Shopping List</a></li>

</ul>

**Relative Navigation**

The path we are using so far was absolute and it begins with ‘/’ forward slash. So when we are using an absolute path we are lacking the flexibility of routing in the application. On every single change in route requirements, it requires you to change the calling references also. So for flexibility in usage, we will use relative routing. The navigate method takes two arguments. The first one is the link parameters array; within this we specify the path first then optional parameters. The link and optional parameters will be relative to this.route.

If the First segment of the route starts with “/“, then the path is considered to be Absolute path

If the First segment begins with “./” or it does not begin with a slash, then the path is considered to be the relative path.

And if the First segment begins with “../“, then the path is relative to the parent route. (one level up)

**this.router.navigate(['../Detail'], { relativeTo: this.activatedRoute });**

**Child Routes/Nested Routes**

Nested routes are routes within other routes. When some routes may only be accessible and viewed within other routes it may be appropriate to create them as child routes.

To specify child routes in angular, we use the Children array property in the routes array in app-routing module.

This children should be inside the path object of which view you to put it inside.

const routes: Routes = [

  {path:'departments',component: DepartmentsComponent},

  { path:'departments/:id',

    component: DepartmentdetailsComponent,

    children:[

      {path:'overview',component:OverViewComponent}

      {path:'details',component:DetailsComponent},

    ]

  }

];

Where would the components for these child routes be displayed? Just like we had a **<router-outlet></router-outlet>** for the root application component, we would have a router outlet inside the DepartmentsComponent component.

**ANGULAR FORMS**

**Flow ==**

**Template(Collect Data) 🡪 Class(Bind Data) 🡪 Service(Send Data) 🡪 Server**

In Angular 8, there are two approaches to handle the user’s input through forms:

**Reactive forms**

Most code is written in component class.

**Template-driven forms**

If we have fundamental form requirements and logic that can be managed in the template and use template-driven forms. Most code is written in component template(html).

|  |  |  |
| --- | --- | --- |
| **Index** | **Reactive Forms** | **Template-driven Forms** |
| **Setup (form model)** | Responsive forms are more explicit. They are created in the component class. | Template-driven forms are less explicit. Directives create them. |
| **Data model** | Structured | Unstructured |
| **Predictability** | Synchronous | Asynchronous |
| **Form Validation** | Functions | Directives |
| **Mutability** | Immutable | Mutable |
| **Scalability** | Low-level API access | Abstraction on top of APIs. |

**Template-driven forms**

First step irrespective of any approach is to import the FormsModule and also add it to imports array in app.module.ts file.

import {FormsModule} from '@angular/forms'

Anytime we use Form tag in HTML after forms module is used, angular works in background.

In html when we create form tag we use the template reference variable and assign the string **ngForm** directive to it.

ngForm exports itself as the string ngForm and by assigning it to a reference variable we have reference to the directive itself, i.e. the directive gives access to the values of the form controls using the value property.

Also place the **ngModel** to each input controls in the form tag, it will help ngForm directive track the values of the controls. For retrieving names from the inputs we must have **name property** of the input HTML along with ngModel.

**ngModelGroup** directive can be used to club together a set of inputs.

<form #simpleForm="ngForm" (ngSubmit)="getUserValue(simpleForm.value)">

  <input type="text" ngModel name="userName" />

  <input type="text" ngModel name="age" />

  <input type="password" ngModel name="password" />

  <button>Get Values</button>

</form>

getUserValue(value)

  {

    console.log(value);

  }

**Important thing in template driven forms is the above three directives ngForm, ngModel, ngModelGroup**

For Two way binding to take place i.e. data from template to model(class) and vice-versa. We use the banana in a box i.e. **Two way binding[()] on the ngModel directive for each control.**

**To bind the control input to the model property use the ngModel with Two-way binding assigning the model property name to it.**

<input [(ngModel)]="userModel.name">

**Creating User Class**

export class User {

    constructor(

    public name : string,

    public email : string,

    public telephone : number,

    public course : string,

    public timepreference : string,

    public subscribe : boolean

    ){}

}

**Creating the object of User Class. We need to import the class first.**

import {User} from './user';

export class AppComponent {

  title = 'templatedrivenforms';

  topics = ['Angular', 'React', 'Vue'];

  userModel = new User('shivam','s@s.com',64654564,'','Morning',true);

}

**HTML**

<div class="container-fluid">

<h1>Bootcamp Enrollment Form</h1>

<form #userForm="ngForm">

{{userForm.value | json}}   *<!--To get values dynamically from the controls to page, userForm is Form template reference variable -->*

