CHAPTER 1

**Introduction**

**Angular**

* Angular is a framework to develop rich client-side applications for the web.
* It is an open source JavaScript framework designed by Google.
* Google changed the name from Angular.js to Angular because the structure of the application varies entirely.
* Applications developed using Angular framework are referred to as Angular applications.

**Angular Application**

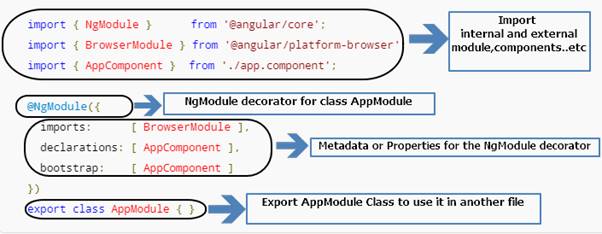
* An Angular application is a collection of modules.
* The execution of an Angular application starts from the module specified in the main.ts file.
* .ts files are written in TypeScript language.

**Example**

1. **import** { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
2. **import** { AppModule } from './app/app.module';
3. platformBrowserDynamic().bootstrapModule(AppModule);

**Angular Module**

* In general, Module is a collection of business/domain functions designed to achieve a single purpose. All the functions within module must work together cohesively to achieve the objective which is defined for the module.
* In Angular, a module is defined in the TypeScript file which is having the decorator @NgModule.
* [NgModule](https://angular.io/api/core/NgModule) is a function that takes a single metadata object, whose properties such as declarations, export, import, bootstrap, and providers, describe the module.



n the above diagram, @NgModule directive takes a single metadata object whose properties describe the module.

* declarations - the *View classes*that belong to this module. Mainly, Angular has three kinds of View classes: [components](https://angular.io/guide/architecture#components), [directives](https://angular.io/guide/architecture#directives), and [pipes](https://angular.io/guide/pipes).
* imports- other modules whose exported classes are needed by component templates declared in this.
* bootstrap- the main application View called the *root component*, that hosts all the other app Views. Only the *root module*should set this bootstrap property.

**Code Representation of NgModule Decoration**

1. **import** { NgModule } from '@angular/core';
2. **import** { BrowserModule } from '@angular/platform-browser';
3. **import** { AppComponent } from './app.component';
4. //Newly created and referenced here
5. **import** { StudentComponent } from './student/app.student'
7. @NgModule({
9. //Here Add all  your Modules used in project
10. imports: [BrowserModule,HttpModule],
12. //Here Add all your component,directives and pipes used in project
14. declarations: [AppComponent, StudentComponent,directive1,directive,pipe1,pipe2,],
16. //Here the component to be bootstrapped or start
17. bootstrap: [AppComponent]
18. })
19. **export** **class** AppModule { }

**Code Explanation**

* In Angular, we have some of the libraries such as '@angular/core', '@angular/platform-browser' and we can install them with the npm package manager.
* All the Angular libraries begin with @angular prefix.
* Many Angular libraries are modules (such as - [FormsModule](https://angular.io/api/forms/FormsModule" \t "_blank), [HttpModule](https://angular.io/api/http/HttpModule" \t "_blank), and [RouterModule](https://angular.io/api/router/RouterModule" \t "_blank).
* Modules can also add services to the application. Such services might be internally developed, such as the application logger. Services can come from outside sources, such as the Angular Router and Http client.

**Additional Note**

We can also say that Metadata can be used for the following purposes.

* Declare which components, directives, and pipes belong to the module.
* Make some of those classes public so that the other component templates can use them.
* Import other modules with the components, directives, and pipes needed by the components in this module.
* Provide services at the application level that any application component can use.

**Conclusion**

* NgModule decorates the above class. Once it is decorated, then the normal class will become an NgModule class.
* Every application should have at least one Module which is the *root* module that you [bootstrap](https://angular.io/guide/bootstrapping#main) to launch the application.
* And by default, Aangular app we will have an NgModule class called AppModule that is presented in the module.ts file.
* You can call it anything you want. The conventional name is AppModule.

CHAPTER 2

**Brief Info About Component**

* A Component is nothing but a simple TypeScript class where you can create your own methods and properties as per your requirement which is used to bind with a UI (html or cshtml page) of our application.
* In Angular 2, normal TypeScript class will become a Component class once it has been decorated with @component decorator.
* @component decorator provides an additional metadata that determines how the component should be processed, instantiated, and used at runtime.
* We can also say that Components are the most basic building blocks of a UI in an Angular application
* Components are also referred to as one of the types of directives just like a structural and attribute directive, which we will see in-depth in the upcoming article.
* Finally, component must belong to an NgModule in order for it to be usable by another component or application. To specify that a component is a member of an NgModule, you should list it in the declarationsfield of that NgModule

**Brief of @Component Decorators**

* Decorators in Typescript are like annotations in Java or attributes in C#.

Component decorator consists of many metadata properties or attributes, but in this article we will see the most frequently used five properties with an example those are selector, template, templateUrl, style and styleUrl.

Kindly refer this link for the complete list: *https://angular.io/api/core/ComponentDecorator*

**Selector**

It is used to replace the html at runtime in the index.html. The replacement is happening using the directive name. For example in the below code we have used the <myapp> as a directive and replace content as we want

**Simple Example**

File name: app.component.ts

1. **import** { Component } from "@angular/core";
3. //decorator
4. @Component({
6. selector: 'my-App',
7. template: '<h1>{{name}}</h1>'
8. })
10. **export** **class** AppComponent {
11. name: string = "Angular Framework"

}

File name index.html

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <title>Angular QuickStart</title>
5. <base href="/src/">
6. <meta charset="UTF-8">
7. <meta name="viewport" content="width=device-width, initial-scale=1">
8. <link rel="stylesheet" href="styles.css">
10. <!-- Polyfill(s) **for** older browsers -->
11. <script src="/node\_modules/core-js/client/shim.min.js"></script>
13. <script src="/node\_modules/zone.js/dist/zone.js"></script>
14. <script src="/node\_modules/systemjs/dist/system.src.js"></script>
16. <script src="systemjs.config.js"></script>
17. <script>
18. System.**import**('main.js').**catch**(**function**(err){ console.error(err); });
19. </script>
20. </head>
22. <body>
24. **<!--Here is the selector mapped-->**
25. **<my-app>Loading AppComponent content here ...</my-app>**


29. </body>
30. </html>

**Run the code**

**>ng serve--o**

**Template**It is used to specify the html which is shown as an output to the user

**Filename app.component.ts**

Example

1. **import** { Component } from "@angular/core";
3. @Component({
5. selector: 'my-App',

8. template: '<h1>Hi {{name}}</h1>'
9. })
11. **export** **class** AppComponent {
12. name: string = "Karthik"
13. }

**Important Note**  
  
In the above code, selector: 'my-App’ will refer to the before mentioned index.html which is the starting page of our application.



**TemplateUrl**   
  
URL to an external file containing a template for the view.

**Example**

FileName app.component.ts

1. **import** { Component } from "@angular/core";
3. @Component({
5. selector: 'my-App',
7. templateUrl: 'app/app.component.html'
8. })
10. **export** **class** AppComponent {
11. name: string = "Karthik"
12. }

**Note**  
Always remember that, the URL that is specified in the templateUrl property is always relative to the index.html of the angular project .

FileName app.component.html

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <title></title>
5. <meta charset="utf-8" />
6. </head>
7. <body>
8. <h1>This is {{name}} Home Page</h1>
9. </body>
10. </html>



**Additional Notes about Template and TemplateUrl**

In the component we can define either template or templateUrl and cannot define both template and templateUrl property. In other words, only one of templateUrl or template can be defined per Component.

**Styles** - inline-defined styles to be applied to this component's view.

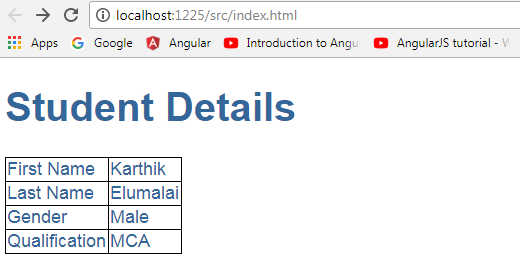
**Example**

**File name** app.student.ts

1. **import** { Component } from "@angular/core"
3. //decorater declaration
4. @Component({
6. selector: 'my-student',
7. templateUrl: 'app/student/app.student.html',
8. styles: ['table { color: #369; font-family: Arial, Helvetica, sans-serif; font-size: large; border-collapse: collapse;}', 'td {border: 1px solid black; }']
10. })
12. //class creation
13. **export** **class** StudentComponent {
14. firstName: string = "Karthik";
15. lastName: string = "Elumalai";
16. gender: string = "Male";
17. qualification: string = "MCA";
18. }

**File name**

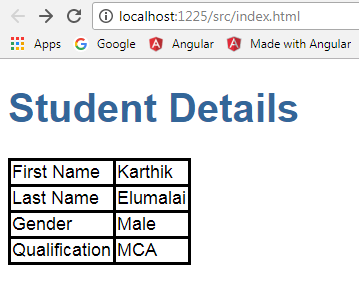
Output



**styleUrl**   
  
List of url style sheets to be applied to this component's view.

**Example**

1. **import** { Component } from "@angular/core"
3. //decorater declaration
4. @Component({
6. selector: 'my-student',
7. templateUrl: 'app/student/app.student.html',
8. //changed the property style to styleUrls
9. styleUrls: ['app/student/app.student.css']
11. })
13. //class creation
14. **export** **class** StudentComponent {
15. firstName: string = "Karthik";
16. lastName: string = "Elumalai";
17. gender: string = "Male";
18. qualification: string = "MCA";
19. }



**Nested Component**

If a component used by an another component then it is termed as nested component.

A component must belong to an NgModule in order for it to be usable by another component or application.

To specify that a component is a member of an NgModule, you should list it in the declarations field of that NgModule.

