About RI-TECH

Ri-Tech established in 2006, a leading IT Training company from Pune. We are established with goal in mind to develop and generate the future ready Software Developers which will lead the IT Industry. RI-TECH know it very well that today's university graduate or post graduate courses are not able to fulfill the industrial requirements, our .NET and Java Training modules are designed in such a way that we help graduates and undergraduates to upgrade or sharpen their skills to bridge the gap between university academics and industrial requirements. We are the only institute who not only dares but successfully executed latest technology trainings like ASP.NET MVC 5, Angular, Knock Out JS, Hibernate and Struts. Our training modules not only build student technical or programming skills but also it boosts the confidence of the students which helps them to face the interview or technical tests conducted by industries. We had trained more than 2000+ students from our establishment, 99.99% students are currently working with Top MNCS.

Our Training Model

Why RI-Tech?

RI-Tech established in 2006, a leading IT Training company from Pune. We are established with goal in mind to develop and generate the future ready software developers which will lead the IT Industry. We had trained more than 2000+ students from our establishment, 99.99% students are currently working with Top MNCS. We had conducted training courses for ASP.NET MVC, Angular JS and Knockout JS.

-10+ years of Experience in Trainings.
- Trained to more than 2000+ Students.
- More than 1000+ Students Working in MNC's.
- Free Upgrade's to Latest Technologies and Release.
- Free Technical and Interview Skills.
- State Of Art Infrastructure.
- Practical Implementation of Live Projects.
- Exposure to Industry Standards.
- 100% Placement Assistance.
- Study Material Designed by Experts having more than 10+ years of experience.
-Tie up's with more than 100+ companies for industrial trainings.
- Latest technology workshops and seminars.
-Application Development and Complete Experience of SDLC while implementing projects.

Type Script

TypeScript

TypeScript is a typed superset of JavaScript, which means that all JavaScript code is valid TypeScript code. TypeScript adds a lot of new features on top of that.

Limitations of JavaScript: -

Still JavaScript is very popular for web application development has few limitations –

- Java Script has dynamic types.
- No Support for static Types
- No good Intelligence is available for JavaScript.
- Since JavaScript is objet based we cannot use Java script for large applications. (It will make your application unmanageable and difficult to maintain if we use JavaScript)

TypeScript makes JavaScript more like a strongly-typed, object-oriented language akin to C# and Java. This means that TypeScript code tends to be easier to use for large projects and that code tends to be easier to understand and maintain.

TypeScript provides developers

- Type safety
- Compile time type checking
- Object oriented constructs
- Let developers think in object oriented terms when writing JavaScript.
- It compiles in JavaScript so we don't have to support any new VM to run TypeScript

Type Script Basics:-

- **Type Declarations –**
  You can add type declarations to variables, function parameters and function return types. The type is written after a colon following the variable name, like this:
  var num: number = 5;
  The compiler will then check the types (where possible) during compilation and report type errors

```
var isDone: boolean = false;
var decimal: number = 6;
var hex: number = 0xf00d;
var binary: number = 0b1010;
var  octal: number = 0o744;
var  color: string = "blue";
var color = 'red';
Any -   let notSure: any = 4; notSure = "maybe a string instead";
NULL & Undefined
let u: undefined = undefined; let n: null = null
```

- **Arrays and Tuple**

```
Var arrayname:datatype [] [=initial value]
var nums: number [] = [1, 2, 3, 4, 5];
Built in methods to push, pop array elements.
var pair: [string, number];
```

```
pair = ["Hello", 23];
alert(pair[0]);
alert(pair[1])
```

- **Enums**

```
enum Cities { Pune, Mumbai=10, Delhi };
alert(Cities.Pune);
alert(Cities.Mumbai);
alert(Cities.Delhi);
var c: Cities = Cities.Delhi;
alert(c);
```

- **Var, Const and Let Keyword**

```
var – function / global scope
function f(condition, x) { if (condition) { let x = 100; return x; } return x; } f(false, 0); //
returns '0' f(true, 0); // returns '100'
let – local scope

Constant – no reassigment
const numLivesForCat = 9; const kitty = { name: "Aurora", numLives: numLivesForCat, } //
Error kitty = { name: "Danielle", numLives: numLivesForCat };
```

- **Functions: -** In type script we can define anonymous functions or named functions just like JavaScript additionally it will have the parameters and return type.

```
function functioname([parameters])
{ function defination}

Parameter with Types:-
function addnum(a: number, b: number)
{alert("Addition is:" + (a + b));}

Return Value:
function warnUser(): void {
alert("This is my warning message");
 return 1;}
```

- **Rest Parameter: -** reset parameters enables you to define any number any type argument functions.

```
function values(...vals: number[])     {
Var  i: number = 0;
for (i = 0; i < vals.length; i++)
{
 alert(vals[i]);
  }
}
values(1,2,3,4,5,6,7);
```

- **Optional Parameters**

```
function DVal(a: number,b?:number){
    alert("val:" + a);
    alert("val:" + b);
}
DVal(10);
```

- **Anonymous and Arrow Functions:-**

```
 var variablename=([parametrs])=>{defination;};
var add = (a: number, b: number) => a + b;
var result = add(10, 20);
alert(result);

var add = (a: number) => {
 if (a % 2 == 0)
alert("Even Number");
else
alert("Odd Number");
};
add(13);
```

- **Interfaces and Type shaping:-**

One of TypeScript's core principles is that type-checking focuses on the shape that values have. This is sometimes called "duck typing" or "structural subtyping". In TypeScript, interfaces fill the role of naming these types

Not Shaped

```
function PrintLabel(llbl: {label:string}){
    console.log(llbl.label);
}
let obj = { Size: 10, label: "Hello" };
PrintLabel(obj);
```

Shaped

```
interface Student {
    Name: string,
    RollNo?: number,
    Address?:string
}
function ShowStudent(obj: Student){
    console.log(obj.RollNo);
    console.log(obj.Name);
    console.log(obj.Address);
}
let s: Student = { RollNo: 121};
ShowStudent(s);
```

- **Classes in TypeScript:-**
  Typescript support classes methods, properties and constructors with following access types -
  -Public – default
  -Private
  -Protected

```
We can create class in typescript as follows –
class Greeter {
    greeting: string;
    public show(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
let greeter = new Greeter();
greeter.show("Hello");
```

- **Constructors in Type Script:-**

```
class Emp{
    constructor(private eid: number,private ename: string)  {
    }
    public ShowEmp() {
        console.log(this.eid);
        console.log(this.ename);
    }}
let e = new Emp(121,"Ramesh");
e.ShowEmp();
```

- **Shaped**

```
interface Student {
    Name: string,
    RollNo?: number,
    Address?:string
}
function ShowStudent(obj: Student) {
    console.log(obj.RollNo);
    console.log(obj.Name);
    console.log(obj.Address);
}
let s: Student = { RollNo: 121};
ShowStudent(s);
```

- **Classes in TypeScript:-**

We can create class in typescript as follows –

```
class Greeter {
```

```
    greeting: string;
    public show(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
let greeter = new Greeter();
greeter.show("Hello");
```

- **Constructors in Type Script:-**

```
class Emp{
    constructor(private eid: number,private ename: string)    {
    }
    public ShowEmp() {
        console.log(this.eid);
        console.log(this.ename);
    }
}
let e = new Emp(121,"Ramesh");
e.ShowEmp();
```

Inheritance In TypeScript:-

This example shows the most basic inheritance feature: classes inherit properties and methods from base classes. Here, Dog is a derived class that derives from the Animal base class using the extends keyword. Derived classes are often called subclasses, and base classes are often called super classes.

Because Dog extends the functionality from Animal, we were able to create an instance of Dog that could both bark() and move()

```
class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 5) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}
class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 45) {
        console.log("Galloping...");
        super.move(distanceInMeters);
    }
}
let sam = new Snake("Sammy the Python");
```

```
let tom: Animal = new Horse("Tommy the Palomino");
sam.move();
tom.move(34);
```

- **Static Keyword in Type Script**

```
class CountCls         {
 private static count: number = 0;s
  public IncCount()    {
      CountCls.count++;
      console.log(CountCls.count);
   }
}
let c=new  CountCls();
c.IncCount();
let c1 = new CountCls();
c1.IncCount();
```

- **Abstract Keyword:-**

You can define class and method using abstract keyword with class and methods –

```
abstract class Animal {
   abstract makeSound(): void;
   move(): void {
      console.log("roaming the earth...");
   }
}
class Dog extends Animal{
   makeSound(): void {
      alert("Dog Abstract Class");
   }
}
let d = new Dog();
d.makeSound();
```

Modules in Typescript

When tsc compiles typescript into JavaScript, you end up with a bunch of js files on your local system. They somehow need to be loaded into a browser. You have two options, either put them all into your index.html file in the correct order of dependencies, or you can use a loader to do that all for you. You specify the root for all modules, and then all files are loaded and executed by that loader in the correct order of dependencies. There are many loaders: requirejs, webpack, systemjs and others.

