* **Key differences between Angular 1.x & Angular 2**

**Angular 2**

1. Angular 2 is not an upgrade of Angular 1 but it is completely rewritten.
2. Angular 2 uses TypeScript which is a superset of JavaScript (It doesn’t mean only TypeScript but dart also).
3. Angular 1.x was not built with mobile support in mind, where Angular 2 is mobile-oriented.
4. Angular 1's core concept was $scope but you will not find $scope in Angular 2.0 and above.
5. Angular 1.x Controllers are gone in Angular v2. We can say that Controllers are replaced with “Components” in Angular 2.
6. In Angular 2, Structural directives syntax is changed. ng-repeat is replaced with \*ngFor.
7. In Angular 2, local variables are defined using hash(#) prefix.
8. Two-way data binding: ng-model has been replaced with [(ngModel)]
9. Angular 2 uses Hierarchical Dependency Injection system which is the major performance booster of it.
10. Angular 2 implements unidirectional-tree based change detection which, again, increases the performance.
11. If you compare the file size, Angular 2 is 20 kb less than Angular 1 which helps in decreasing the load time for apps.
12. Angular 2 provides more choice for languages. You can use any of the languages from ES5, ES6, TypeScript or Dart to write Angular 2 code while Angular 1.x had ES5, ES6, and Dart only. Addition of TypeScript is a great step as TypeScript is an awesome way to write JavaScript.
13. To filter output in our templates in Angular 1.x, we used the pipe character (|) and one or more filters. Now, in Angular 2, they are called pipes. The syntax remains same.
14. Angular 2 uses camelCase syntax for built-in directives. For example, ng-class is now ngClass and ng-model is now ngModel.
15. One of the biggest advantages of Angular is Dependency Injection. In Angular 2, DI is there but now there is a different way to inject dependencies. As everything is a class in Angular, so DI is achieved via a constructor.
16. In Angular 1.x, we can define a service via 5 different ways.  
    1. Factory
    2. Service
    3. Provider
    4. Constant
    5. Values

And, in Angular 2, class is the only way to define a service.

* **Features of Angular 2**

Following are the key features of Angular 2 −

* **Components** − The earlier version of Angular had a focus of Controllers but now has changed the focus to having components over controllers. Components help to build the applications into many modules. This helps in better maintaining the application over a period of time.
* **TypeScript** − The newer version of Angular is based on TypeScript. This is a superset of JavaScript and is maintained by Microsoft.
* **Services** − Services are a set of code that can be shared by different components of an application. So for example if you had a data component that picked data from a database, you could have it as a shared service that could be used across multiple applications.

In addition, Angular 2 has better event-handling capabilities, powerful templates, and better support for mobile devices.

* **Components of Angular 2**

Angular 2 has the following components −

* **Modules** − This is used to break up the application into logical pieces of code. Each piece of code or module is designed to perform a single task.
* **Component** − This can be used to bring the modules together.
* **Templates** − This is used to define the views of an Angular JS application.
* **Metadata** − This can be used to add more data to an Angular JS class.
* **Service** − This is used to create components which can be shared across the entire application.
* **Environment Setup**:

To start working with Angular 2, you need to get the following key components installed.

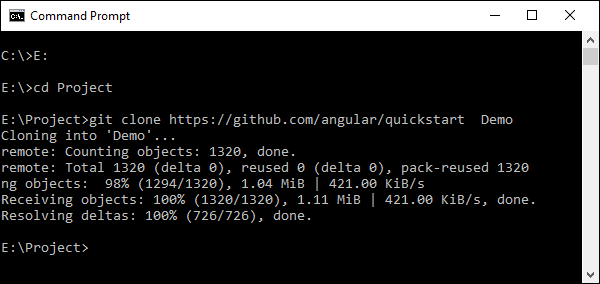
* **Npm** − This is known as the node package manager that is used to work with the open source repositories. Angular JS as a framework has dependencies on other components. And **npm** can be used to download these dependencies and attach them to your project. <https://www.nodejs.org/>
* **Git** − This is the source code software that can be used to get the sample application from the **GitHub** angular site.  <https://git-scm.com/>
* **Editor** − There are many editors that can be used for Angular JS development such as Visual Studio code and WebStorm. <https://code.visualstudio.com/>
* **Hello World Application**