**Example**

Component 1 **app.component.ts**

1. **import** { Component } from "@angular/core";
3. @Component({
5. selector: 'my-App',
6. template: `
7. <div>
8. <h1>{{pageheader}}</h1>
9. </div>
10. //Here is the nesting happens by calling another component selector
12. <my-student></my-student>
13. `,
15. })
17. **export** **class** AppComponent {
18. pageheader: string = "Student Details"
19. }

**Component 2 app.student.ts**

1. **import** { Component } from "@angular/core"
3. //decorater declaration
4. @Component({
6. selector: 'my-student',
7. templateUrl : 'app/student/app.student.html'
9. })
11. //class creation
12. **export** **class** StudentComponent {
13. firstName: string = "Karthik";
14. lastName: string = "Elumalai";
15. gender: string = "Male";
16. qualification: string = "MCA";
17. }

**AppModule the application root module**

Every Angular app has a root module class. By convention, the root module class is called AppModule and it exists in a file named app.module.ts

AppModule is the root module which bootstraps and launches the angular application. You can name it anything you want, but by convention it is named AppModule

**Example**

**Filename app.module.ts**

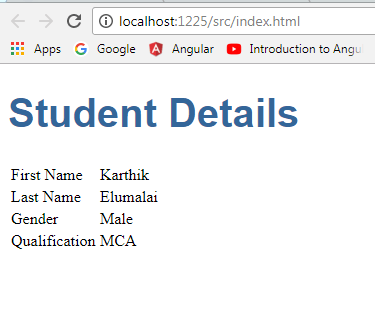
1. **import** { NgModule } from '@angular/core';
2. **import** { BrowserModule } from '@angular/platform-browser';
4. **import** { AppComponent } from './app.component';
5. **import** { StudentComponent } from './student/app.student'
7. @NgModule({
8. imports: [BrowserModule],
9. declarations: [AppComponent, StudentComponent],
10. bootstrap: [AppComponent]
11. })
12. **export** **class** AppModule { }

**Important Note**Once we entered our second component in the app.module.ts file, we can inject that component in any other component by using their selector.

Exact line in Component 1(app.component.ts) where nesting happens,

**template**

1. <div>
2. <h1>{{pageheader}}</h1>
3. </div>
4. <my-student></my-student>

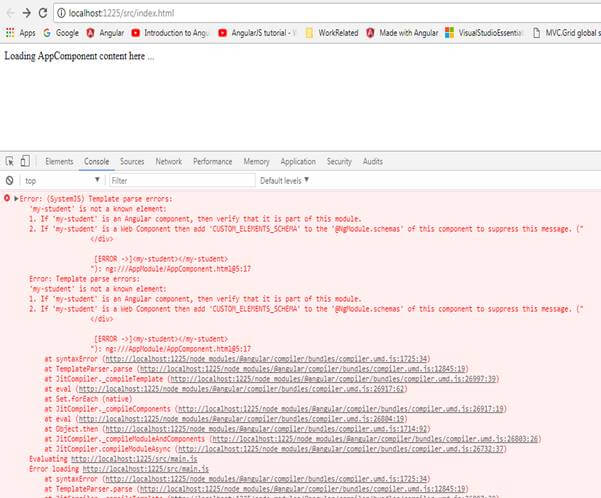


**Additional Note**

if you don’t register your component in the app.module.ts file and try to use it in the other components, then you won’t get the output and console will show the following error.

Same app.module.ts code without registering the second component,

1. **import** { NgModule } from '@angular/core';
2. **import** { BrowserModule } from '@angular/platform-browser';
4. **import** { AppComponent } from './app.component';
5. //import { StudentComponent } from './student/app.student'
7. @NgModule({
8. imports: [BrowserModule],
9. declarations: [AppComponent],
10. bootstrap: [AppComponent]
11. })
12. **export** **class** AppModule { }



CHAPTER 3 -DATA BINDING

Data binding is one of the most powerful features of software development technologies. Data binding is the connection bridge between View and the business logic (View Model) of the application. Data binding in Angular is the automatic synchronization between Model and the View. When the Model changes, the Views are automatically updated and vice-versa.

Classification of Data Binding

In the below sections, we will see both the concepts one by one with examples.

## One-way Data Binding: [Component to View]

It will bind the data from Component to View using the following different ways.

**Different types of one-way data binding**

* Interpolation Binding
* Property Binding

**Interpolation Binding**

* With interpolation, we place the component property name in the View template, enclosed in double curly braces: {{property Name}}.
* In simple words, interpolation is nothing but how we use this binding expression {{}} in our project. We will see that with an example.

File Name: app.component.ts

1. **import** { Component } from "@angular/core";
2. @Component({
3. selector: 'my-App',
4. template: `
5. <div>
6. <strong>{{firstname}}</strong>
7. <strong>{{lastname}}</strong>
8. </div>
9. })
10. **export** **class** AppComponent {
11. firstname: string = "Sachin";
12. lastname:string = "Tendulkar"
13. }



**Property Binding**

* It is used to bind values of component/model properties to the HTML element.
* Depending on the values, it will change the existing behavior of the HTML element.

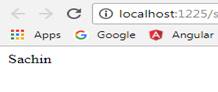
**Syntax**  
*[property] =’expression’*

In property binding, there is source and target. For this example, we can define it as [innerHTML] = 'firstname'. Here, innerHTML is a target that is a property of span tag and source is a component property i.e. firstname.

**Example**

1. **import** { Component } from "@angular/core";
3. @Component({
5. selector: 'my-App',
6. template: `
7. <div>
8. <span [innerHTML]='firstname'></span>
9. </div>
10. })
11. **export** **class** AppComponent {
12. firstname: string = "Sachin";
13. }

**Output**

****

Property Binding Demo2

**propertybindingdemo.component.ts**

1. **import** {
2. Component,
3. OnInit
4. } from '@angular/core';
5. @Component({
6. selector: 'app-propertybindingdemo',
7. templateUrl: './propertybindingdemo.component.html',
8. styleUrls: ['./propertybindingdemo.component.css']
9. })
10. **export** **class** PropertybindingdemoComponent **implements** OnInit {
11. name: string = ’jinal shah’;
12. constructor() {}
14. ngOnInit() {}
15. onTest() {
16. **return** **true**;
17. }
18. }

**propertybindingdemo.component.html**

1. <h1>Property Binding Demo</h1>
2. <input type="text" [value]="name"/>
3. <p [ngClass]="{redborder: onTest()}">is it styled?</p>
4. <p [ngStyle]="{color:'blue'}">Text Color</p>



**One-way binding**

In Angular , one-way data binding directive is replaced with [property]. Angular uses HTML DOM element property for one-way binding. The square brackets are used with property name for one-way data binding in Angular.

**app.component.ts**

1. **import** { Component } from '@angular/core';
2. @Component({
3. selector: 'test-app',
4. templateUrl: './app/databinding.html'
5. })
6. **export** **class** AppComponent {
7. name = 'sai krishna';
8. welcomeText = 'Welcome Sai!'
9. }

**databinding.html**

1. <div>
2. <h5>One way binding Example</h5>
3. Hello <span [innerText] = "name" ></span>!
4. <br/><br/>
5. <input type = 'text'  [value]="welcomeText" />
6. </div>

Run the program to see the output.

**Two-way binding**

The [(ngModel)] is used for two-way data binding. ngModel directive is part of a built-in Angular module called "FormsModule". So, we must import this module in to the template module before using the ngModel directive.

**app.module.ts**

1. **import** { NgModule }      from '@angular/core';
2. **import** { BrowserModule } from '@angular/platform-browser';
3. **import** { FormsModule } from '@angular/forms';
5. **import** { AppComponent }  from './app.component';
7. @NgModule({
8. imports:      [ BrowserModule, FormsModule],
9. declarations: [ AppComponent],
10. bootstrap:    [ AppComponent ]
11. })
12. **export** **class** AppModule {
13. }

**databinding.html**

1. <div>
2. <h5>Two way binding Example</h5>
3. Enter you Name:  <input [(ngModel)]="enterName"  />
4. <br/><br/>
5. <span> WelCome {{enterName}} ! </span>
6. </div>

**Event Binding**

Event binding flows or binds the data from an HTML element to a component.

**Syntax**

Within parentheses on the left of the equal sign, we have the target event ("click" in this case) and on the right side, we have the template statements such as component properties and methods.

1. <button (click)="onClick()">Click me</button>

In this case, the onClick() method of the component class is called when the click event occurs.

**Example**

File name app.component.ts

1. **import** { Component } from "@angular/core";
3. @Component({
5. selector: 'my-App',
6. template: `<button (click)='onClick()' >Click me</button>`
8. })
10. **export** **class** AppComponent {
11. onClick(): **void** {
12. console.log('you clicked me!!');
13. }
14. }

**Output**



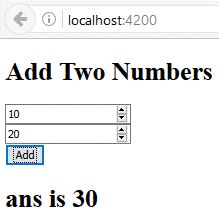
Event Binding Demo 2

**addtwonumberdemo.component.ts**

1. **import** { Component, OnInit } from '@angular/core';
2. @Component({
3. selector: 'app-addtwonumberdemo',
4. templateUrl: './addtwonumberdemo.component.html',
5. styleUrls: ['./addtwonumberdemo.component.css']
6. })
7. **export** **class** AddtwonumberdemoComponent **implements** OnInit {
8. number1:number=0;
9. number2:number=0;
10. ans:number=0;
11. constructor() { }
12. ngOnInit() {
13. }
14. add(){
15. **this**.ans=**this**.number1+**this**.number2;
16. }
17. }

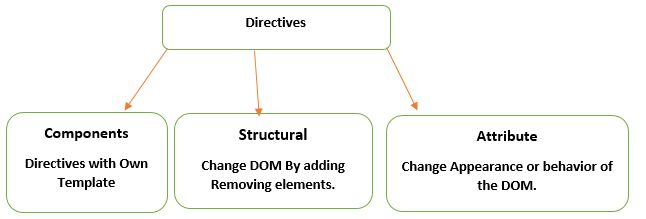
**addtwonumberdemo.component.html**

1. <h1>Add Two Numbers</h1>
2. <input type="number" [(ngModel)]="number1"/><br/>
3. <input type="number" [(ngModel)]="number2"/><br/>
4. <button (click)="add()"> Add</button>
5. <h1>ans is {{ans}}</h1>



CHAPTER 4- DIRECTIVES

Directives are elements which change the appearance or behavior of the DOM element. There are 3 types of the Directives mainly. We have categorized them, as follows -



**What are Structural Directives?**

Structural directives are responsible for the HTML layout. They shape or reshape the HTML view by simply adding or removing the elements in the DOM. These directives are the way to handle how the component or the element renders in a template.