<hr/>

 {{userModel | json}}       *<!--To get values from the model class to page, userModel is object created in app.component.ts file -->*

  <div class="form-group">

    <label>Name</label>

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

  </div>

  <div class="form-group">

    <label>Course</label> <br>

    <select class="custom-select" name="userCourse" [(ngModel)]="userModel.course">

      <option value="" selected>Choose Course</option>

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

    </select>

  </div>

  <div class="form-group">

    <label>Timing Slot</label>

    <div class="form-check">

<input class="form-check-input" type="radio" name="timepreference" value="morning" [(ngModel)]="userModel.timepreference">

    <label class="form-check-label">Morning</label>

  </div>

 <div class="form-check">

<input class="form-check-input" type="radio" name="timepreference" value="evening" [(ngModel)]="userModel.timepreference">

    <label class="form-check-label">Evening</label>

  </div>

 </div>

*<!-- <div ngModelGroup="address">*

*<div>*

*<label>Street</label><input type="text" class="form-control" name="userStreet" ngModel>*

*<label>City</label><input type="text" class="form-control" name="userCity" ngModel>*

*<label>State</label><input type="text" class="form-control" name="userState" ngModel>*

*</div>*

*</div> -->*

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

</form>

</div>

* **Validations**

**Validation directive are provided like-**

**Visited Control = ng-touched (if true) // ng-untouched**

**Value Changed = ng-dirty (if true) // ng-pristine**

**Value Valid = ng-valid (if true) // ng-invalid**

<input type="text"

required #name="ngModel" [class.is-invalid]="name.invalid && name.touched"

class="form-control" name="userName" [(ngModel)]="userModel.name">

Here [class.is-invalid] is bootstrap class(is-invalid) it will be true if name is invalid and touched.

* To display error messages use the \*ngIf directive or similar validation directives on the HTML message tag.

<input required pattern="^\d{10}$" #phone="ngModel"

[class.is-invalid]="phone.invalid && phone.touched" type="tel"

class="form-control" name="userTel" [(ngModel)]="userModel.telephone">

*<!-- <small class="text-danger" [class.d-none]="phone.valid || phone.untouched">Phone No. is required and must be 10-digit</small> -->*

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

    <small class="text-danger" \*ngIf="phone.errors.required">Phone No. is required</small>

    <small class="text-danger" \*ngIf="phone.errors.pattern">Phone No. must be 10-digit</small>

  </div>

* **POSTING DATA TO SERVER**

Create a service and import the User model, catchError and throwerror for errorhandling.

For URL refer below points on creating express server.

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

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

import { User } from './user';

import { catchError } from 'rxjs/operators';

import { throwError } from 'rxjs';

@Injectable({

  providedIn: 'root'

})

export class EnrollmentService {

  \_url = 'http://localhost:3000/enroll';

  constructor(private \_http : HttpClient) { }

  enroll(userobj : User){

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

.pipe(catchError(this.errorHandler));

  }

  errorHandler(error: HttpErrorResponse)

  {

return throwError(error);

  }

}

**Also import HttpClientModule in app.module.ts and add in imports array.**

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

**Component.ts**

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 {

  title = 'templatedrivenforms';

  topics = ['Angular', 'React', 'Vue'];

submitted = false;

errorMsg = '';

constructor(private \_enrollobj:EnrollmentService){

}

  userModel = new User('shivam','s@s.com',6465456485,'Angular','morning',true);

  onSubmit(){

    this.submitted = true;

    this.\_enrollobj.enroll(this.userModel)

    .subscribe(

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

*// error => console.error('Error',error)*

      error => this.errorMsg = error.statusText

      )

  }

}

**Use the ngSubmit directive on the form tag for submitting data.**

<form #userForm="ngForm" \*ngIf="!submitted" (ngSubmit)="onSubmit()" novalidate></form>

**To create a server using Express**

**Create a folder and in its command terminal type**

**npm init -- yes**

**npm install --save express body-parser cors**

**node server** == To run the server

const express = require('express');

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

const cors = require('cors');

const port = 3000;

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(401).send({"message": "Data Received"})*

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

})

app.listen(port,function(){

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

})

**REACTIVE FORMS**

Code and logic reside in component class

No two way binding

Dynamic Form Fields

For complex structure

* **First step is to import ReactiveFormsModule and add it to imports array**

import { ReactiveFormsModule } from '@angular/forms'

**FormGroup** and **FormControl** are building blocks of reactive forms which are derived from **ReactiveFormsModule.**

In reactive forms the form is represented by model in component class. The HTL controls are defined as instances of FormControl class and the collective is called as FormGroup class.