Over the years there's been a steadily increasing ecosystem of JavaScript components to choose from. The sheer amount of choices is fantastic, but this also infamously presents a

difficulty when components are mixed-and-matched. And it doesn't take too long for budding developers to find out that not all components are built to play nicely together.

To address these issues, the competing module specs AMD and CommonJS have appeared on the scene

Asynchronous Module Definition (AMD) has gained traction on the frontend, with RequireJS being the most popular implementation.
CommonJS is a style you may be familiar with if you're written anything in Node (which uses a slight variant). It's also been gaining traction on the frontend with Browserify.

UMD: Universal Module Defination
Since CommonJS and AMD styles have both been equally popular, it seems there's yet no consensus. This has brought about the push for a "universal" pattern that supports both styles, which brings us to none other than the Universal Module Definition.

Modules are executed within their own scope, not in the global scope; this means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the export forms. Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the import forms.

Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level.

- **Abstract Keyword:-**
  You can define class and method using abstract keyword with class and methods –

```
abstract class Animal {
   abstract makeSound(): void;
   move(): void {
      console.log("roaming the earth...");
   }
}
class Dog extends Animal{
   makeSound(): void {
      alert("Dog Abstract Class");
   }
}
let d = new Dog();
d.makeSound();
```

- **Modules in Typescript**
  When tsc compiles typescript into JavaScript, you end up with a bunch of js files on your local system. They somehow need to be loaded into a browser. You have two options, either put them all into your index.html file in the correct order of dependencies, or you can use a loader to do that all for you. You specify the root for all modules, and then all files are loaded

and executed by that loader in the correct order of dependencies. There are many loaders: requirejs, webpack, systemjs and others.

Over the years there's been a steadily increasing ecosystem of JavaScript components to choose from. The sheer amount of choices is fantastic, but this also infamously presents a difficulty when components are mixed-and-matched. And it doesn't take too long for budding developers to find out that not all components are built to play nicely together.To address these issues, the competing module specs AMD and CommonJS have appeared on the scene **Asynchronous Module Definition (AMD)** has gained traction on the frontend, with RequireJS being the most popular implementation.
CommonJS is a style you may be familiar with if you're written anything in Node (which uses a slight variant). It's also been gaining traction on the frontend with Browserify.

- **UMD: Universal Module Defination**

Since CommonJS and AMD styles have both been equally popular, it seems there's yet no consensus. This has brought about the push for a "universal" pattern that supports both styles, which brings us to none other than the Universal Module Definition.

```
Validation.ts
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
ZipCodeValidator.ts
export const numberRegexp = /^[0-9]+$/;
export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

Namespaces in TypeScript

Namespaces are simply named JavaScript objects in the global namespace. This makes namespaces a very simple construct to use. They can span multiple files, and can be concatenated using --outFile. Namespaces can be a good way to structure your code in a Web Application, with all dependencies included as <script> tags in your HTML page.

```
namespace Validation { export interface StringValidator {
isAcceptable(s: string): boolean;
}
}
Reference Namespaces –
/// <reference path="Validation.ts" />
Include it in actual page –
<script src="Validation.js" type="text/javascript" />
```

Angular 4/5

Angular (What & Why)

Angular is a platform that makes it easy to build applications with the web. Angular combines declarative templates, dependency injection, end to end tooling, and integrated best practices to solve development challenges. Angular empowers developers to build applications that live on the web, mobile, or the desktop

What?

- Angular is JavaScript Framework which help you develop and build client side applications.
- It Uses CSS, HTML and JavaScript to Build Applications.

Why?

- Expressive HTML
- Powerful Data Binding
- Modular By Design
- Built In Backend Integration.
- Speed.
- Modern (use lot of API's, CLI and Type Script).
- Simplified API.
- Enhances Productivity.
- Component-Based.-Angular 2 is entirely component based.
- Angular 2 was written in TypeScript, a superset of JavaScript that implements many new ES2016+ features.

Difference between Angular and Angular JS

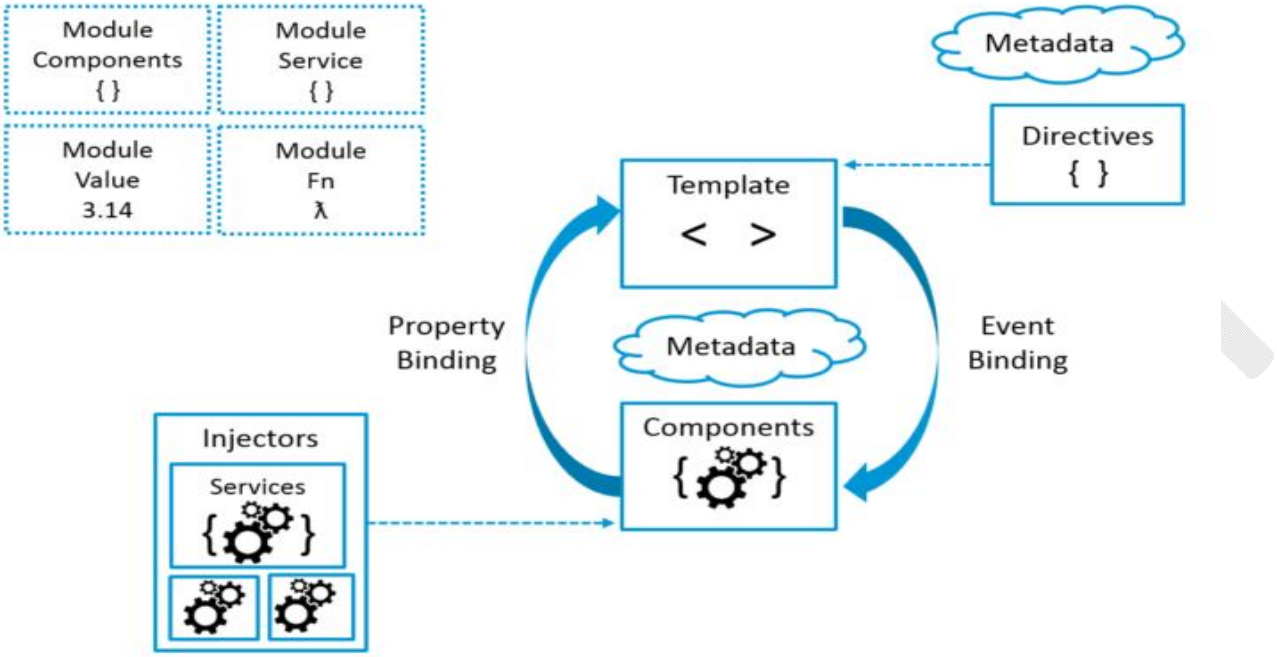
- The architecture of an Angular application is different from AngularJS. The main building blocks for Angular are modules, components, templates, metadata, data binding, directives, services and dependency injection. We will be looking at it in a while.
- Angular was a complete rewrite of Angular JS, entirely developed using typescript.
- Angular does not have a concept of "scope" or controllers instead; it uses a hierarchy of components as its main architectural concept.
- Angular has a simpler expression syntax, focusing on "[ ]" for property binding, and "( )" for event binding
- Mobile development – Desktop development is much easier when mobile performance issues are handled first. Thus, Angular first handles mobile development.
- Modularity – Angular follows modularity. Similar functionalities are kept together in same modules. This gives Angular a lighter & faster core.

Angular Application Architecture:-

Angular application is modular and consists of following things –

- Modules
- Components
- Templates
- Metadata

- Data binding
- Directives
- Services
- Dependency injection



- **Modules:-**
  Angular apps are modular and to maintain modularity, we have Angular modules or you can say NgModules. Every Angular app contains at least one Angular module, i.e. the root module. Generally, it is named as AppModule. The root module can be the only module in a small application. While most of the apps have multiple modules. You can say, a module is a cohesive block of code with a related set of capabilities which have a specific application domain or a workflow. Any angular module is a class with @NgModule decorator.

- **Components:-**
  A component controls one or more section on the screen called a view. For example, if you are building a movie list application, you can have components like App Component (the bootstrapped component), Movielist Component, Movie Description Component, etc.

- **Templates:-**
  You associate component's view with its companion template. A template is nothing but a form of HTML tags that tells Angular about how to render the component. A template looks like regular HTML, except for a few differences.

- **Metadata:-**
  Metadata tells Angular how to process a class. To tell Angular that Movie List Component is a component, metadata is attached to the class.

- **Data binding:-**
  If you are not using a framework, you have to push data values into the HTML controls and turn user responses into some actions and value updates.

- **Directives:-**
  Angular templates are dynamic. When Angular renders them, it transforms the DOM according to the instructions given by directives.

- **Services:-**
  Service is a broad category encompassing any value, function, or feature that your application needs. A service is typically a class with a well-defined purpose. Anything can be a service.

- **Dependency injection:-**
  Dependency injection is a way to supply a new instance of a class with the fully-formed dependencies it requires.

Install and Run Angular Application:-

Before we can begin, you need to ensure you have a few dependencies installed.
- Node.js with NPM (Node Package Manager)
- Angular-CLI (Command Line Interface)

To check whether or not you have Node.js installed, visit your console / command line and type:

```
node -v
```

It will show you the node js version installed on your coputer for angular 2 we need node.js version > 6.0

If this command goes unrecognized, you need to install Node.js.

- Visit the Node.js download page and choose either the Windows or Mac installer based on your OS.
- Accept the agreement, choose an installation folder, and hit Next on the Custom Setup page.
- By default, it will install the npm package manager which we will need.
  After it's installed, close your console / command line and reload it. You can now run the node -v command and it will provide you with the current version number.
- Next, we need to install the Angular-CLI. This tool allows you to create Angular projects as well as help you with various development tasks.
  At the console, type:

```
npm install –g @angular/cli
```

Once finished, type:

```
ng -v
```

- It will show you following screen depending on your machine configuration and version available –

```
     _                      _                 ____ _     ___
    / \   _ __   __ _ _   _| | __ _ _ __     / ___| |   |_ _|
   / △ \ | '_ \ / _` | | | | |/ _` | '__|   | |   | |    | |
  / ___ \| | | | (_| | |_| | | (_| | |      | |___| |___ | |
 /_/   \_\_| |_|\__, |\__,_|_|\__,_|_|       \____|_____|___|
                |___/
@angular/cli: 1.0.0
node: 6.10.0
os: win32 x64
```

Now we are ready for angular

- Create a new Project using Angular CLI and run it.

```
ng new my-project-name
// let it install

cd my-project-name

ng serve --open
```

- The ng serve command launches the server, watches your files, and rebuilds the app as you make changes to those files.
- Using the --open (or just -o) option will automatically open your browser on http://localhost:4200/.

Angular Modules in Detail

Angular apps are modular and Angular has its own modularity system called NgModules. An NgModule is a container for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities. It can contain components, service providers, and other code files whose scope is defined by the containing NgModule. It can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.

Every Angular app has at least one NgModule class, the root module, which is conventionally named AppModule and resides in a file named app.module.ts. You launch your app by bootstrapping the root NgModule.

While a small application might have only one NgModule, most apps have many more feature modules. The root NgModule for an app is so named because it can include child NgModules in a hierarchy of any depth.

NgModule metadata

An NgModule is defined as a class decorated with @NgModule. The @NgModule decorator is a function that takes a single metadata object, whose properties describe the module. The most important properties are as follows.

Declarations: - The components, directives, and pipes that belong to this NgModule.
Exports: - He subset of declarations that should be visible and usable in the component templates of other NgModules.

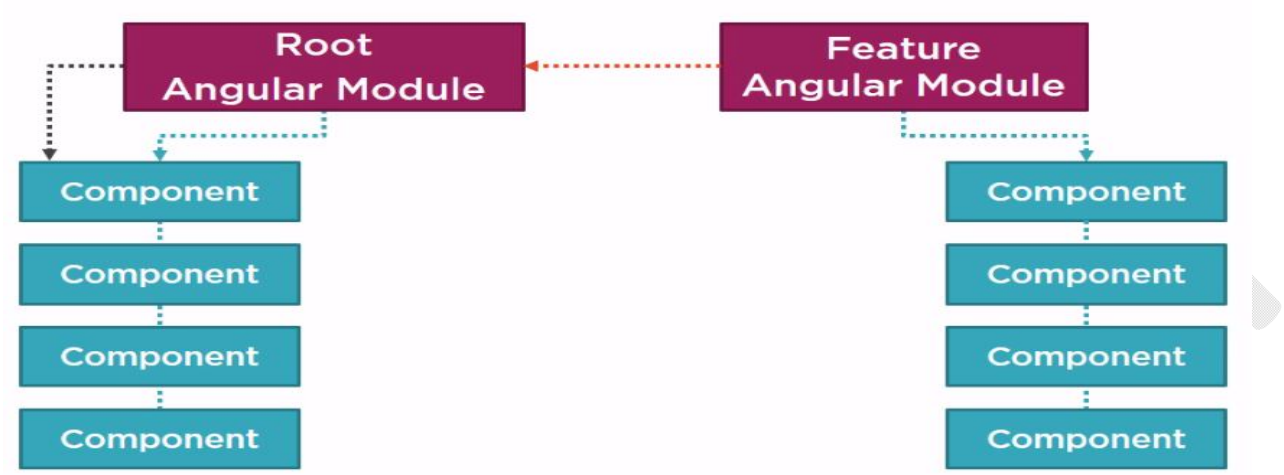
Imports: - There modules whose exported classes are needed by component templates declared in this NgModule.

Providers: - Creators of services that this NgModule contributes to the global collection of services they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)

Bootstrap: - The main application view, called the root component, which hosts all other app views. Only the root NgModule should set this bootstrap property.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
 declarations: [ AppComponent ],
```

```
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

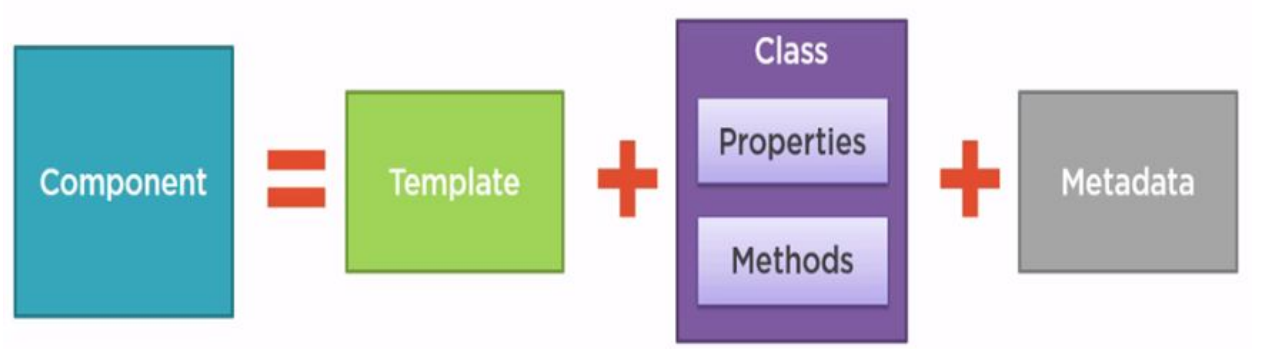


Angular Component in Detail

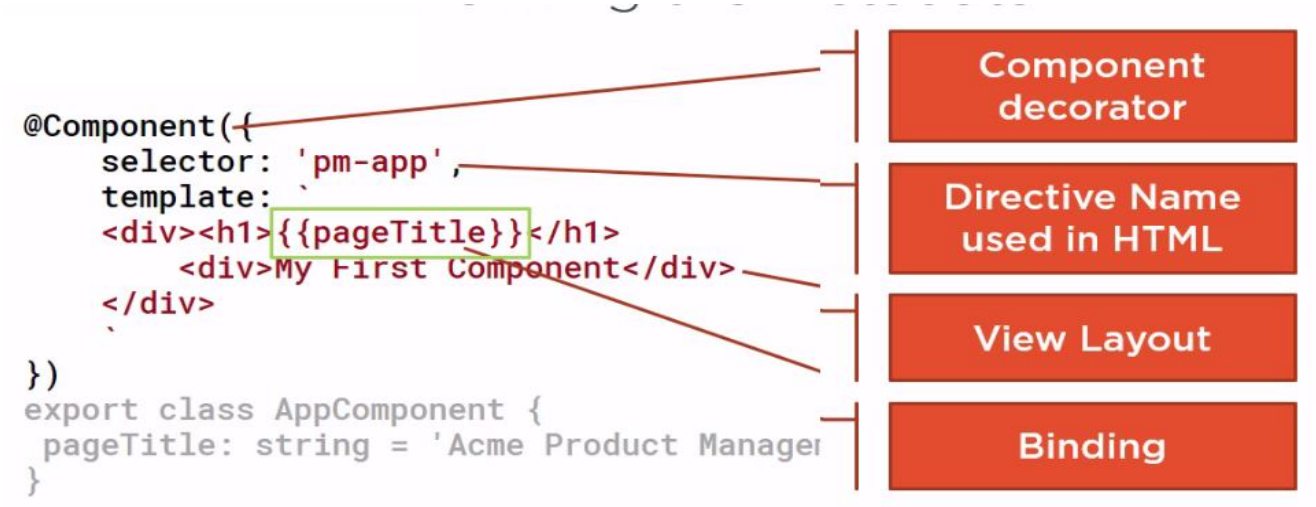
A component controls a patch of screen called a view. The @Component decorator identifies the class immediately below it as a component class, and specifies its metadata. In the example code below, you can see that ProductListComponent is just a class, with no special Angular notation or syntax at all. It's not a component until you mark it as one with the @Component decorator.

The metadata for a component tells Angular where to get the major building blocks it needs to create and present the component and its view. In particular, it associates a template with the component, either directly with inline code, or by reference. Together, the component and its template describe a view.

In addition to containing or pointing to the template, the @Component metadata configures, for example, how the component can be referenced in HTML and what services it requires.



Component is nothing but a class with decorator @Component can be defined as follows –



This example shows some of the most useful @Component configuration options:

Selector: A CSS selector that tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML. For example, if an app's HTML contains <pm-app></pm-app>, then Angular inserts an instance of the Component view between those tags.

Template and Template Url: The module-relative address of this component's HTML template. Alternatively, you can provide the HTML template inline, as the value of the template property. This template defines the component's host view.

■ Define External Templates

When you use the Angular CLI to generate an Angular project, it uses the templateUrl metadata property to define an external template by default.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
}
```