There are basically 3 structural directives available in Angular.

1. NgIf (\*ngIf )
2. NgFor (\*ngFor)
3. NgSwitch (\*ngSwitch)

Structural directives are always preceded with the \* directive attribute name so as to make them easy to recognize.

**NgIf ( \*ngIf)**

This directive conditionally adds or removes the content from the DOM based on whether an expression is true or not.

Let's see the snippet for the component.

1. **import** {
2. Component,
3. OnInit
4. } from '@angular/core';
5. @Component({
6. selector: 'app-appngif',
7. templateUrl: './appngif.component.html',
8. styleUrls: ['./appngif.component.css']
9. })
10. **export** **class** AppngifComponent **implements** OnInit {
11. showdiv = **false**;
12. togglevisibilty(): **void** {
13. **this**.showdiv = !**this**.showdiv;
14. }
15. constructor() {}
16. ngOnInit() {}
17. }

In the above case, we have one property called showdiv which is Boolean property and we have one function called togglevisibility which reverses the value of the showdiv property.

Template for the same can be modified like below and can be seen as in the following code.

1. <input type="button" on-click='togglevisibilty()' value="Show Hide Button">
2. <div \*ngIf='showdiv'> This is Sample text</div> {{showdiv==**true**?sampletext:"mangesh"}}

We bind the function toggleVisibility() on the click of a button. When a click happens, it will call the function in the component and will revert the current value of the variable .

When we look at the \*ngIf directive, it is assigned with the value of the showDiv and depending on that, the div will be visible.

**NgFor (\*ngFor)**

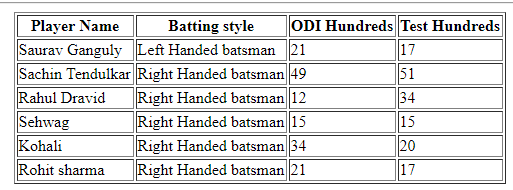
This is the directive which makes it easy to iterate over the collection like an object or an array and create a template for each of them. Let’s check the snippet for the basic ngFor that we have done below.

Let's check the component snippet to see how it gives us the data.

1. **import** {
2. Component,
3. OnInit
4. } from '@angular/core';
5. @Component({
6. selector: 'app-ng-for-directives',
7. templateUrl: './ng-for-directives.component.html',
8. styleUrls: ['./ng-for-directives.component.css']
9. })
10. **export** **class** NgForDirectivesComponent **implements** OnInit {
11. cricketers = [{
12. name: 'Saurav Ganguly',
13. BattingStyle: 'Left Handed batsman',
14. ODI: "21",
15. Test: "17"
16. }, {
17. name: 'Sachin Tendulkar',
18. BattingStyle: 'Right Handed batsman',
19. ODI: "49",
20. Test: "51"
21. }, {
22. name: 'Rahul Dravid',
23. BattingStyle: 'Right Handed batsman',
24. ODI: "12",
25. Test: "34"
26. }, {
27. name: 'Sehwag',
28. BattingStyle: 'Right Handed batsman',
29. ODI: "15",
30. Test: "15"
31. }, {
32. name: 'Kohali',
33. BattingStyle: 'Right Handed batsman',
34. ODI: "34",
35. Test: "20"
36. }, {
37. name: 'Rohit sharma',
38. BattingStyle: 'Right Handed batsman',
39. ODI: "21",
40. Test: "17"
41. }, ];
42. constructor() {}
43. ngOnInit() {}
44. }

Above is the simple collection of the cricketers data and we want to show them one by one on the screen. For this, we need to change the template into something like this.

1. <table border="1">
2. <tr>
3. <th>Player Name</th>
4. <th>Batting style</th>
5. <th>ODI Hundreds</th>
6. <th>Test Hundreds</th>
7. </tr>
8. <tr \*ngFor='let c of cricketers'>
9. <td>{{c.name}}</td>
10. <td>{{c.BattingStyle}}</td>
11. <td>{{c.ODI}}</td>
12. <td>{{c.Test}}</td>
13. </tr>
14. </table>

When we see the output, it will be like the below image. So far, we are able to show the data from the collection on the screen.  


Let’s analyze the syntax of the \*ngfor which is used in the above code. In that, it can be explained in the 3 sections -

1. let c :: it creates the local variable c which will be available in the template.
2. of cricketers :: it is the collection of the cricketers which will be iterated and which should be made available in the component.
3. \* :: it creates the parent template for the same.

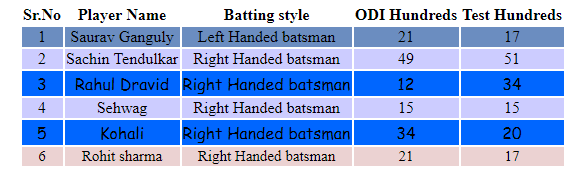
Let’s see some variants of the ngFor which are provided by Angular using various local variables, such as - index, last, first, odd, even. When we see the variable, index will return the index of the collection while other values give us the Boolean indicating the value of the property.

Code snippet for this demo is given below.

1. <table border="0">
2. <tr>
3. <th>Sr.No</th>
4. <th>Player Name</th>
5. <th>Batting style</th>
6. <th>ODI Hundreds</th>
7. <th>Test Hundreds</th>
8. </tr>
9. <tr \*ngFor='let c of cricketers let i = index let even=even,let odd=odd let first=first,let last=last' [ngClass]={first:first,odd:odd,even:even,last:last}>
10. <td>{{i+1}}</td>
11. <td>{{c.name}}</td>
12. <td>{{c.BattingStyle}}</td>
13. <td>{{c.ODI}}</td>
14. <td>{{c.Test}}</td>
15. </tr>
16. </table>

And the Styles that we have applied here can be found here in the css file of the component which has the following classes.

1. .odd {
2. background - color: rgba(0, 0, 255, 0.199);
3. text - align: center;
4. font - family: 'Times New Roman', Times, serif
5. }
6. .even
7. {
8. background - color: #0066ff;
9. text-align: center;
10. ;font-family:cursive
12. }
13. .first{
14. background-color: # 6 b8dc0;
15. text - align: center;
16. font - family: 'Times New Roman', Times, serif
17. }.last {
18. background - color: #ebd2d2;
19. text - align: center;
20. font - family: 'Times New Roman', Times, serif;
21. font - size: 15 px;
22. }



**ngSwitch(\*NgSwitch)**

NgSwitch is a directive which is bound to an expression. It is used to display the element tree based on the set of many elements. It has basic elements like below,

1. *ngSwitch*  
   we bind an expression to it
2. *ngSwitchCase*  
   it defines an element with the matched value we need to provide the \* before it.
3. *ngSwitchDefault*  
   This is the default case which will be executed if no match happened.

Let's see how it works for the example.

1. **import** {
2. Component,
3. OnInit
4. } from '@angular/core';
5. @Component({
6. selector: 'app-ngswitch',
7. templateUrl: './ngswitch.component.html',
8. styleUrls: ['./ngswitch.component.css']
9. })
10. **export** **class** NgswitchComponent **implements** OnInit {
11. Playerselection = ''
12. constructor() {}
13. ngOnInit() {}
14. checkChangedValue(changedvalue) {
15. **this**.Playerselection = changedvalue;
16. }
17. }

This is the component which has a method with one property, playerselction, which holds the value. Next is the checkchangedvalue() which will be called when the drop-down is changed and then assigns the value to the variable.

Let's see the template of the component.

1. <select (change)='checkChangedValue($event.target.value)'>
2. <option value="none" selected>Select</option>
3. <option value="SG">Sourav</option>
4. <option value="ST">Sachin</option>
5. <option value="RD">Rahul</option>
7. </select>
8. <div [ngSwitch]="playerselection">
9. <p \*ngSwitchCase="'SG'">Sourav selected</p>
10. <p \*ngSwitchCase="'ST'">Sachin selected</p>
11. <p \*ngSwitchCase="'RD'">Rahul selected</p>
12. <p \*ngSwitchDefault>NO player selected</p>
13. </div>

Here, we have the option value set and playerselection property in the component used as the expression to bind to the ngswitch. Whenever the change in the value occurs, it sends the value to the event in the component which apparently changes the value in the expression and the paragraphs are displayed accordingly.

**What is attribute directive?**

* It is mainly used to change/modify the behavior of the html element.
* As the name tells, it is used to change the attributes of the existing html element. In Angular 2 there are many built in attribute directives. Some of the useful ones are NgClass, NgStyle

**Built-In Attribute Directive: NgStyle, NgClass**

**NgStyle**NgStyle directive is similar to one of data binding technique called style binding in angular, but the main difference is, NgStyle used to set multiple inline styles for html element.

**NgClass**It is similar to NgStyle attribute but here it is using class attribute of the html element to apply the style.

**Simple Example**  
  
*NgStyle*

**File Name app.component.ts**

1. **import** { Component } from '@angular/core';
3. @Component({
4. selector: 'my-app',
5. template: ` <button style='color:blue' [ngStyle]="ApplyStyles()">Style Applied</button>
6. `
7. })
8. **export** **class** AppComponent {
9. isBold: **boolean** = **true**;
10. fontSize: number = 30;
11. isItalic: **boolean** = **true**;
13. ApplyStyles() {
14. let styles = {
15. 'font-weight': **this**.isBold ? 'bold' : 'normal',
16. 'font-style': **this**.isItalic ? 'italic' : 'normal',
17. 'font-size.px': **this**.fontSize
18. };
20. **return** styles;
21. }
22. }

**Output**

   
**Simple Example**

*NgClass*

**File Name app.component.ts**

1. **import** { Component } from '@angular/core';
3. @Component({
4. selector: 'my-app',
5. template: `
7. <button **class**='colorClass' [ngClass]='applyClasses()'>Style Applied Using Class</button>
9. `,
10. styles: [`
11. .boldClass{
12. font-weight:bold;
13. font-size : 30px;
14. }
16. .italicsClass{
17. font-style:italic;
18. }
20. .colorClass{
21. color:grey;
22. }
23. `]
24. })
25. **export** **class** AppComponent {
26. applyBoldClass: **boolean** = **true**;
27. applyItalicsClass: **boolean** = **true**;
29. applyClasses() {
30. let classes = {
31. boldClass: **this**.applyBoldClass,
32. italicsClass: **this**.applyItalicsClass
33. };
35. **return** classes;
36. }
37. }

**Output**



# **Angular - @Input() Directive**

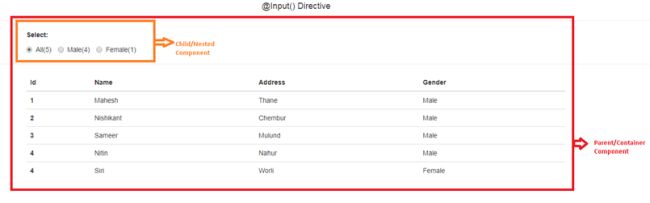
As we know that component is the basic building block of Angular applications, that means that an Angular application is a collection of components and it is possible that one component contains another component for the purpose of reusability. The component which contains another component is called a parent/container component and the component which is contained/present inside another component is called a child/nested component. Then:

* How these components communicate with each other?
* How do we send data from parent component to child component?
* How do we send data from child to parent component?

we will learn how to send data from parent component to the child component. We will build the below screen and will send the data from parent to child component using *@Input().*Please note that I have used bootstrap classes to add some basic style.