**Step 1🡪** Define HTML form in component template. i.e. create HTML form.

<form>

<label>UserName</label>

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

<label>Password</label>

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

</form>

**Step 2🡪** Define form model in component class.

Import the FormControl and FormGroup in the class file.

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

Create a new FormGroup which takes FormControls as arguments.

registration = new FormGroup();

You have add the FormControls to it as objects.

registration = new FormGroup({

    userName : new FormControl(''),

    passWord : new FormControl(''),

    confirmPassword : new FormControl('')

  });

FormGroups can be **nested**.For this you will have to create a **nested div** in the HTML and associate that div with nested FormGroup and the individual controls too.

registration = new FormGroup({

    userName : new FormControl(''),

    passWord : new FormControl(''),

    confirmPassword : new FormControl(''),

    address : new FormGroup({

      city : new FormControl(''),

      state : new FormControl(''),

      pincode : new FormControl('')

    })

  });

**Step 3 🡪** Use the directives to associate the form model with the view.

Assign the respective FormGroup and FormControl to the HTML tags.

Use the **formGroup** directive to associate the **FormGroup** instance to the form tag. And the **formControlName** directive to associate the **FormControl** to HTML tags.

<form [formGroup]="registration">

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

<input formControlName="confirmPassword" type="password" class="form-control">

</form>

* **To set values from the model to the HTML controls.We use the setValue function of the FormGroup**.

<button (click)="loadapidata()" type="button" class="btn btn-primary ml-6">Load</button>

loadapidata(){

    this.registration.setValue({

      userName:"Shivam",

      passWord:"Shivam123",

      confirmPassword:"Shivam123"

    });

  }

The **setValue** function takes an **object** similar to the FormGroup model created.

By this you can load the data from model in the HTML.

setValue has a disadvantage that it uses strict typechecking i.e. value for each FormCOntrol must be provided in the setValue function according to the FormGroup model created.

To overcome this we can use patchValue function instead of setValue.

this.registration.patchValue({

      userName:"Shivam",

      passWord:"Shivam123"

    });

**Another way of using FormControl, bind the formControl directive with property in component class. It can be used if we don’t want to use FormGroup and individual elements.**

<input type="text" [formControl]="name">

export class NameEditorComponent {

name = new [FormControl](https://angular.io/api/forms/FormControl)('');

}

* **FormBuilder**

Creating form control instances manually can become repetitive when dealing with multiple forms. The [FormBuilder](https://angular.io/api/forms/FormBuilder) service provides convenient methods for generating controls.

Use the following steps to take advantage of this service.

1. Import the [FormBuilder](https://angular.io/api/forms/FormBuilder) class.

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

1. Inject the [FormBuilder](https://angular.io/api/forms/FormBuilder) service.

The [FormBuilder](https://angular.io/api/forms/FormBuilder) service is an injectable provider that is provided with the reactive forms module. Inject this dependency by adding it to the component constructor.

constructor(private fb: [FormBuilder](https://angular.io/api/forms/FormBuilder)) { }

1. Generate the form contents.

The [FormBuilder](https://angular.io/api/forms/FormBuilder) service has three methods: **control(), group(), and array()**. These are factory methods for generating instances in your component classes including form controls, form groups, and form arrays.

registration = this.fb.group({

  userName:[''],

  passWord:[''],

  confirmPassword:['']

})

* **Validations**

Validations can also be given in classes.

We first import Validators module from angular/forms.

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

You can specify the validation type after the value parenthesis.

registration = this.fb.group({

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

})

**Now for visual effects on HTML, it is similar to what we do in template driven forms.**

<input [class.is-invalid]="registration.get('userName').invalid && registration.get('userName').touched"

formControlName="userName" type="text" class="form-control">

Or you can create a getter for the field and use that in the HTML.

get userName(){

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

}

<input [class.is-invalid]="userName.invalid && userName.touched" formControlName="userName" type="text"

class="form-control">

* **Creating Dynamic Forms**

Import FormArray in class.

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

Create a FormArray in the FormGroup or individually.

registration = this.fb.group({

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

  passWord:[''],

  confirmPassword:[''],

**email : this.fb.array([])**

})

Create a getter for the formbuilder array and create a method for click event of button.

get email(){

  return this.registration.get('email') as FormArray;

}

addemail(){

  this.email.push(this.fb.control(''));

}

Create a button in HTML and a div for for looping the FormArray.