■ Defining Inline HTML Templates

Using the code above, let's change the templateUrl property to template, and declare a single line of HTML.

```
template: '<h1>Hey guys!</h1>',
```

If you want the ability to define multiple lines of inline HTML, single quotes will not work. Instead, you will need to change them to backticks ``.

```
template: `
  <h1>Hey guys!</h1>
  <p>How are you dong?</p>
  `,
```

Providers: An array of dependency injection providers for services that the component requires. In the example, this tells Angular that the component's constructor requires a Service instance in order to get the list of records to display.

Styles and Style Files in component metadata:
You can add a styles array property to the @Component decorator. Each string in the array defines some CSS for this component**.**

```
@Component({
  selector: 'app-root',
  template: `
    <h1>Sample</h1>
  `,
  styles: ['h1 { font-weight: normal; }']
})
export class AppComponent {
/* . . . */
}
```

You can load styles from external CSS files by adding a styleUrls property to a component's @Component decorator:

```
@Component({
  selector: 'app-root',
  template: `
    <h1>Sample Application</h1>
  `,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
/* . . . */
}
```

Angular Data Binding in Detail:-

Data binding enables you to bind the values from component class to ui elements inside your template. We have following options for data binding in angular –

1. Interpolation
2. Property Binding

3. Event Binding
4. Two way binding.

Interpolation

Whenever you need to communicate properties (variables, objects, arrays, etc..) from the component class to the template, you can use interpolation.

The format for defining interpolation in a template is: **{{ propertyName }}**
Define a property in component class as possible –

```
export class HomeComponent implements OnInit {
  itemCount: number = 4;
  constructor() { }
  ngOnInit() {
  }
}
```

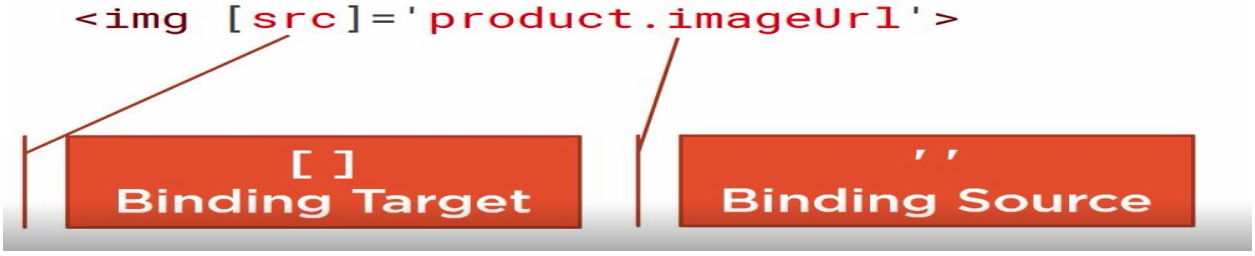
Then, in /home.component.html update the following:

```
<!-- From: -->
<p>Your bucket list</p>
<!-- To: -->
<p>Your bucket list ({{ itemCount }})</p>
```

If you save the project, you'll notice in the browser that we've used interpolation {{ }} to show the itemCount property in the browser.

Property Binding

Property binding enables you to bind the values or properties from component class to properties of html tags like src, title, size etc. It is also called one way binding , since when you change the value of the property from component class it will reflect those changes to html tag from template.




```
export class HomeComponent implements OnInit {
  itemCount: number = 4;
  constructor() { }
  ngOnInit() {
  }
}
```

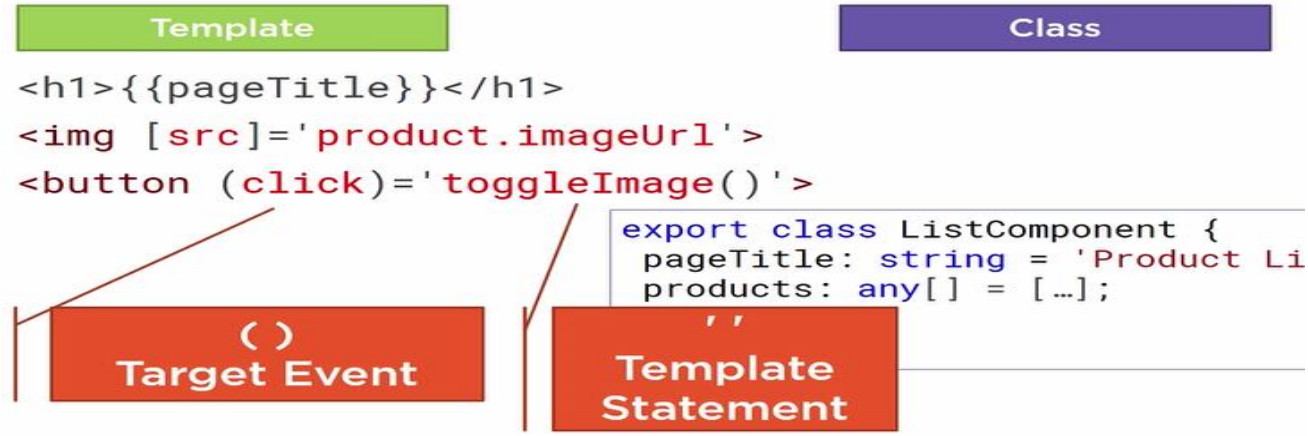
Then, in /home.component.html update the following:

```
<!-- From: -->
<p>Your bucket list</p>

<!-- To: -->
<p>Count is : <input type='text' [value]='itemCount' />  <p>
```

Event Binding

Event Binding enables us to bind the methods or functions from the component class to the vent generated by Html controls from template (view) so that when that event get fired it will call the function/method from the component class.




```
export class HomeComponent implements OnInit {
  showMessage():void
  {
    alert('Show Message Function Called!');
  }
}
```

Then, in /home.component.html update the following:

```
<!-- From: -->
<p>Your bucket list</p>
 <input type="button" (click)="showMessage()" />
```

One Way and Two way Binding using NgModel

1-Way

In oneway model binding when you change the value of model property from component class will be reflected to the UI element from template class, we can do it using NgModel directive as follows.

To use NgModel directive you need to use formsModule, to use it you need to add the FormsModule in imports selection of your root module called app modules.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import {AppComponent} from './App';
@NgModule({
  declarations: [
    AppComponent,

  ],
  imports: [
    BrowserModule,
    FormsModule,

  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

/home.component.ts and add properties:

```
  itemCount: number = 4;
  btnText: string = 'Add an Item';
  goalText: string = 'My first life goal';
```

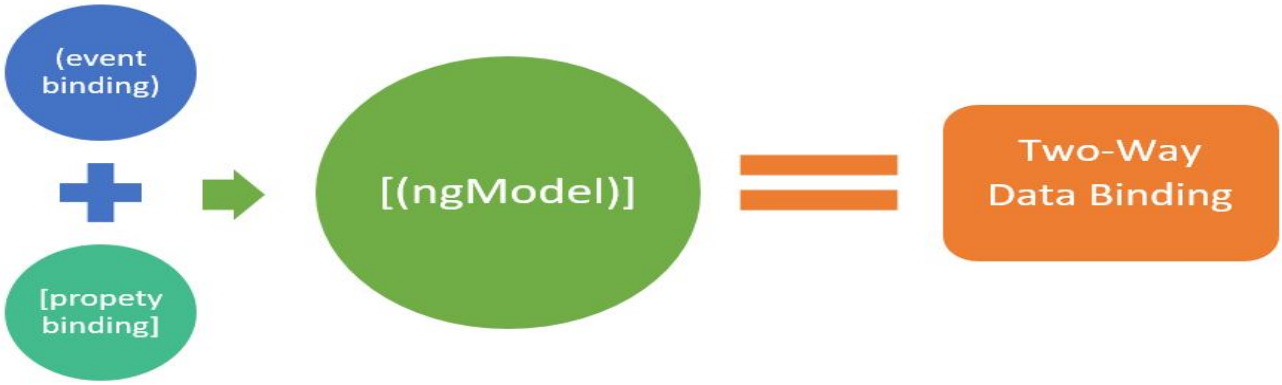
In the template file, add this to create a 1-way data binding:

```
<!-- From: -->
<input type="text" class="txt" name="item" placeholder="Life goal..">
<!-- To: -->
<input type="text" class="txt" name="item" placeholder="Life goal.."
[ngModel]="goalText">
<br><span>{{ goalText }}</span><br>
```

2-Way

Two way biding enabls you track value changes from Component class to UI element from Template. If any changes happened in class property will be automatically reflected to UI element from template and if any change in UI element from template will be reflected to property from class. To implement two way binding we need to use NgModel directive from FormsModule.