To build the below screen, we need two components,

1. *student-list.component.ts (Container/Parent component) - T*his component is responsible for displaying the student list.
2. *student-count.component.ts(Nested/Child component) - T*his component is responsible for displaying the 3 Radio buttons named: All, Male, Female, and we are calling this component inside student-list.component.html using the component’s selector directive.



Now, let’s open student-list.component.ts file and paste the below code.

1. **import** {
2. Component
3. } from '@angular/core'
4. **import** {
5. FormGroup,
6. FormControl
7. } from '@angular/forms'
8. @Component({
9. selector: 'my-app',
10. templateUrl: './app/student-list.component.html',
11. })
12. // This is parent(container) component which contains child(nested) Component(student-count.component.ts)
13. **export** **class** StudentListComponent {
14. **public** students: any; // this variable holds the list of students
15. constructor() {
16. //Initialized student array to hold some student object
17. **this**.students = [{
18. Id: 1,
19. Name: "Mahesh",
20. Address: "Thane",
21. Gender: "Male"
22. }, {
23. Id: 2,
24. Name: "Nishikant",
25. Address: "Chembur",
26. Gender: "Male"
27. }, {
28. Id: 3,
29. Name: "Sameer",
30. Address: "Mulund",
31. Gender: "Male"
32. }, {
33. Id: 4,
34. Name: "Nitin",
35. Address: "Nahur",
36. Gender: "Male"
37. }, {
38. Id: 4,
39. Name: "Siri",
40. Address: "Worli",
41. Gender: "Female"
42. }];
43. }
44. //Below for simplicity I have created 3 method to get the count of students based on the gender
45. //We can do that in single method by passing the parameter.
46. //we will use these functions to pass data to the input properties in child component
47. getTotalStudentsCount() {
48. **return** **this**.students.length;
49. }
50. //Get the maleStudentsCount
51. getMaleStudentsCount() {
52. **return** **this**.students.filter(student => student.Gender == 'Male').length;
53. }
54. //Get the femaleStudentsCount
55. getFemaleStudentsCount() {
56. **return** **this**.students.filter(student => student.Gender == 'Female').length;
57. }
58. }

In the above code, I have just created and initialized the "students" array in the constructor and added three methods which will return us the count of the students based on the gender and we are going to use these methods to give values to input properties using property binding.

Now, open student-list.component.html to add some HTML code.

1. <body>
2. <!--Here we are using selector as directive to call (render) the child component(student-count.component.ts)-->
3. <!--Now using the property binding **in** selector tag we are passing data to input properties
5. present **in** child component(student-count.component.ts)
6. ex.[totalStudentsCount]="getTotalStudentsCount()"
8. -->
9. <student-count [totalStudentsCount]="getTotalStudentsCount()" -- [maleStudentsCount]="getMaleStudentsCount()" [femaleStudentsCount]="getFemaleStudentsCount()"> </student-count>
10. <hr>
11. <div **class**="container">
12. <table **class**="table table-sm">
13. <thead>
14. <tr>
15. <th scope="col">Id</th>
16. <th scope="col">Name</th>
17. <th scope="col">Address</th>
18. <th scope="col">Gender</th>
19. </tr>
20. </thead>
21. <tbody>
22. <!--using \*ngFor directive to iterate over student list-->
23. <tr \*ngFor="let student of students">
24. <th scope="row">{{student.Id}}</th>
25. <td>{{student.Name}}</td>
26. <td>{{student.Address}}</td>
27. <td>{{student.Gender}}</td>
28. </tr>
29. </tbody>
30. </table>
31. </div>
32. </body>

In the above HTML code, I have used child component, i.e., student-count component selector as a directive to render inside the student-list component and I am passing the values to the input properties of child component using property binding.

1. <student-count [totalStudentsCount]="getTotalStudentsCount()" --[totalStudentCount] is the input property **in** student-count.component and I am assigning values by calling respective method which is present **in** .ts file [maleStudentsCount]="getMaleStudentsCount()" [femaleStudentsCount]="getFemaleStudentsCount()">
2. </student-count>

And then, I am rendering the table using the \*ngFor directive.

Now, open the student-count.component.ts file and paste the following code.

1. **import** {
2. Component,
3. Input
4. } from "@angular/core";
5. @Component({
6. selector: 'student-count',
7. templateUrl: './app/student-count.component.html'
8. })
9. **export** **class** StudentCountComponent {
10. //here we have decorated variables using @Input directive means that these
11. //variables will get data from parent component
12. //in our case,
13. //it is from student-list.component
14. @Input()
15. totalStudentsCount: any;
16. @Input()
17. maleStudentsCount: any;
18. @Input()
19. femaleStudentsCount: any;
20. //this variable is use to select the All Radio button by default
21. selectedRadioButton: any = "All";
22. }

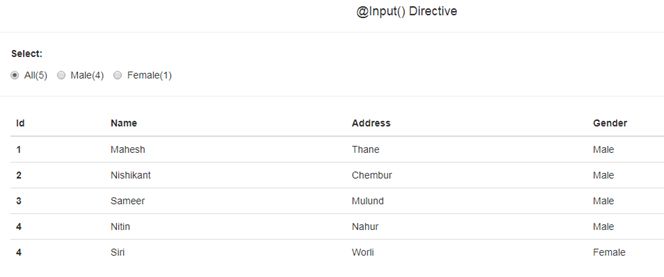
Here, I have declared 3 variables and decorated those variable with *@Input()* directives which means these variables will get values from the parent component.

See the comments in the above code for more information.

Now, open the student-count.component.html file and paste the following code.

1. <div **class**="container">
2. <p style="font-weight: bold;">Select:</p>
3. <form> <label **class**="radio-inline">
5. <input type="radio" name="optradio" value="All" [(ngModel)]="selectedRadioButton">All({{totalStudentsCount}})
7. </label> <label **class**="radio-inline">
9. <input type="radio" name="optradio" value="Male" [(ngModel)]="selectedRadioButton">Male({{maleStudentsCount}})
11. </label> <label **class**="radio-inline">
13. <input type="radio" name="optradio" value="Female" [(ngModel)]="selectedRadioButton">Female({{femaleStudentsCount}})
15. </label> </form>
16. </div>

In the above code, I’m rendering 3 radio buttons using HTML and displaying the respected count of students based on gender using [interpolation](https://lishman.io/angular-2-interpolation) ( {{ //TODO }} ).

**HTML output screenshot  
**

# **Angular - Creating Custom Pipe(|)**

* How to create a custom pipe in Angular

As we already know that Pipes (|) in Angular are used to transform the data before displaying in a browser, so in this article, my requirement is to filter the student's record based on the value, i.e., name of the student entered in the search textbox given at the top.

See the below screen.



Once the user enters anything in the textbox, the student record will get filtered based on the name of the student. Please note that I have used bootstrap classes to apply the basic styles.

To build the above screen, I have used only one component (AppComponent) and Respective HTML template file. To create the custom pipe, I have created one TypeScript file and implemented PipeTransforms interface and implemented the custom logic inside the transform method.

**Step 1 - Create the custom pipe**

To create custom pipe, create one .ts file and paste the below code.

1. **import** { Pipe } from "@angular/core";
2. **import** { PipeTransform } from "@angular/core";
4. @Pipe({
5. name: 'namefilterpipe' //name of the pipe
7. })
8. // This custom pipe implements the PipeTransform interface
9. // as you can see in below code I have provided the custom implementation to the transform method to filter the record
10. //based on the character typed in the search textbox
11. **export** **class** NamePipe **implements** PipeTransform {
12. transform(studentList: any, searchText: any) {
13. **if** (searchText)
14. **return** studentList.filter(x => x.Name.toLowerCase().startsWith(searchText.toLowerCase());
16. **return** studentList;
17. }
19. }

As you can see in the above code, I have implemented Pipetransform interface and provided the body to transform method and in the decorator, i.e, @Pipe, I have given the name of the pipe as namefilterpipe. The transform method takes two arguments as we are applying a filter on the student list. The first argument will be the student list and the second argument will be our textbox value which a user will type. And based on the input values, I am filtering the student’s record and returning it.

After creating the file, declare that pipe inside the declaration section of the AppModule.

**Step 2**Open the app.commponent.ts file and paste the following code.

1. **import** { Component } from '@angular/core'
2. @Component({
3. selector: 'my-app',
4. templateUrl: './app/app.component.html',
5. })
7. **export** **class** AppComponent {
9. **public** students: any;// this variable holds the list of students
10. **public** searchText: any;
12. constructor() {
13. **this**.students = [
14. { Id: 1, Name: "Mahesh", Address: "Thane", Gender: "Male" },
15. { Id: 2, Name: "Nishikant", Address: "Chembur", Gender: "Male" },
16. { Id: 3, Name: "Sameer", Address: "Mulund", Gender: "Male" },
17. { Id: 4, Name: "Nitin", Address: "Nahur", Gender: "Male" },
18. { Id: 4, Name: "Siri", Address: "Worli", Gender: "Female" }
19. ];
20. }
21. }

As you can see, in the above code, I have just created one student's array and I am initializing this array inside constructor.