<button type="button" (click)="addemail()" class="btn btn-primary btn-sm m-2">Add Email</button>

<div formArrayName="email" \*ngFor="let mail of email.controls; let i = index">

<input type="text" class="form-control my-1" [formControlName]="i">

</div>

**Lifecycle sequence**

After creating a component/directive by calling its constructor, Angular calls the lifecycle hook methods in the following sequence at specific moments:

|  |  |
| --- | --- |
| **Hook** | **Purpose and Timing** |
| ngOnChanges() | Respond when Angular (re)sets data-bound input properties(@input). The method receives a SimpleChanges object of current and previous property values.Called before ngOnInit() and whenever one or more data-bound input properties change. It happens when some element is changed or value is changed. |
| ngOnInit() | Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties.Called once, after the first ngOnChanges(). Or component is initialized. It will run after constructor. |
| ngDoCheck() | Detect and act upon changes that Angular can't or won't detect on its own.Called during every change detection run, immediately after ngOnChanges() and ngOnInit(). |
| ngAfterContentInit() | Respond after Angular projects external content into the component's view.Called once after the first ngDoCheck(). A component-only hook. Called after content (ng-content) has been projected into view. |
| ngAfterContentChecked() | Respond after Angular checks the content projected into the component. Called after the ngAfterContentInit() and every subsequent  ngDoCheck(). A component-only hook. |
| ngAfterViewInit() | Respond after Angular initializes the component's views and child views.Called once after the first ngAfterContentChecked(). A component-only hook.  You can check get the values from DOM only after this lifecylec hook is done and not in any of the previous hooks.i.e. once the view is rendered you can get values from it. |
| ngAfterViewChecked() | Respond after Angular checks the component's views and child views. Called after the ngAfterViewInit and every subsequent ngAfterContentChecked(). A component-only hook. |
| ngOnDestroy | Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks. Called just before Angular destroys the directive/component. |

**How Angular App starts and works?**

In main.ts file, it’s the first code that gets executed.

The PlatformBrowseDynamic().bootstrapModule(AppModule); line starts the app by passing the AppModule.

The bootstrap Array lists the components which angular needs to know at the point of time it analyses the index.html. Here we refer the AppComponent in bootstrap array.

**What is Appmodule?**

Angular uses different components for views and appmodule to basically bundle different pieces of app(service,component) into packages.

The @NgModule decorator separetes it from other classes.

**Databinding**

String Interpolation {{data}} Output Data to HTML

Property Binding [property] = “data”

Event Binding (click)=”expression” 🡺 React to User Events

Two Way Binding [(ngModel)] = “data”

**Diff. in Attribute Directives and Structural Directives?**

|  |  |
| --- | --- |
| **Attribute Directives** | **Structural Directives** |
| Looks like normal HTML attribute(possibly with databinding or event binding) | Look like a normal HTML attribute but having a leading \*(for desugaring) |
| Only affect/change the element they are added to | Affect a whole area in the DOM(elements get added/removed) |

There can be only one structural directive on one element.

**Attribute Directives and Structural Directives?**

**Subject & BehaviourSubject**

Both are used to transfer data from one component to other. A Subject is **a special type of Observable that allows values to be multicasted to many Observers**. The subjects are also observers because they can subscribe to another observable and get value from it, which it will multicast to all of its subscribers. Basically, a subject can act as both observable & an observer.

The subject next method is **used to send messages to an observable which are then sent to all angular components that are subscribers (a.k.a. observers) of that observable**.

1) Create a service

import { Observable, Subject } from 'rxjs';

export class MessageService {

  private subject = new  Subject<string>();

  constructor() { }

  sendmessage(message:string){

    this.subject.next(message);

  }

  receivemessage(): Observable<string>{

    return this.subject.asObservable();

  }

}

2) In one component(Sender)

<div class="header">

    <input type="text" #food>

    <button (click)="sendmessage(food.value)">Send</button>

</div>

export class HeaderComponent implements OnInit {

  constructor(private msgservice: MessageService) { }

  ngOnInit(): void {

  }

  sendmessage(message:string){

    this.msgservice.sendmessage(message);

  }

}

2)Second Component(Receiver)

{{message}}

export class HomeComponent implements OnInit {

  message: string = "";

  constructor(private msgservice: MessageService) { }

  ngOnInit(): void {

    this.msgservice.receivemessage().subscribe((d) => {

      this.message = d;

    })

  }

}