**Step 1 :** Go the github url - <https://github.com/angular/quickstart>

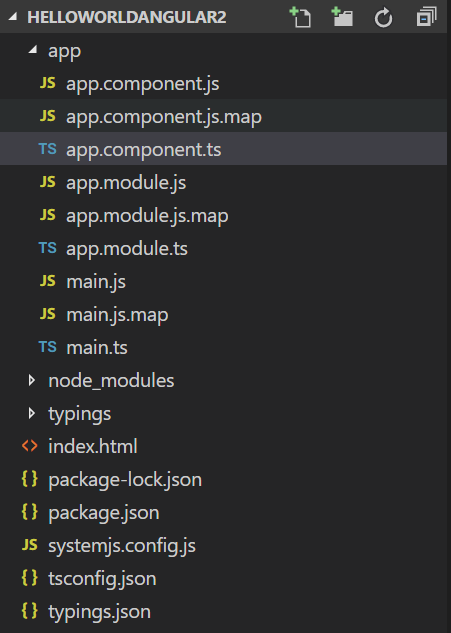
**Step 2** : Go to your command prompt, create a project directory. This can be an empty directory. In our example, we have created a directory called Project.

**Step 3** : Next, in the command prompt, go to this directory and issue the following command to clone the github repository on your local system. You can do this by issuing the following command –

git clone https://github.com/angular/quickstart Demo



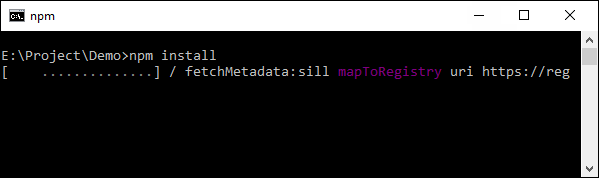
**Step 4** − Open the code in Visual Studio code.



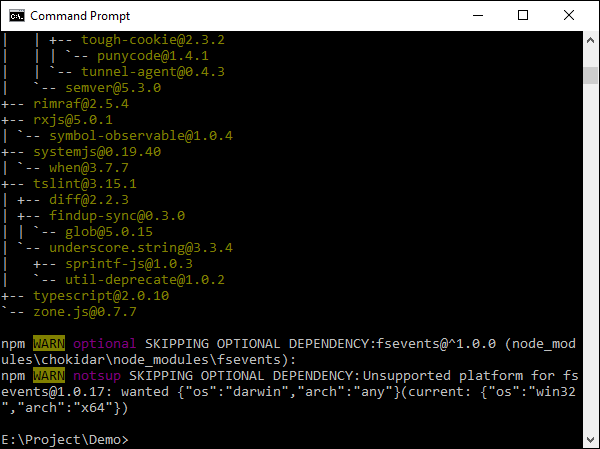
**Step 5** − Go to the command prompt and in your project folder again and issue the following command −

npm install

This will install all the necessary packages which are required for the Angular JS application to work.

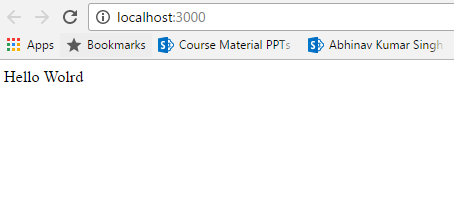


Once done, you should see a tree structure with all dependencies installed.



**Step 6 :** Now we will start the server

npm start - Angular 2



* 1. **tsconfig.json** : Typescript compiler configuration file.

It guides the compiler to generate JavaScript files

{

"compilerOptions": {

"target": "es5",

"module": "commonjs",

"moduleResolution": "node",

"sourceMap": true,

"emitDecoratorMetadata": true,

"experimentalDecorators": true,

"removeComments": false,

"noImplicitAny": false

},

"compileOnSave": false,

"buildOnSave": false

}

* 1. **typings.json :** In this file we add the famous JavaScript libraries, so that we can use their features in our project.

{

"globalDependencies": {

"core-js": "registry:dt/core-js#0.0.0+20160725163759",

"jasmine": "registry:dt/jasmine#2.2.0+20160621224255",

"node": "registry:dt/node#6.0.0+20160831021119"

}

}

* 1. **package. json:**
* package. json file contains all the project dependencies.
* Using reference of these packages in dependency section allows you to use module bundler like webpack, browserify etc.
* Installing these packages again on lets say another machine requires only package. json and dependency section in it.