**Step  3**

Now, open the app.component.html and paste the following code.

1. <div **class**="container">
2. <b>Enter Name :</b> <input type="text" [(ngModel)]="searchText"><br/>
3. <table **class**="table table-sm">
4. <thead>
5. <tr>
6. <th scope="col">Id</th>
7. <th scope="col">Name</th>
8. <th scope="col">Address</th>
9. <th scope="col">Gender</th>
10. </tr>
11. </thead>
12. <tbody>
13. <!--using \*ngFor directive to iterate over student list-->
14. <tr \*ngFor="let student of students | namefilterpipe : searchText">
15. <th scope="row">{{student.Id}}</th>
16. <td>{{student.Name }}</td>
17. <td>{{student.Address}}</td>
18. <td>{{student.Gender}}</td>
19. </tr>
20. </tbody>
21. </table>
22. </div>

Here,  I am displaying the students in the table using \*ngFor structural directive and at the same time, I have applied the custom filter; i.e namefilterpipe, and passing the entered value as a parameter.

See the following code snippet.

1. <tr \*ngFor="let student of students | namefilterpipe : searchText">

Run the code and see the below output screen.



# **Angular @Output() Directive**

we are going to talk about how to send data from child component to parent component using the @Output() directive. For simplicity, I am going to use the same code which I have used in my previous article and I will add some more code to explain output directive so in case you missed reading my previous article, then I would recommend you read that first, using the above link.



See the above screen; in this example, my requirement is to notify and send the selected radio button value, i.e., [Male, Female, All] from child component to parent component so that once the user selects the radio button, the respective data gets displayed in the parent, i.e., student list component.

For exxample – if a user selects Male, then student list component will display only male student and so on. For this, we’ll start from the child component, i.e., student-count component.

**Step 1**

Let’s open student-count.component.ts file and paste the below code.

1. **import** { Component, Input, Output, EventEmitter } from "@angular/core";
2. @Component({
3. selector: 'student-count',
4. templateUrl: './app/student-count.component.html'
5. })
6. **export** **class** StudentCountComponent {
7. @Input()
8. totalStudentsCount: any;
9. @Input()
10. maleStudentsCount: any;
11. @Input()
12. femaleStudentsCount: any;
13. //this variable is use to select the All Radio button by default
14. selectedRadioButton: any = "All";
16. //Now once user click on the radio button we need to raise custom event and notify the parent component
17. //and along with the event we need to send the selected radio button value
18. //That's why I have created the below property and used @Output() directive to turn that into event
19. //and I am using one more class i.e EventEmitterClass to create the custom event
20. //And while creating the object of the EventEmitter I am Passing string as a Payload so that we will send selected radio button value to the parent component
21. @Output()
22. radioButtonSelectionChanged: EventEmitter<string> = **new** EventEmitter<string>();
23. //This function will get called when someone clicks on the radio button
24. //And to this function I am passing the selected radio Button value and inside this function will raise our custom event radioButtonSelectionChanged
25. //using the emit method of EventEmitter class
26. selectedRadioButtonChanged(selectedRadioButtonValue) {
27. **this**.radioButtonSelectionChanged.emit(selectedRadioButtonValue);
28. }
29. }

In the above code, I have just created the custom event using the @Output directive and the Event Emitter class which is present in the @angular/Core package.

And then, I have added one simple function, i.e., selectedRadioButtonChange() which gets called when user clicks on the any, i.e., All, Male, Female, radio button. In that function, we are raising the custom event using the emit method of the EventEmitter class and we are passing the selected radio button value as payload to custom event.

**Step 2**

Open student-count.component.html to add some HTML code.

1. <div **class**="container">
2. <p style="font-weight: bold;">Select:</p>
3. <form>
4. <label **class**="radio-inline">
5. <!-- In below tag I have just added **this** event (change)="selectedRadioButtonChanged('All')"  and the I am passing selected radio button value
6. To the event handler **function**. Same way I have done **for** remaining radio buttons
7. -->
8. <input type="radio" name="optradio" value="All" [(ngModel)]="selectedRadioButton" (change)="selectedRadioButtonChanged('All')">All({{totalStudentsCount}})
9. </label>
10. <label **class**="radio-inline">
11. <input type="radio" name="optradio" value="Male" [(ngModel)]="selectedRadioButton" (change)="selectedRadioButtonChanged('Male')">Male({{maleStudentsCount}})
12. </label>
13. <label **class**="radio-inline">
14. <input type="radio" name="optradio" value="Female" [(ngModel)]="selectedRadioButton" (change)="selectedRadioButtonChanged('Female')">Female({{femaleStudentsCount}})
15. </label>
16. </form>
17. </div>

In the above HTML code, I have added a change event on each and every radio button which will eventually raise our custom event (see the comments in HTML code for more information).

**Step 3**

Open the student-list.component.html file and paste the following code.

1. <h4 style="text-align: center">@Output() Directive(To send data from child to parent)</h4>
2. <hr>
3. <html lang="en">
4. <head>
5. <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
6. </head>
7. <body>
8. <!--Here we are using selector as directive to call(render) the child component(student-count.component.ts) -->
9. <!--Now using the property binding **in** selector tag we are passing data to input properties
10. present **in** child component(student-count.component.ts)
11. ex.[totalStudentsCount]="getTotalStudentsCount()"
12. -->
13. <!--and same we are binding our custom event -->
14. <student-count [totalStudentsCount]="getTotalStudentsCount()"
15. [maleStudentsCount]="getMaleStudentsCount()"
16. [femaleStudentsCount]="getFemaleStudentsCount()"
17. (radioButtonSelectionChanged)="radioButtonChangedEventHandler($event)">
18. </student-count>
19. <hr>
20. <div **class**="container">
21. <table **class**="table table-sm">
22. <thead>
23. <tr>
24. <th scope="col">Id</th>
25. <th scope="col">Name</th>
26. <th scope="col">Address</th>
27. <th scope="col">Gender</th>
28. </tr>
29. </thead>
30. <tbody>
31. <!--using \*ngFor directive to iterate over student list-->
32. <ng-container \*ngFor="let student of students">
33. <tr \*ngIf="selectedRadioButtonValue=='All' || selectedRadioButtonValue == student.Gender">
34. <th scope="row">{{student.Id}}</th>
35. <td>{{student.Name}}</td>
36. <td>{{student.Address}}</td>
37. <td>{{student.Gender}}</td>
38. </tr>
39. </ng-container>
40. </tbody>
41. </table>
42. </div>
43. </body>
45. </html>

In the above code of student list component, I have used my previous article's code as it is and apart from that, I have added the below line on my child components selector directive to raise and handle the custom event.

1. (radioButtonSelectionChanged)="radioButtonChangedEventHandler($event)"

So, we are raising radioButtonSlection changed custom event using the @Output and EventEmitter class and radioButtonChangedEventHandler($event)" is our event handler function which will get called whenever the user clicks on any of the radio buttons.

1. <tr \*ngIf="selectedRadioButtonValue=='All' || selectedRadioButtonValue == student.Gender">

 And in my code, I am using one more directive; i.e. ng-container directive, as we cannot add two structure directives on the same element - \*ngIf and \*ngFor

1. <ng-container \*ngFor="let student of students">

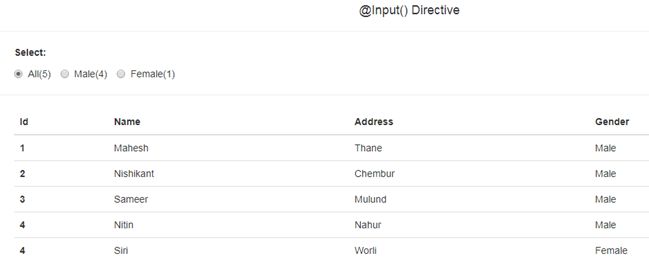
 See the comments in the above code for more information.

**Step 4**

Open the student-list.component.ts file and paste the following code.

1. **import** { Component } from '@angular/core'
2. **import** { FormGroup, FormControl } from '@angular/forms'
3. @Component({
4. selector: 'my-app',
5. templateUrl: './app/student-list.component.html',
6. })
7. // This is parent(container) component which contains child(nested) Component(student-count.component.ts)
8. **export** **class** StudentListComponent {
9. selectedRadioButtonValue: any = 'All';
10. **public** students: any;// this variable holds the list of students
11. constructor() {
12. **this**.students = [
13. { Id: 1, Name: "Mahesh", Address: "Thane", Gender: "Male" },
14. { Id: 2, Name: "Nishikant", Address: "Chembur", Gender: "Male" },
15. { Id: 3, Name: "Sameer", Address: "Mulund", Gender: "Male" },
16. { Id: 4, Name: "Nitin", Address: "Nahur", Gender: "Male" },
17. { Id: 4, Name: "Siri", Address: "Worli", Gender: "Female" }
18. ];
19. }
20. //Below for simplicity i have created 3 method to get the count of students based on the gender
21. //We can do that in single method by passing the parameter.
22. getTotalStudentsCount() {
23. **return** **this**.students.length;
24. }
25. //Get the maleStudentsCount
26. getMaleStudentsCount() {
27. **return** **this**.students.filter(student => student.Gender == 'Male').length;
28. }
29. //Get the femaleStudentsCount
30. getFemaleStudentsCount() {
31. **return** **this**.students.filter(student => student.Gender == 'Female').length;
32. }
33. radioButtonChangedEventHandler(radioButtonValue) {
34. **this**.selectedRadioButtonValue = radioButtonValue;
35. }
36. }

In the above code, I have used the same code from the input directive example and apart from that, I have added one event handler function which will handle our custom event. In this function, I’m just assigning the value to one variable which we are using to filter the data in our HTML template.

**Html output screenshot**  


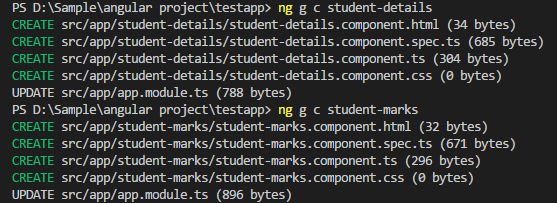
In the above image, we are getting the respective students once the user clicks on the radio button using @Output() directive. In this way, we can pass data from Child component to Parent component and create and raise the custom event using the @Output directive and Event Emitter class.