They are commonly referred to as banana in a box because the parentheses in the brackets appear like a banana inside a box. Two-way bindings listen for particular events such as change or click and update the data model with new data when a change is detected.



/home.component.ts and add properties:

```
itemCount: number = 4;
btnText: string = 'Add an Item';
goalText: string = 'My first life goal';
```

In the template file, add this to create a 2-way data binding:

```
<!-- From: -->
<input type="text" class="txt" name="item" placeholder="Life goal..">
<!-- To: -->
<input type="text" class="txt" name="item" placeholder="Life goal.."
[(ngModel)]="goalText">
<br><span>{{ goalText }}</span><br>
```

Angular Directives in Details

Directives are the most fundamental unit of Angular applications. As a matter of fact, the most used unit, which is a component, is actually a directive. Components are high-order directives with templates and serve as building blocks of Angular applications.Angular view is made up of html and directive. Angular has following three directives -

1. Component with Template is Directive.
2. Attribute Directives
3. Property Directives

Component is Directive: -

Component with template is also a directive where we define selector and use that selector inside html view or UI.
We has already implemented component.

Attribute Directives:-

Attribute Directive is basically used to modify or alter the appearance and behavior of an element.The selector is the property that identifies the attribute. It is used as HTML tag to

target & insert an instance of the directive class where it finds that tag. The directive class implements the desired directive behavior.

Some Built in Directives:-

hidden :- hidden directive will help you to hide element if property value is true and vice versa.

```
<div [hidden]="!ename.errors.minlength">
        Min Length Error</div>
 </div>
```

Disabled: - disable directive will help you to disable the element if property value is true and vice versa.

```
<<input type="submit" value="Save" [disabled]="!frm.form.valid" />
```

ngModel:- this directive will help you to implement one way or two way binding with html control to property from component class.

```
<input type="text" class="txt" name="item" placeholder="Life goal.."
[(ngModel)]="goalText">
```

ngStyle:- this directive will help you apply css tyle to any html emement from template.

```
 [ngStyle]="{backgroundColor: temp.CustomerID % 2==0 ? 'red':'green'}
```

ngClass:- this directive will help you to apply css class to any html element from template.

```
 [ngClass]="{heighlite:temp.CustomerID%2==0}“
```

Structural Directives:-

Structural directives are responsible for shape or reshape the DOM's structure, typically by adding, removing, or manipulating elements. Similar to other directives, you apply a structural directive to a host element. The directive then does whatever it's designed to do with that host element. Structural directives are easy to recognize. An asterisk (*) precedes the directive attribute name. It does not require brackets or parentheses like attribute directive.

Few Built In Directives –
*ngIf
*ngFor
*ngSwitch
*ngIf-else

***ngIf:-**

NgIf is the simplest structural directive and the easiest to understand. It takes a boolean expression and makes an entire chunk of the DOM appear or disappear. You can assume it similar as if statement in a programming language.

The ngIf directive doesn't hide elements. It adds and removes them physically from the DOM. You can confirm it by using browser developer tools to inspect the DOM.

```
<div *ngIf="movie">{{movie.name}}</div>
```

***ngFor:**
ngFor directive help you to repeatedly add html elements in dom. It needs a variable to travel the collection and a list or array to traverse. E.g. ngFor needs a looping variable (let movie) and a list (movies).ngFor will help you to iterate through the list or array and repeat some html tags.

```
<div template="ngFor let movie of movies">{{movie.name}}</div>
```

***ngSwitch:-**
The Angular NgSwitch is actually a set of cooperating directives: NgSwitch, NgSwitchCase, and NgSwitchDefault.The switch value assigned to NgSwitch (movie.genre) determines which of the switch cases are displayed.

NgSwitchCase and NgSwitchDefault are structural directives. You attach them to elements using the asterisk (*) prefix notation. A NgSwitchCase displays its host element when its value matches the switch value. The NgSwitchDefault displays its host element when no sibling NgSwitchCase matches the switch value.

```
<div [ngSwitch]="movie?.genre">
  <action-movie *ngSwitchCase="'action'" [movie]=" movie "></action-movie>
  <horror-movie *ngSwitchCase="'horror'" [movie]=" movie "></horror-movie>
  <thriller-movie *ngSwitchCase="'thriller'" [movie]=" movie "></thriller-movie>
  <unknown-movive *ngSwitchDefault [movie]=" movie "></unknown-movie>
</div>
```

***ngIf-else:-**
This directive is similar to ngIf only difference that it will have elese template that will be added to DOM when condition specified in ngIf is false.

```
<div *ngIf="title; else login">
   The user is logged in.
</div>
<ng-template #login>Please login to continue.</ng-template>
```

Custom Directive:-
Angular also provides you opportunity to create a custom directive that can be implemented as follows –

For writing a custom structural directive:

Import the Directive decorator (instead of the Component decorator).
Import the Input, TemplateRef, and ViewContainerRef symbols; you'll need them for any structural directive.

Apply the decorator to the directive class.

Set the CSSattribute selector that identifies the directive when applied to an element in a template.
This is how you start creating a directive:

```
import{Directive, Input, TemplateRef, ViewContainerRef} from '@angular/core';
@Directive({
selector: '[myCustom]'
})
export class myCustomDirective {}
```

The directive's selector is typically the directive's attribute name in square brackets, [myCustom]. The directive attribute name should be spelled in lowerCamelCase and begin with a prefix. Don't use ng. That prefix belongs to Angular. The directive class name ends in Directive

Angular Life Cycle Hooks

A component has a lifecycle managed by Angular.
Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.

Angular offers lifecycle hooks that provide visibility into these key life moments and the ability to act when they occur. A directive has the same set of lifecycle hooks.

Directive and component instances have a lifecycle as Angular creates, updates, and destroys them. Developers can tap into key moments in that lifecycle by implementing one or more of the lifecycle hook interfaces in the Angular core library.

Each interface has a single hook method whose name is the interface name prefixed with ng. For example, the OnInit interface has a hook method named ngOnInit() that Angular calls shortly after creating the component.
Example of implementing on init hook –

```
export class PeekABoo implements OnInit {
```

```
  constructor(private logger: LoggerService) { }
  // implement OnInit's `ngOnInit` method
  ngOnInit() { this.logIt(`OnInit`); }
  logIt(msg: string) {
    this.logger.log(`#${nextId++} ${msg}`);
  }
}
```

List of Life Cycle Hooks:-

ngOnChanges()
Respond when Angular (re)sets data-bound input properties. 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.

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().

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 / the view that a directive is in.Called once after the first ngDoCheck().

ngAfterContentChecked()
Respond after Angular checks the content projected into the directive/component.Called after the ngAfterContentInit() and every subsequent ngDoCheck().

ngAfterViewInit()
Respond after Angular initializes the component's views and child views / the view that a directive is in.Called once after the first ngAfterContentChecked().

ngAfterViewChecked()
Respond after Angular checks the component's views and child views / the view that a directive is in.Called after the ngAfterViewInit and every subsequent ngAfterContentChecked().

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.

Pipes In Angular

A pipe takes in data as input and transforms it to a desired output. In this page, you'll use pipes to transform a component's birthday property into a human-friendly date
Pipes are used to transform data, when we only need that data transformed in a template.
If we need the data transformed generally we would implement it in our model, for example we have a number 1234.56 and want to display it as a currency such as $1,234.56.
You can use the pipes for –

- You can display only some filtered elements from an array.
- You can modify or format the value.
- You can use them as a function.
- You can do all of the above combined.