{

"name": "primeng-quickstart",

"version": "v1.0.0-SNAPSHOT",

"scripts": {

"postinstall": "npm run typings install",

"tsc": "tsc",

"tsc:w": "tsc -w",

"lite": "lite-server",

"start": "concurrently \"npm run tsc:w\" \"npm run lite\" ",

"typings": "typings"

},

"license": "ISC",

"dependencies": {

"@angular/common": "2.0.0-rc.5",

"@angular/compiler": "2.0.0-rc.5",

"@angular/core": "2.0.0-rc.5",

"@angular/forms": "0.3.0",

"@angular/http": "2.0.0-rc.5",

"@angular/platform-browser": "2.0.0-rc.5",

"@angular/platform-browser-dynamic": "2.0.0-rc.5",

"@angular/router": "3.0.0-rc.1",

"@angular/upgrade": "2.0.0-rc.5",

"angular2-in-memory-web-api": "0.0.15",

"core-js": "^2.4.0",

"primeng": "^1.0.0-beta.15",

"reflect-metadata": "^0.1.3",

"rxjs": "5.0.0-beta.6",

"systemjs": "0.19.27",

"zone.js": "^0.6.12"

},

"devDependencies": {

"concurrently": "^2.0.0",

"lite-server": "^2.2.0",

"typescript": "^1.8.10",

"typings": "^0.8.1"

}

}

* 1. **systemjs.config.js**
* It helps you to configure *SystemJS*to load modules compiled using the *Typescript compiler*.
* For anonymous modules, It allows to map the name of modules to JS files that actually contains the module JavaScript code.

/\*\*

\* System configuration for Angular 2 samples

\* Adjust as necessary for your application needs.

\*/

(function(global) {

// map tells the System loader where to look for things

var map = {

'app': 'app', // 'dist',

'@angular': 'node\_modules/@angular',

'angular2-in-memory-web-api': 'node\_modules/angular2-in-memory-webapi',

'rxjs': 'node\_modules/rxjs',

'primeng': 'node\_modules/primeng'

};

// packages tells the System loader how to load when no filename and/or no extension

var packages = {

'app': { main: 'main.js', defaultExtension: 'js' },

'rxjs': { defaultExtension: 'js' },

'angular2-in-memory-web-api': { main: 'index.js', defaultExtension: 'js' },

'primeng': { defaultExtension: 'js' }

};

var ngPackageNames = [

'common',

'compiler',

'core',

'forms',

'http',

'platform-browser',

'platform-browser-dynamic',

'router',

'router-deprecated',

'upgrade',

];

// Individual files (~300 requests):

function packIndex(pkgName) {

packages['@angular/'+pkgName] = { main: 'index.js', defaultExtension:'js' };

}

// Bundled (~40 requests):

function packUmd(pkgName) {

packages['@angular/'+pkgName] = { main: 'bundles/' + pkgName + '.umd.js', defaultExtension: 'js' };

}

// Most environments should use UMD; some (Karma) need the individual index files

var setPackageConfig = System.packageWithIndex ? packIndex : packUmd;

// Add package entries for angular packages

ngPackageNames.forEach(setPackageConfig);

var config = {

map: map,

packages: packages

};

System.config(config);

})(this);

* 1. **app.components.ts :** This is our root component.

**Components:** Angular applications are made up of components. A component is the combination of an HTML template and a component class that controls a portion of the screen.

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

@Component({

template: `<h1>Hello {{name}}</h1>`,

selector: 'my-app'

})

export class AppComponent { name = 'Angular'; }

* 1. **app.module.ts:** The root module that tells Angular how to assemble the application.

import {NgModule, Inject, EventEmitter} from '@angular/core';

import { bootstrap } from '@angular/platform-browser-dynamic';

import {AppComponent} from './app.component';

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

import {HttpModule, HTTP\_PROVIDERS} from '@angular/http';

import {BrowserModule} from '@angular/platform-browser';

import {ROUTER\_DIRECTIVES, RouterModule} from '@angular/router';

import 'rxjs/add/operator/toPromise';

import { MultiSelectModule,

TooltipModule,

RadioButtonModule,

GrowlModule,

CalendarModule,

InputTextModule,

InputTextareaModule,

DataTableModule,

ButtonModule,

DialogModule,

DropdownModule,

PanelModule,

InputMaskModule,

PasswordModule,

CheckboxModule,

MessagesModule,

MenubarModule,

DataGridModule,

TreeModule } from 'primeng/primeng';

@NgModule({

imports: [MultiSelectModule,

TooltipModule,

RadioButtonModule,

ReactiveFormsModule,

GrowlModule,

BrowserModule,

CalendarModule,

InputTextModule,

FormsModule,

HttpModule,

DataTableModule,

ButtonModule,

DialogModule,

DropdownModule,

PanelModule,

InputMaskModule,

PasswordModule,

CheckboxModule,

DataGridModule,

MessagesModule,

TreeModule,

MenubarModule

],

declarations: [AppComponent],

bootstrap: [AppComponent],

directives: [ROUTER\_DIRECTIVES]

})

export class AppModule { }

* 1. **main.ts :** Tells Angular to load the component.

import { platformBrowserDynamic } from '@angular/platform-browserdynamic';

import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);