SERVICES IN ANGULAR

Let us start with a simple application having details of students in two different comonents. Create two components -

* >ng g c student-details
* >ng g c student-marks

The following files will be added.



Open student-details.component.ts and add the below code.

1. **import** { Component, OnInit } from '@angular/core';
3. @Component({
4. selector: 'app-student-details',
5. templateUrl: './student-details.component.html',
6. styleUrls: ['./student-details.component.css']
7. })
8. **export** **class** StudentDetailsComponent **implements** OnInit {
10. **public** students = [
11. {"id" : 1001, "name" : "Irshad", "marks" : 90},
12. {"id" : 1002, "name" : "Imran", "marks" : 80},
13. {"id" : 1003, "name" : "Rahul", "marks" : 70},
14. {"id" : 1004, "name" : "Ajay", "marks" : 85},
15. {"id" : 1005, "name" : "Sunny", "marks" : 60}
16. ];
17. constructor() { }
18. ngOnInit() {
19. }
20. }

Open student-details.component.html and add the below lines of code.

1. <h2>Student Details:</h2>
2. <div \*ngFor="let stud of students">
3. Id : {{stud.id}}, Name : {{stud.name}}
4. </div>

Open student-marks.component.ts and add the below lines.

1. **import** { Component, OnInit } from '@angular/core';
3. @Component({
4. selector: 'app-student-marks',
5. templateUrl: './student-marks.component.html',
6. styleUrls: ['./student-marks.component.css']
7. })
8. **export** **class** StudentMarksComponent **implements** OnInit {
10. **public** students = [
11. {"id" : 1001, "name" : "Irshad", "marks" : 90},
12. {"id" : 1002, "name" : "Imran", "marks" : 80},
13. {"id" : 1003, "name" : "Rahul", "marks" : 70},
14. {"id" : 1004, "name" : "Ajay", "marks" : 85},
15. {"id" : 1005, "name" : "Sunny", "marks" : 60}
16. ];
17. constructor() { }
18. ngOnInit() {
19. }
20. }

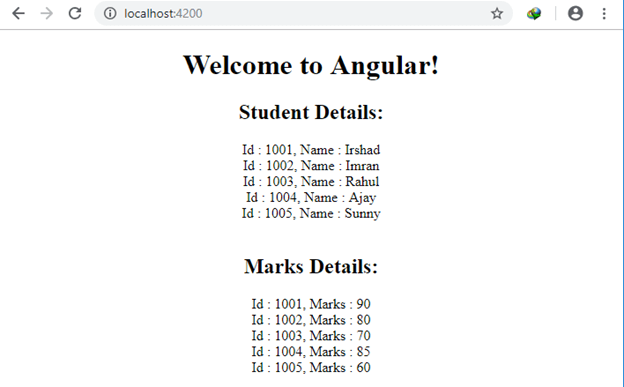
Open student-marks.component.html and add this code.

1. <h2>Marks Details:</h2>
2. <div \*ngFor="let stud of students">
3. Id : {{stud.id}}, Marks : {{stud.marks}}
4. </div>

Open app.component.html and add the following code.

1. <!--The content below is only a placeholder and can be replaced.-->
2. <div style="text-align:center">
3. <h1>
4. Welcome to {{ title }}!
5. </h1>
6. <app-student-details></app-student-details>
7. <br>
8. <app-student-marks></app-student-marks>
9. </div>

Run the application.



The above application is working but it is not a good programming practice. You can see that we have added the same set of data to both the files - student-details.component.ts and in student-marks.component.ts.

1. **public** students = [
2. {"id" : 1001, "name" : "Irshad", "marks" : 90},
3. {"id" : 1002, "name" : "Imran", "marks" : 80},
4. {"id" : 1003, "name" : "Rahul", "marks" : 70},
5. {"id" : 1004, "name" : "Ajay", "marks" : 85},
6. {"id" : 1005, "name" : "Sunny", "marks" : 60}
7. ];

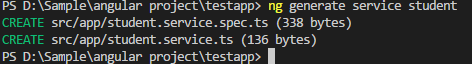
This same data is being used in both the components, which means the component is not only responsible for displaying or working with the data, it is also taking the responsibility of generating the data, which actually should not be its responsibility. These components should only be responsible to use the given data. This is breaking the single responsibility rule of the programming.

Here comes Angular Services into the picture. We should have a service that will be injected into these two components and should solely be responsible to deliver the data to these components. These components should only perform one responsibility: dealing with how to use the data given by the service.

So, when we inject the service, the component will only use the data given by the Service instead of knowing how the data is coming from that service. This makes the service responsible for generating the data and components are responsible to make it display.

Let us start creating a service and use that in your application.

*>ng generate service student*



Open student.service.ts and add the below code.

1. **import** { Injectable } from '@angular/core';
3. @Injectable({
4. providedIn: 'root'
5. })
6. **export** **class** StudentService {
7. students = [
8. {"id" : 1001, "name" : "Irshad", "marks" : 90},
9. {"id" : 1002, "name" : "Imran", "marks" : 80},
10. {"id" : 1003, "name" : "Rahul", "marks" : 70},
11. {"id" : 1004, "name" : "Ajay", "marks" : 85},
12. {"id" : 1005, "name" : "Sunny", "marks" : 60}
13. ];
14. constructor() { }
15. getStudents(){
16. **return** **this**.students;
17. }
18. }

Open app.module.ts and add these services in provider.

1. **import** { BrowserModule } from '@angular/platform-browser';
2. **import** { NgModule, Pipe } from '@angular/core';
3. **import** {FormsModule} from '@angular/forms';
4. **import** { AppComponent } from './app.component';
5. **import** { TestComponent } from './test/test.component';
6. **import** { StudentDetailsComponent } from './student-details/student-details.component';
7. **import** { StudentMarksComponent } from './student-marks/student-marks.component';
9. @Pipe({
10. name: "namePipe"
11. })
13. **class** NamePipe{
14. transform(value : string, defaultValue : string) : string{
15. **if**(value != ""){
16. **return** value;
17. } **else** {
18. **return** defaultValue;
19. }
20. }
21. }
23. @NgModule({
24. declarations: [
25. AppComponent,
26. TestComponent,
27. NamePipe,
28. StudentDetailsComponent,
29. StudentMarksComponent
30. ],
31. imports: [
32. BrowserModule,
33. FormsModule
34. ],
35. providers: [],
36. bootstrap: [AppComponent]
37. })
38. **export** **class** AppModule { }

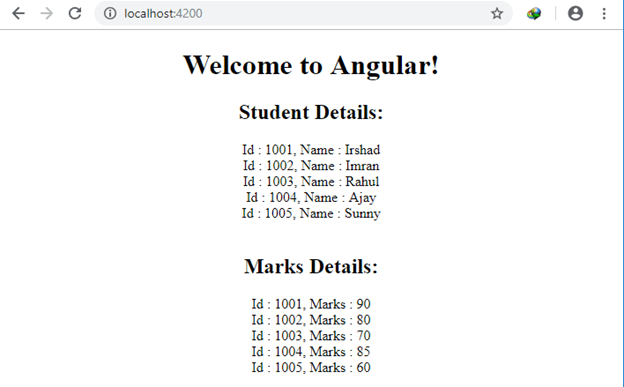
Open student-details.component.ts and add the below code.

1. **import** { Component, OnInit } from '@angular/core';
2. **import** { StudentService } from '../student.service';
4. @Component({
5. selector: 'app-student-details',
6. templateUrl: './student-details.component.html',
7. styleUrls: ['./student-details.component.css']
8. })
9. **export** **class** StudentDetailsComponent **implements** OnInit {
11. **public** students = [];
12. constructor(**private** studentService : StudentService) {
13. **this**.students = studentService.getStudents();
14. }
15. ngOnInit() {
16. }
17. }

Open student-marks.component.ts and add this code.

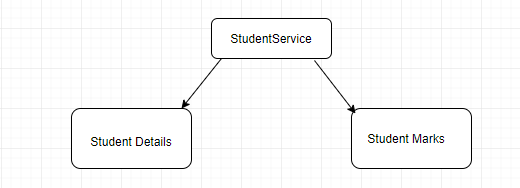
1. **import** { Component, OnInit } from '@angular/core';
2. **import** { StudentService } from '../student.service';
4. @Component({
5. selector: 'app-student-marks',
6. templateUrl: './student-marks.component.html',
7. styleUrls: ['./student-marks.component.css']
8. })
9. **export** **class** StudentMarksComponent **implements** OnInit {
11. **public** students = [];
12. constructor(**private** studentService : StudentService) {
13. **this**.students = studentService.getStudents();
14. }
15. ngOnInit() {
16. }
17. }

Run the application.



You have achieved the same thing that we got in our previous example, however, this time, the data is coming from a service and our component is only performing single functionality, i.e., to render the data.

So, we have the application that is utilizing the service and we know what the dependency injection is and how we avoid the repetition of code.

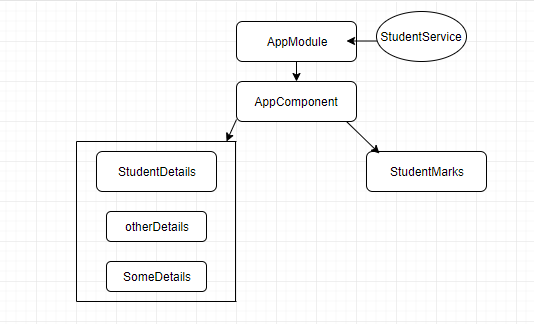


So. in our application, there is a student service that provides the details and marks of the students and we don’t have to worry about the data in both the components. We are simply getting the data and rendering it to the Template View of both the components.

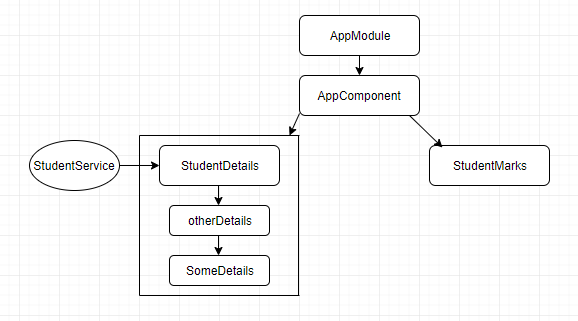
When we generate a service through Angular CLI, it will generate the basic template of the service. We have modified the contents and completed the first step in generating the service.