Example of Angular Pipe:-

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-emp-birthday',
  template: `<p>The emp's birthday is {{ birthday | date }}</p>`
})
export class EmpBirthdayComponent {
  birthday = new Date(1988, 3, 15); // April 15, 1988
}
```

Angular 4 provides some built-in pipes. The pipes are listed below –

- Lowercasepipe
- Uppercasepipe
- Datepipe
- Currencypipe
- Percentpipe
- Decimalpipe

Cusotm Pipes: -

Angular Provides use pipe directivewhich enables us to develop our own custom pie to execute some logic.

Pipe decorator

One pipe I really find useful when building web applications is a default pipe, which I use for things like avatar images.

I use this pipe in an image field, like so:

```
<img [src]="imageUrl | default:'<default-image-url>'"/>
```

The pipe is called default and we pass to it a default image to use if the imageUrl variable is blank.

To create a pipe we use the @Pipe decorator and annotate a class like so:

```
import { Pipe } from '@angular/core';
.
@Pipe({
  name:"default"
})
class DefaultPipe { }
```

The name parameter for the Pipe decorator is how the pipe will be called in templates.

Transform function

The actual logic for the pipe is put in a function called transform on the class, like so:

```
class DefaultPipe {
  transform(value: string, fallback: string): string {
    let image = "";
    if (value) {
      image = value;
    } else {
      image = fallback;
    }
     return image;
  }
}
```

The first argument to the transform function is the value that is passed into the pipe, i.e. the thing that goes before the | in the expression.

The second parameter to the transform function is the first param we pass into our pipe, i.e. the thing that goes after the: in the expression.

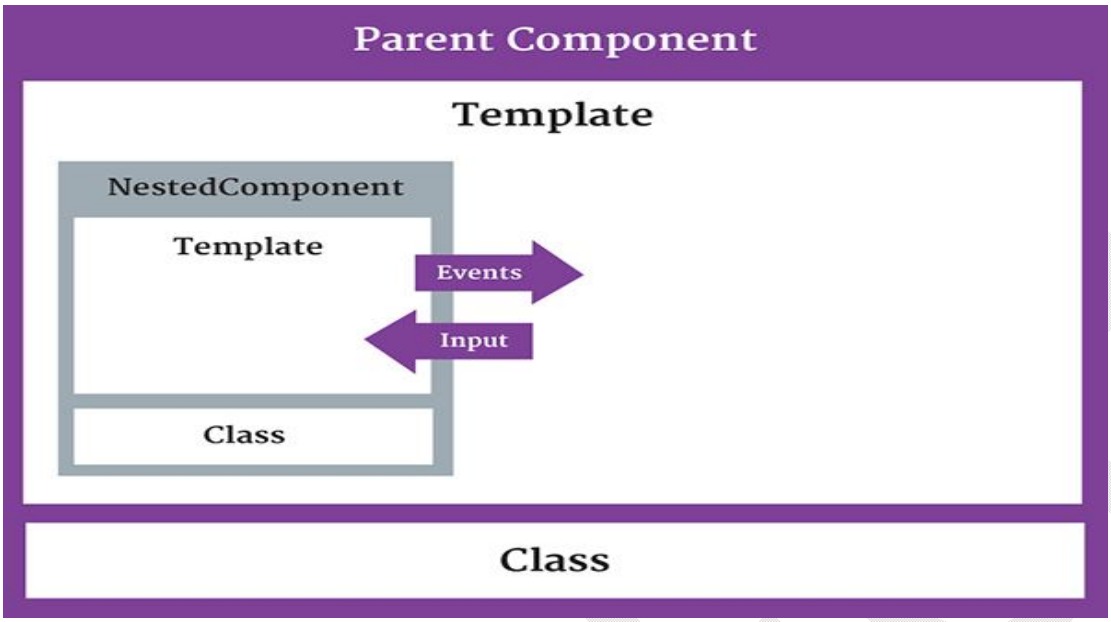
Specifically with this example:-

```
@Component({
  selector: 'app',
  template: `
  <img [src]="imageUrl |
default:'http://s3.amazonaws.com/uifaces/faces/twitter/sillyleo/128.jpg'"/>
 `
})
class AppComponent {
  imageUrl: string = "";
}
```

Value gets passed imageUrl which is blank.
Fallback gets passed 'http://s3.amazonaws.com/uifaces/faces/twitter/sillyleo/128.jpg'

Nested Components In Angular

Since angular is component based we need to nest the components in side another components to build complex UI's.



Let's first create a basic component that will be nested in another component later on. This component has a title property that we use in its template:

```
@Component({
  selector: 'child-selector',
  template: 'child.component.html'
})
export class ChildComponent {
  title = 'I\'m a nested component';
}
```

The child.component.html is just an HTML file that shows the value of the title property:

```
/* child.conponent.html */
<h2>{{title}}</h2>
```

Now we want to create a container component. It looks almost identical to the nested component, except we have to specify that we want to use the nested component. We do that by adding the ChildComponent to the directives property of the Component decorator. Without doing this, the ChildComponent can not be used.

```
@Component({
  selector: 'parent-selector',
  template: 'parent.component.html',
  directives: [ChildComponent]
})
export class ParentComponent { }
```

The container component uses the nested component by specifying its directive in the template:

```
/* parent.component.html */
<div>
  <h1>I'm a container component</h1>
  <child-selector></child-selector>
</div>
```

Passing data to a nested component

If a nested component wants to receive input from its container, it must expose a property to that container. The nested component exposes a property it can use to receive input from its container using the @Input decorator.

We use the Input decorator to decorate any property in the nested component class. This works with every property type, including objects. In our example, we'll pass a string value to the nested component's title property, so we'll mark that property with the @Input decorator:

```
@Component({
  selector: 'child-selector',
  template: 'child.component.html'
})
export class ChildComponent {
  @Input() title:string;
}
```

Now our nested component is ready to receive input from its parent component.

In the container component, we need to define the property we want to pass to the nested component. We call it childTitle:

```
@Component({
  selector: 'parent-selector',
  template: 'parent.component.html',
  directives: [ChildComponent]
})
export class ParentComponent {
  childTitle:string = 'This text is passed to child';
}
```

Services In Angular

In angular service is nothing but the class which focused propose which contains a business logic or cose that can be reused across multiple components or used to do some specific task.

Services can be used for –

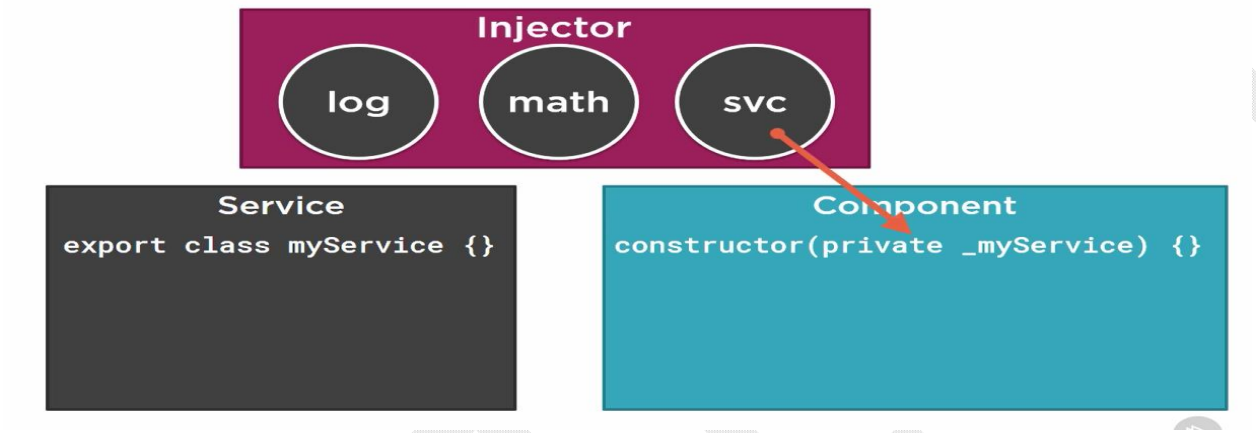
- Defining some logic independent from any specific component.
- Share data or logic across multiple components.
- Make external http calls to retrieve or update data in database.

  Etc.

Dependency Injections:-

Dependency Injection is coding pattern in which a class receives the instance of object it needs (called dependencies) from an external source rather than creating them itself. Angular has built in dependency injections and injector. Angular dependency injector is hierchical so you should be care full while injecting services. If you inject service in parent component then it's instance will be also available to child components and will be shared between all the child components.

Angular provides you built in services as well as you can also create your custom service by defining the class decorated using @Injectable() attribute which can be injected in any component.



To inject any serverice in any component we need to tell it to injector by registering that service. We can register the service to component or module using providers attribute or parameters. Secondly we need to define dependency on service in component by creating object of that service in component constructor so that injector will create the instance of that service and inject it in side component class.
Let us create a service which returns me the list of cars when called Simple example of custom service service in angular –

1. **Create Service: -** service is just a class similar to component with any decorator, you can decorate the service using @Injectable decorator if you suppose to inject some dependency in service, if the service you are creating does not have dependency then we does not need to to decoreate the service using @Injectable attribute. We create data service which gives us list of car compnies as follows -

```
import { Injectable } from '@angular/core';
@Injectable()
export class DataService {
 constructor() { }
 cars = [
  'Ford','Chevrolet','Buick'
 ];
```

```
  myData() {
    return 'This is my data, man!';
  }
}
```

2. Register Service :-

We can register the service at two levels -

- We can do it at module level by importing service class and specifying it in providers array.

```
  // Other imports removed
  import { DataService } from './data.service';
  @NgModule({
    // Other properties removed
    providers: [DataService],
   })
```

- We can do it at compioent level by importing service class and specifying it in providers array.

```
  // Other imports removed
  import { DataService } from './data.service';
  @Component({
   // Other properties removed
   providers: [DataService],
  })
```

3. Define Dependency & Use IT:-

To use any service in any component we need to define the dependency in component by creating it's object in constructor of component class so that injector can create the instance of the service and inject in component constructor as follows-

```
  import {NgInit} from '@angular/core'
  import { DataService } from './data.service';
  @Component({
    // Other properties removed
    template: `
   <p>{{ someProperty }}</p>  `
  })
  export class AppComponent {
    constructor(private dataService:DataService) {
    }
  someProperty:string = '';
    ngOnInit() {
      console.log(this.dataService.cars);
      this.someProperty = this.dataService.myData();
    }
  }
```

Angular provides lot of built in services few of them are listed here –

1. Http
2. Router
3. Activated Router

Http Client Service in Angular: -

Angular has a built in service Http which will help you to make asynchorounous calls to the server and retrieve result form the server and process it inside your component class. Http Service will help us fetch external data, post to it, etc. We need to import the http module to make use of the http service. All these methods are asynchronous and return you Obserables to which you can subscribe your component and use those services.

Before going to http client we need to understand the observables, observables uses observer pattern , in observer pattern their will be observable who suppose to immit data over time period and their will be subscriber who suppose to subscribe that observer so that when observable emit data it will be received by the observer. Indirecly we can say observable is nothing but the data array whose values suppose to be emitted over a time. Observables also provides us opportunity to manipulate , transform calculate or to do some operation on the value received , that can be possible using observable attributes few popular attributes are –

- map
- filter
- take
- Merge etc.

Since Angular does not support observables we need to use reactive extensions to use the observable in angular and we can use it by importing them as follows –

```
import { Observable } from 'rxjs/Observable';
import { catchError, retry } from 'rxjs/operators';
```

Let us implement some example using angular and web api-

We want to do the curd of article and we had assumed that a Web Service / Web aPI for article is already implementd and running at – http://www.ritechpune.com/articles. Final UI or output will look as follows –

Create New Article

Enter Title	
Enter Category	
CREATE	

Article Details

Id	Title	Category		
1	Android AsyncTask Example	Android	Edit	Delete
2	Angular 2 Tutorial using CLI	Angular	Edit	Delete
3	Spring Boot Getting Started	Spring Boot	Edit	Delete

1. Use Http Module :-

Http service is part of HttpModule so you need to import it from @angular/http and add it at import selection in @NgModule decorator.

```
import { HttpModule } from '@angular/http';
@NgModule({
  imports: [
        HttpModule
  ],
})
export class AppModule { }
```

2. Create a Service:-

In this step we will create a service which allows us to make http calls (get/post/put/delete to web api/web service), import Http and define dependency on Http service.

```
import { Http } from '@angular/http';

@Injectable()
export class ArticleService {
  constructor(private http:Http) {
  }
  ------
```

3. Http Post Call:-

We will perform create operation using Angular Http.post() method. It hits the request URL using HTTP POST method. Http.post() method syntax is as follows.
post(url: string, body: any, options?: RequestOptionsArgs) : Observable<Response> The description of parameters is given as below.

url: This is the REST web service URL to create article.
body: This is of any type object that will be passed to REST web service server. In our example we will create an Angular class as Article and pass its instance to body parameter.
options: This is optional. This accepts the instance of Angular RequestOptions that is instantiated using Angular RequestOptionsArgs. Using RequestOptions we pass request parameter, request headers etc.

Http.post() returns instance of Observable. Observable is a representation of any set of values over any amount of time.
Find the code to create the article. Here we will use Angular Http.post() method.

```
createArticle(article: Article):Observable<number> {
    let cpHeaders = new Headers({ 'Content-Type': 'application/json' });
    let options = new RequestOptions({ headers: cpHeaders });
    return this.http.post(this.articleUrl, article, options)
```

```
        .map(success => success.status)
        .catch(this.handleError);
}
```

4. Http.get Call

We will perform read operation using Angular Http.get() method. It hits the URL using HTTP GET method. Find its syntax.
get(url: string, options?: RequestOptionsArgs) : Observable<Response> Find the description of the parameters.

url: Web service URL to read article.
options: This is optional. It is used to pass request parameter, headers etc.
Http.get() returns the instance of Observable.

Find the Angular code using Http.get() that will pass request parameter to filter the result.

```
getArticleById(articleId: string): Observable<Article> {
let cpHeaders = new Headers({ 'Content-Type': 'application/json' });
let cpParams = new URLSearchParams();
cpParams.set('id', articleId);
let options = new RequestOptions({ headers: cpHeaders, params: cpParams });
return this.http.get(this.articleUrl, options).map(this.extractData).catch(this.handleError);
}
```

URLSearchParams creates the query string in the URL.

5. Http.put Call:-

We will perform update operation using Angular 2 Http.put() method. It hits the URL using HTTP PUT method. Find its syntax.
put(url: string, body: any, options?: RequestOptionsArgs) : Observable<Response> Find the description of parameters.

url: This is the REST web service URL to update article.
body: This is of any type object that will be passed to REST web service server. In our example we will create an Angular class as Article and pass its instance to body parameter.
options: This is optional. This is used to pass request parameter, request headers etc.

Http.put() returns the instance of Observable.

```
updateArticle(article: Article):Observable<number> {
    let cpHeaders = new Headers({ 'Content-Type': 'application/json' });
    let options = new RequestOptions({ headers: cpHeaders });
    return this.http.put(this.articleUrl +"/"+ article.id, article, options)
          .map(success => success.status)
          .catch(this.handleError);
}
```

In our Angular application we have created a class as Article and we are passing its instance to Http.put() method. The article will be updated for the given article id.

6. Http.delete

We will perform delete operation using Angular Http.delete() method. Http.delete() hits the URL using HTTP DELETE method. Find its syntax.
delete(url: string, options?: RequestOptionsArgs) : Observable<Response> Find the description of the parameters.
url: Web service URL to delete article.
options: This is optional. It is used to pass request parameter, headers etc.

Find the code using Http.delete() method to delete article by id.

```
deleteArticleById(articleId: string): Observable<number> {
    let cpHeaders = new Headers({ 'Content-Type': 'application/json' });
    let options = new RequestOptions({ headers: cpHeaders });
    return this.http.delete(this.articleUrl +"/"+ articleId)
            .map(success => success.status)
   .catch(this.handleError);}
```

In the path parameter we are passing article id to delete the article.

Routing In Angular

Routing in angular enables us to move/navigate between the components of your application. Let us try to understand the routing mechanism in angular –
Most routing applications should add a <base> element to the index.html as the first child in the <head> tag to tell the router how to compose navigation URLs.

If the app folder is the application root, as it is for the sample application, set the href value exactly as shown here.

```
<base href="/">
```

The Angular Router is an optional service that presents a particular component view for a given URL. It is not part of the Angular core. It is in its own library package, @angular/router. Import what you need from it as you would from any other Angular package.

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

Configuration

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

A router has no routes until you configure it. The following example creates four route definitions, configures the router via the RouterModule.forRoot method, and adds the result to the AppModule's imports array.

src/app/app.module.ts (excerpt)

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'product/:id',      component: ProductDetailComponent },
  {
    component: ProductListComponent,
    data: { title: 'Product List' }
  },
  { path: '',
    redirectTo: '/products',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent }
];
@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
    // other imports here
  ],
  ...
})
export class AppModule { }
```

The appRoutes array of routes describes how to navigate. Pass it to the RouterModule.forRoot method in the module imports to configure the router.

Each Route maps a URL path to a component. There are no leading slashes in the path. The router parses and builds the final URL for you, allowing you to use both relative and absolute paths when navigating between application views.
The: id in the second route is a token for a route parameter. In a URL such as /product/42, "42" is the value of the id parameter. The corresponding ProductDetailComponent will use that value to find and present the product whose id is 42. You'll learn more about route parameters later in this guide.

Router outlet

Given this configuration, when the browser URL for this application becomes /products, the router matches that URL to the route path /products and displays the ProductListComponent after a RouterOutlet that you've placed in the host view's HTML.