1. **import** { Injectable } from '@angular/core';
3. @Injectable({
4. providedIn: 'root'
5. })
6. **export** **class** StudentService {
7. students = [
8. {"id" : 1001, "name" : "Irshad", "marks" : 90},
9. {"id" : 1002, "name" : "Imran", "marks" : 80},
10. {"id" : 1003, "name" : "Rahul", "marks" : 70},
11. {"id" : 1004, "name" : "Ajay", "marks" : 85},
12. {"id" : 1005, "name" : "Sunny", "marks" : 60}
13. ];
14. constructor() { }
15. getStudents(){
16. **return** **this**.students;
17. }
18. }

Now, the second step is to register the service. If we don’t register the service, then it will be considered as a regular class of Angular. There is a different way of registering the service in Angular. It will maintain the hierarchy of dependency injection.



If we register the service to the AppModule, i.e, in the top level, then it will be available to all the components. AppComponent, StudentDetails, StudetntMarks, and so on. All will be able to use this service by injecting it through dependency injection.



In the above diagram, we are registering the service to the StudentDetails component, then the child components and itself StudentDetails Component will be able to use this service and other Components like StudentMarks component will not be able to use this service.

In simple words, the component which is having the service registered will be able to use the service and its child component will be able to use the service.

So, we have to take care of the creation and registration of the service while developing the application. It is always good to register the service to the AppModule so that it will not create a problem in injecting it to any component and to register it to the module. It will use the providermetadata.

In the app.module.ts, we have registered it as below.

First, we imported the service.

1. **import** { StudentService } from './student.service';

Then, we registered it.

*providers: [StudentService]*

The two above statements will already be done by Angular CLI. So, now we are done with the registration of the service. Now, we are going to mention the service to the component that needs to utilize the service.

In the student-details.component.ts, we are going to utilize this service by using the syntax of dependency injection to the constructor. Open student-details.component.ts and see the below contents.

1. **import** { Component, OnInit } from '@angular/core';
2. **import** { StudentService } from '../student.service';
4. @Component({
5. selector: 'app-student-details',
6. templateUrl: './student-details.component.html',
7. styleUrls: ['./student-details.component.css']
8. })
9. **export** **class** StudentDetailsComponent **implements** OnInit {
11. **public** students = [];
12. constructor(**private** studentService : StudentService) {
13. }
15. ngOnInit() {
16. **this**.students = **this**.studentService.getStudents();
17. }
18. }

We have injected the service called dependency injection also, as below in the constructor.

1. constructor(**private** studentService : StudentService) {
2. }

Also, don’t forget to import the service.

1. **import** { StudentService } from '../student.service';

The ‘studentService’ object will have the reference to the StudentService. It is better to utilize the fetching of data inside ngOnInit() method of the component because ngOnInit() method is called once the component has been initialized.

**Use of Injectable decorator in the component**

Open student.service.ts.

1. **import** { Injectable } from '@angular/core';
3. @Injectable({
4. providedIn: 'root'
5. })
6. **export** **class** StudentService {
7. students = [
8. {"id" : 1001, "name" : "Irshad", "marks" : 90},
9. {"id" : 1002, "name" : "Imran", "marks" : 80},
10. {"id" : 1003, "name" : "Rahul", "marks" : 70},
11. {"id" : 1004, "name" : "Ajay", "marks" : 85},
12. {"id" : 1005, "name" : "Sunny", "marks" : 60}
13. ];
14. constructor() { }
15. getStudents(){
16. **return** **this**.students;
17. }
18. }
19. @Injectable defines that this service might also inject some dependencies. So, if you want the service to be injectable to another service, then you must define the Injectable decorator. Don’t include this decorator if you are sure this will not inject any other service. It is recommended to add the @Injectable decorator as soon as we create the service class. Angular CLI already adds this decorator when creating the service.
20. The component class has a @Component decorator that lets us know if it may have dependency injection systems. But for service, there is no @Component decorator. If we remove the @Injectable decorator, then it will become a plane class and there is nothing to do with the service. So, to make the service, the @Injectable decorator is required.

# **Introduction To Routing**

Routing is a process of changing the state of your Application by loading different components depending upon the URL that the user enters. Angular 2 parses the entered URL by the user and try to identify the routes according the different segments of URL. In this article, we learn how routing works in Angular2. We will create an app, where we have two sections, Home and Student. When we click on Student section; a list of the students will open further, when we click on a student name; we display all the details of that student. Let’s start today’s section.

We will create three components in this Application. Home will be the home page for our student dashboard, student-list component will contain the list of all the students and last one is student component that will display the information of a particular student.

Thus, create three components “home”, “student” and “student-list”.  
  


To perform the routing in our Application, we need to define the routes. First, we create a route file in “app” folder that will contain the routes of our Application. Create a “app.routes.ts” file in “app” folder and insert the code given below into “app.routes.ts” file.

1. **import** {RouterModule,Routes} from '@angular/router';
2. **import** {HomeComponent} from './home/home.component';
3. **import** {StudentListComponent} from './student-list/student-list.component';
4. **import** {StudentComponent} from './student/student.component';
5. **const** APP\_ROUTES:Routes=[
6. {path:'',component:HomeComponent},
7. {path:'student',component:StudentListComponent},
8. {path:'student/:id',component:StudentComponent}
9. ];
11. **export** **const** Routes\_Provider=RouterModule.forRoot(APP\_ROUTES);

In the code given above, we import home, student and student-list components that we created earlier. Here, we create a “APP\_ROUTES” constant of routes type. This constant contains all the routes, which we require. Path property contains the routes that we want to match and component property defines the component that will be called when the path matches the routes value. In the next line, we define another constant variable. This variable contains the result generated from “RouterModule.forRoot” method. RouterModule.forRoot method creates a module that contains all the Directives, the given routes and the router Service itself.

After creating all the required routes, we need to configure these routes at the root level of the Application. Now, go to “app.modules.ts” file and import the routes file that we created earlier.

**App.Modules.ts**

1. **import** { BrowserModule } from '@angular/platform-browser';
2. **import** { NgModule } from '@angular/core';
3. **import** { FormsModule } from '@angular/forms';
4. **import** { HttpModule } from '@angular/http';
6. **import** { AppComponent } from './app.component';
7. **import** { HomeComponent } from './home/home.component';
8. **import** { StudentComponent } from './student/student.component';
9. **import** { StudentListComponent } from './student-list/student-list.component';
10. **import** {Routes\_Provider} from 'app/app.routes';
12. @NgModule({
13. declarations: [
14. AppComponent,
15. HomeComponent,
16. StudentComponent,
17. StudentListComponent,
18. HomeComponent
19. ],
20. imports: [
21. BrowserModule,
22. FormsModule,
23. HttpModule,
24. Routes\_Provider
25. ],
26. providers: [],
27. bootstrap: [AppComponent]
28. })
29. **export** **class** AppModule { }

Angular is used to create the “SPA” Application, so we follow this convention in our Application. We require a “home” button. Using it, we can navigate to home section of the Application and “studentList” button, using which we can navigate to studentList section of the Application. When we click on any student name, then student section will display the information of that particular student. We will create a common header, which will display on all the sections of our Application.

Now, write the code given below in “app.component.html” page.

1. <h1>
2. {{title}}
3. </h1>
4. <ul>
5. <!--<li><a href="/">Home</a></li>
6. <li><a href="/student">Student List</a></li>-->
7. <li><a [routerLink]="['']">Home</a></li>
8. <li><a [routerLink]="['student']">Student List</a></li>
9. </ul>
10. <router-outlet></router-outlet>

In the code given above, we create two anchor tags for Home and Student List. Instead of “href” attribute of anchor tag, we use the “routerLink” attribute of Angular. The routerLink attribute allows us to link to a specific route of the Application. In the last line of this page, we use the “router-outlet” Angular Directive. This Directive acts as a placeholder that is dynamically filled by Angular and is based on the current router state. This is the section, where we display the content of the components, which are based on the route value.

Now, our setup is ready. Let’s move to our component sections.

**Home.component.html**

1. <p>
2. This is Home Page,<br/> click On Student List **for** Student Information.
3. </p>

In home component, we are displaying only some text information. Let’s move to our student-list component.

**Student-list.component.html**

1. <ul>
2. <li \*ngFor="let data of studentList">
3. <a [routerLink]="['/student/',data.id]"> {{data.name}}</a>
4. </li>
5. </ul>

**Student-list.component.ts**

1. **import** { Component, OnInit } from '@angular/core';
3. @Component({
4. selector: 'app-student-list',
5. templateUrl: './student-list.component.html',
6. styleUrls: ['./student-list.component.css']
7. })
9. **export** **class** StudentListComponent **implements** OnInit {
10. **public** studentList:student[]=[];
12. constructor() {
14. **this**.studentList=[
15. **new** student(100,"Pankaj Choudhary"),
16. **new** student(101,"Sandeep Jangid"),
17. **new** student(102,"Rahul Prajapat"),
18. **new** student(103,"Sanjeev Baldia")];
19. }
21. ngOnInit() {
22. }
24. }
26. **class** student{
27. id:number;
28. name:string;
29. constructor(id:number,name:string){
30. **this**.id=id;
31. **this**.name=name;
32. }
33. }

In student-list component, we will create a “studentList” array object of student type. In constructor function, we will insert some information of the student in this array object. In the template section of this component, we use the “ngFor” directives to create a list of the students and also create an anchor tag for each value of the “studentList” list. In “routerLink” Directive, we define two segments “/student/” and data.id, which are similar to “/student/data.id” of an anchor tag and match the ‘student/:id’ route of our route file. For example, when we click for “Sanjeev Baldia” student “localhost:4200/sudent/103” URL will generate when 103 is the Id of this student. Now, we move to student component, which is the most important segment of this Application.