<router-outlet></router-outlet>
<! -- Routed views go here -->

Router links

Now you have routes configured and a place to render them, but how do you navigate? The URL could arrive directly from the browser address bar. But most of the time you navigate as a result of some user action such as the click of an anchor tag.

Consider the following template:

src/app/app.component.ts (template)

```
template: `
  <h1>Angular Router</h1>
  <nav>
 <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
 <a routerLink="/products" routerLinkActive="active">Product List</a>
  </nav>
  <router-outlet></router-outlet>
`
```

The RouterLink directives on the anchor tags give the router control over those elements. The navigation paths are fixed, so you can assign a string to the routerLink (a "one-time" binding).

Router state

After the end of each successful navigation lifecycle, the router builds a tree of ActivatedRoute objects that make up the current state of the router. You can access the current RouterState from anywhere in the application using the Router service and the routerState property.

Each ActivatedRoute in the RouterState provides methods to traverse up and down the route tree to get information from parent, child and sibling routes.

Activated route

The route path and parameters are available through an injected router service called the ActivatedRoute. It has a great deal of useful information including:

url

An Observable of the route path(s), represented as an array of strings for each part of the route path.

data

An Observable that contains the data object provided for the route. Also contains any resolved values from the resolve guard.

paramMap

An Observable that contains a map of the required and optional parameters specific to the route. The map supports retrieving single and multiple values from the same parameter.

Angular Forms

Forms are the mainstay of business applications. You use forms to log in, submit a help request, place an order, book a flight, schedule a meeting, and perform countless other data-entry tasks.
Angular Allow you following types of forms -

1. Template Driven Forms
2. Model Driven Forms
3. Model Driven Forms using Form Builder

Template Driven Forms:-

You can build forms by writing templates in the Angular template syntax with the form-specific directives and techniques

Angular provides form-specific directives that you can use to bind the form input data and the model. The form-specific directives add extra functionality and behavior to a plain HTML form. The end result is that the template takes care of binding values with the model and form validation.

To Build Template Driven Form

- Add FormsModule to app.module.ts.
- Create a class for the Customer model.
- Create initial components and layout for the product form in Template(Using Html).
- Use Angular form directives like ngModel, ngModelGroup, and ngForm.
- Add validation using built-in validators & display validation errors meaningfully.
- Handle form submission using ngSubmit.

- **Add Forms Module to AppModule.ts**

  To use the template-driven form directives, we need to import the FormsModule from @angular/forms and add it to the imports array in app.module.ts.

  app/app.module.ts

```
import { FormsModule } from '@angular/forms';
@NgModule({
 imports: [
BrowserModule,
FormsModule
],
})
export class AppModule { }
```

- **Customer Model Class as Follows –**

```
 import {Component} from '@angular/core'
import {ICustomer} from './customer.interface'
import {CustomerService} from './customer.service'
@Component(    {
```

```
      selector:'custnew',
      templateUrl:'customernew.component.html',
 styles:['input.ng-invalid {border-right:4px solid red} input.ng-valid{border-right:4px solid
green;}'],
      providers:[CustomerService]
  })
export class CustomerNew{
   constructor(private _custsvr:CustomerService)    { }
   SaveCustomer(rec:ICustomer):void     {
        this._custsvr.postCustomer(rec)
        .subscribe((data)=> alert("Response:"+ data));
    } }
```

- **Use Angular directives to build the template driven form –**

```
<h2>New Customer </h2>
 <form    #frm="ngForm"  >
   <div>
      Enter Name:
      <div>
        <input name="CustomerName"  />
      </div>
   </div>
   <div>
       Email ID:
       <div>
         <input type="text"  name="EmailID"  required ngModel />
       </div>
    </div>
    <div>
       Mobile No:
       <div>
         <input type="text"  name="MobileNo" ngModel  />
       </div>
    </div>
    <div>
       Credit Limit:
       <div>
         <input type="text" name="CreditLimit" ngModel   />
       </div>
    </div>
   <!-- <div ngModelGroup="Address">
       <div>
         House No/Flat No:
       <div>
         <input type="text" name="HouseNo" ngModel />
```

```
        </div>
        </div>
        <div>
           Area:
        <div>
         <input type="text" name="Area" ngModel  />
        </div>
        </div>
        <div>
           Pin Code
           <div>
              <input type="text" name="PinCode" ngModel />
           </div>
        </div>
        </div> -->
     <div>
        <input type="submit" value="Save" [disabled]="!frm.form.valid" />
     </div>
</form>
```

- **Add Validation Errors and Messages as follows –**

Using ngModel in a form gives you more than just two-way data binding. It also tells you if the user touched the control, if the value changed, or if the value became invalid.

The NgModel directive doesn't just track state; it updates the control with special Angular CSS classes that reflect the state. You can leverage those class names to change the appearance of the control.
When state of model changes angular apply some classes, and set values of some built in properties to ture using which we can enable validation in angular with angular directives and few html5 attributes.

State	Class if True	Class if False
The control has been visited.	ng-touched	ng-untouched
The control's value has changed.	ng-dirty	ng-pristine
The control's value is valid.	ng-valid	ng-invalid

```
<h2>New Customer </h2>
<form #frm="ngForm"   (ngSubmit)="SaveCustomer(frm.value)">
   <div>
      Enter Name:
      <div>
<input #cname="ngModel" type="text" name="CustomerName"  ngModel  required
minlength="5" maxlength="10" />
         <div style="color:red" *ngIf="cname.errors && (cname.dirty || cname.touched)">
            <div [hidden]="!cname.errors.required">
               Customer Name Required
```

```
            </div>
            <div [hidden]="!cname.errors.minlength">
                  Invalid Minimum Length
            </div>
          </div>
          </div>
    </div>
```

- **Submit the form to component**

Component:-

```
SaveCustomer(rec:ICustomer):void
{
        this._custsvr.postCustomer(rec)
        .subscribe((data)=> alert("Response:"+ data));
}
```

Template:-

```
<form #frm=”ngForm”   (ngSubmit)=”SaveCustomer(frm.value)”>
    <div>
        <input type="submit" value="Save" [disabled]="!frm.form.valid" />
     </div>
```

Model Driven Forms / Reactive Forms in Angular.

Angular offers two form-building technologies: reactive forms and template-driven forms. The two technologies belong to the @angular/forms library and share a common set of form control classes. But they diverge markedly in philosophy, programming style, and technique. They even have their own modules: the ReactiveFormsModule and the FormsModule.
Angular reactive forms facilitate a reactive style of programming that favors explicit management of the data flowing between a non-UI data model (typically retrieved from a server) and a UI-oriented form model that retains the states and values of the HTML controls on screen. Reactive forms offer the ease of using reactive patterns, testing, and validation.

With reactive forms, you create a tree of Angular form control objects in the component class and bind them to native form control elements in the component template, using techniques described in this guide.

For Model Driven Form use – ReactiveFormsModule from '@angular/forms‘
Add it to imports in Root module.
Use FormGroup and FormControl classes from angular/forms
To Implment a class create a object of type FormGroup and Form Control as follows –

```
productform = new FormGroup({
      name: new FormControl("Sample"),
      address: new FormGroup(          {
            street: new FormControl(),
```

```
            pincode: new FormControl(),
        }),
      emailid: new FormControl(),
      mobileno: new FormControl()
  });
```

Create template as follows –

```
<form [formGroup]="productform" (ngSubmit)="OnSubmit()">
<input type="text" formControlName="name"  />
<div formGroupName="address">
<input type="text" formControlName="street" />
<input type="text" formControlName="pincode" />
<input type="text" formControlName="cityname" />
```

- **Validation in Model Driven Form:-**
  Every FormControl accept second parameter which will help you to define the validation for the formcontrol.
  You need to import Validators from angular/forms
  Validator's class provides you properties like required, minlength, maxlength that can be specified as array in second parameter of the FormControl.
  Once you define the validation in form control then you can use it in HTML.

Code in Component Class:-

```
name: new FormControl("Sample", [Validators.required, Validators.minLength(4),
Validators.maxLength(10)]),
 mobileno: new FormControl("",[Validators.required])
```

Code in Templates:-

```
{{ refname.className}}
                    </div>
[hidden]="(custfrm.controls.CustomerName.valid)||
(custfrm.controls.CustomerName.pristine)
          <div *ngIf="productform.controls['name'].hasError('required')">
             Plz Enter Value
          </div>
<div *ngIf="productform.controls['name'].hasError('minlength')">
             Plz Enter minimum 4 Characters
          </div>
<div *ngIf="productform.controls['name'].hasError('maxlength')">
             Plz Enter minimum 4 Characters
                  </div>
```

Disable Button:-

```
<input type="submit" [disabled]="productform.invalid" value="Save" />
```