**Student.component.ts**

1. **import** { Component, OnInit } from '@angular/core';
2. **import** { ActivatedRoute } from '@angular/router';
3. @Component({
4. selector: 'app-student',
5. templateUrl: './student.component.html',
6. styleUrls: ['./student.component.css']
7. })
8. **export** **class** StudentComponent **implements** OnInit {
9. **public** studentList:student[]=[
10. **new** student(100,"Pankaj Choudhary","P-20 Gandhi Nagar Alwar","Software Developer"),
11. **new** student(101,"Sandeep Jangid","G14 Vasant Vihar , Jaipur","Electrical Enginner"),
12. **new** student(102,"Rahul Prajapat","K1-104, CR Park Delhi","DBA"),
13. **new** student(103,"Sanjeev Baldia","D-19, Malviya Nagar, Jaipur","Sales Head")];
15. **public** studentInfo:student;
16. **public** studentId:number;
17. constructor(**private** route: ActivatedRoute)
18. {
19. **this**.route.params.subscribe(params=>{
20. **this**.studentId=params['id'];
21. });
23. **this**.studentInfo=**this**.studentList.find(x=>x.id==**this**.studentId);
24. }
26. ngOnInit() {
27. }
29. }
30. **class** student{
31. id:number;
32. name:string;
33. address:string;
34. jobProfile:string;
36. constructor(id:number,name:string,address:string,jobProfile:string){
37. **this**.id=id;
38. **this**.name=name;
39. **this**.address=address;
40. **this**.jobProfile=jobProfile;
41. }
42. }

In the code given above, we create a “studentList” array object of student class type. This object contains same students information, which we created in “student-list” component but with two extra properties “address” and “jobProfile”. We import the “ActivatedRoute” interface in this component. ActivatedRoute contains the information about a route associated with a component loaded in an outlet. An ActivatedRoute can also be used to traverse the router state tree.

1. constructor(**private** route: ActivatedRoute)
2. {
3. **this**.route.params.subscribe(params=>{
4. **this**.studentId=params['id'];
5. });

In the line of code given above, we define route property of ActivatedRoute type. ActivatedRoute comes with a params property i.e. Observable. If you are not aware of Observable and Promises, then don’t worry, as I will cover these topics in my upcoming articles. Observable is like a stream and allows us to pass the events, where the callback is called for each event. Subscribe is the function that is called when the Observable is initially subscribed to. This function is given a Subscriber to which new values can be `next`ed, or an `error` method can be called to raise an error or `complete` can be called to notify of a successful completion. On the completion, we are getting the value of “id” parameter, which we passed from the student-list component.

1. **this**.studentInfo=**this**.studentList.find(x=>x.id==**this**.studentId);

In the lines of code, we are getting the student information, whose id is similar to the id, which we fetched out from the URL and insert into “studentInfo” object.

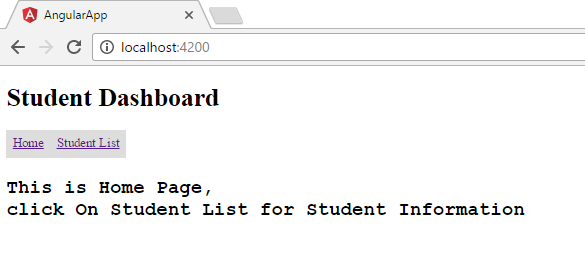
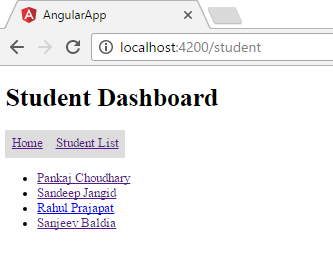
**Student.component.html**

1. <h2>
2. Student Information
3. </h2>
4. <div>
5. <table>
6. <tr>
7. <td>Student Name : </td>
9. <td>{{studentInfo.name}}</td>
11. </tr>
12. <tr>
13. <td>Address : </td>
15. <td>{{studentInfo.address}}</td>
16. </tr>
17. <tr>
18. <td>Job Profile : </td>
20. <td>{{studentInfo.jobProfile}}</td>
21. </tr>
22. </table>
23. </div>

**Student.component.css**

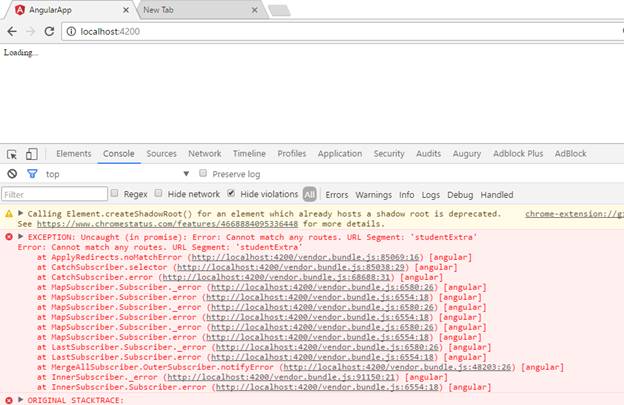
1. table {
2. border-collapse: collapse;
3. width: 50%;
4. }
6. th, td {
7. text-align: left;
8. padding: 8px;
9. }
11. tr:nth-child(even){background-color: #f2f2f2}
12. th, td {
13. border: 1px solid #ddd;
14. }

In the lines of code given above, we are displaying the information of a student and apply some CSS. Now, our setup is ready. Let’s run the Application.

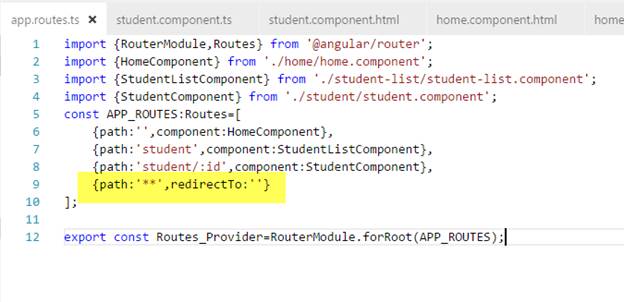
  
This is the home screen of our Application. When we click on “Student List” menu item, then this screen will be displayed.  
  
  
In this screen, we can see the list of all the students. Afterwards, click on any student name and we can get the information of that particular student.  
  
  
We cover the basics of Angular 2 routing. Now, let’s consider some other points in Angular 2 routing:

**Double Star(\*\*) Path**

What if any user inserts a URL that doesn’t match any route defined in route file. In such case, we will get the error, as shown below.

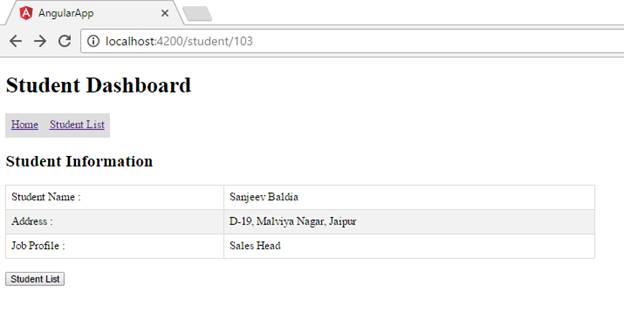


To prevent this error, we need to define the default route in our route file, as shown below.

  
The “\*\*” must be the last route in the route list. The router selects this route automatically, if the requested URL doesn’t match any routes. We can use the routes to display our home or “404- Not found” page.

**Triggered Routing from back-end**

Till now, we triggered the routing, using the anchor tag. Now, learn how to trigger the routing from back-end code. In “student.component.html” page, add a button and on click, we will be redirected from the student details to student list page.

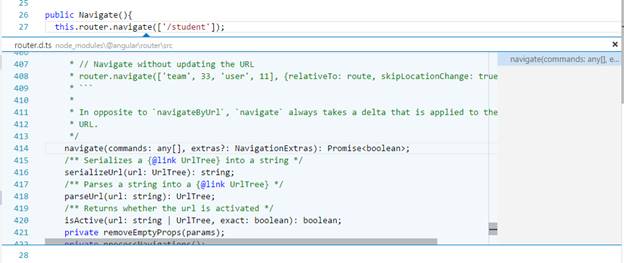


1. <h2>
2. Student Information
3. </h2>
4. <div>
5. <table>
6. <tr>
7. <td>Student Name : </td>
9. <td>{{studentInfo.name}}</td>
11. </tr>
12. <tr>
13. <td>Address : </td>
15. <td>{{studentInfo.address}}</td>
16. </tr>
17. <tr>
18. <td>Job Profile : </td>
20. <td>{{studentInfo.jobProfile}}</td>
21. </tr>
22. </table>
23. </div>
24. <br/>
25. <input type="button" value="Student List" (click)=Navigate() />

In the lines of code given above, we add a “Student List” button and on click event, we are calling “Navigate()” function.

**Student.component.ts**

1. **import** { Component, OnInit } from '@angular/core';
2. **import** { ActivatedRoute,Router } from '@angular/router';
3. @Component({
4. selector: 'app-student',
5. templateUrl: './student.component.html',
6. styleUrls: ['./student.component.css']
7. })
8. **export** **class** StudentComponent **implements** OnInit {
9. **public** studentList:student[]=[
10. **new** student(100,"Pankaj Choudhary","P-20 Gandhi Nagar Alwar","Software Developer"),
11. **new** student(101,"Sandeep Jangid","G14 Vasant Vihar , Jaipur","Electrical Enginner"),
12. **new** student(102,"Rahul Prajapat","K1-104, CR Park Delhi","DBA"),
13. **new** student(103,"Sanjeev Baldia","D-19, Malviya Nagar, Jaipur","Sales Head")];
15. **public** studentInfo:student;
16. **public** studentId:number;
17. constructor(**private** route: ActivatedRoute, **private** router:Router)
18. {
19. **this**.route.params.subscribe(params=>{
20. **this**.studentId=params['id'];
21. });
23. **this**.studentInfo=**this**.studentList.find(x=>x.id==**this**.studentId);
24. }
26. **public** Navigate(){
27. **this**.router.navigate(['/student']);
28. }
30. ngOnInit() {
31. }
33. }
34. **class** student{
35. id:number;
36. name:string;
37. address:string;
38. jobProfile:string;
40. constructor(id:number,name:string,address:string,jobProfile:string){
41. **this**.id=id;
42. **this**.name=name;
43. **this**.address=address;
44. **this**.jobProfile=jobProfile;
45. }
46. }



In the code given above, we import the “Router” class and using the Navigate method of this class, we can update the URL without refreshing the page. Hence, when we click on “Student List” button, we will move to the Student list segment of the Application.