* 1. **index.html :**

<!DOCTYPE html>

<html>

<head>

<base href="/">

<title>PrimeNG QuickStart</title>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="stylesheet" type="text/css" href="node\_modules/primeng/resources/themes/omega/theme.css" />

<link rel="stylesheet" type="text/css" href="app/resources/icons/css/font-awesome.min.css" />

<link rel="stylesheet" type="text/css" href="node\_modules/primeng/resources/primeng.min.css" />

<link rel="stylesheet" type="text/css" href="node\_modules/primeui/themes/omega/theme.css" />

<link rel="stylesheet" type="text/css" href="node\_modules/primeui/primeui-ng-all.min.css" />

<link rel="stylesheet" type="text/css" href="app/resources/css/jpafWebAngular.css" />

<!-- 1. Load libraries -->

<!-- Polyfill(s) for older browsers -->

<script src="node\_modules/core-js/client/shim.min.js"></script>

<script src="node\_modules/zone.js/dist/zone.js"></script>

<script src="node\_modules/reflect-metadata/Reflect.js"></script>

<script src="node\_modules/systemjs/dist/system.src.js"></script>

<!-- Optional if a jquery dependant widget is used -->

<script src="node\_modules/primeui/primeui-ng-all.min.js"></script>

<script src="node\_modules/jquery.inputmask-3.x/dist/jquery.inputmask.bundle.js"></script>

<!-- 2. Configure SystemJS -->

<script src="systemjs.config.js"></script>

<script>

System.import('app').catch(function(err){ console.error(err); });

</script>

</head>

<!-- 3. Display the application -->

<body>

<my-app>Loading...</my-app>

</body>

</html>

* **Angular2 Modules :**

Modules are used in Angular JS to put logical boundaries in your application. Hence, instead of coding everything into one application, you can instead build everything into separate modules to separate the functionality of your application.

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

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

@NgModule ({

imports: [ BrowserModule ],

declarations: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

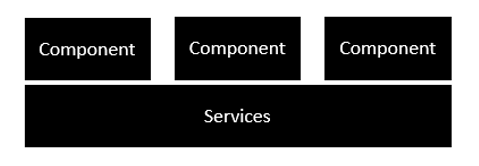
Let’s go through each line of the code in detail.

* The import statement is used to import functionality from the existing modules. Thus, the first 3 statements are used to import the NgModule, BrowserModule and AppComponent modules into this module.
* The NgModule decorator is used to later on define the imports, declarations, and bootstrapping options.
* The BrowserModule is required by default for any web based angular application.
* The bootstrap option tells Angular which Component to bootstrap in the application.

A module is made up of the following parts −

* **Bootstrap array** − This is used to tell Angular JS which components need to be loaded so that its functionality can be accessed in the application. Once you include the component in the bootstrap array, you need to declare them so that they can be used across other components in the Angular JS application.
* **Export array** − This is used to export components, directives, and pipes which can then be used in other modules.
* **Import array** − Just like the export array, the import array can be used to import the functionality from other Angular JS modules.
* **Angular2 Architecture :**

The following screenshot shows the anatomy of an Angular 2 application. Each application consists of Components. Each component is a logical boundary of functionality for the application. You need to have layered services, which are used to share the functionality across components.

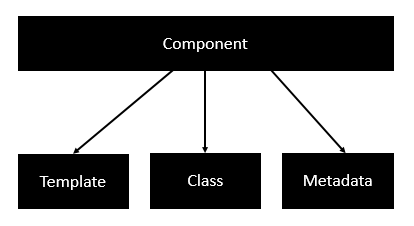


Following is the anatomy of a Component. A component consists of −

**Class −** This is like a C++ or Java class which consists of properties and methods.

**Metadata −** This is used to decorate the class and extend the functionality of the class.

**Template −** This is used to define the HTML view which is displayed in the application.



Following is an example of a component.

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

@Component ({

selector: 'my-app',

templateUrl: 'app/app.component.html'

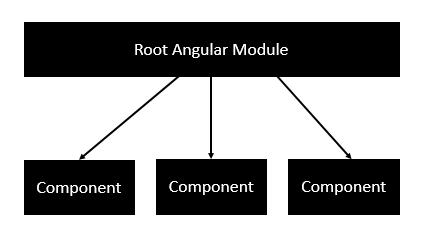
})

export class AppComponent {

appTitle: string = 'Welcome';

}

Each application is made up of modules. Each Angular 2 application needs to have one Angular Root Module. Each Angular Root module can then have multiple components to separate the functionality.



Following is an example of a root module.

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

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

@NgModule ({

imports: [ BrowserModule ],

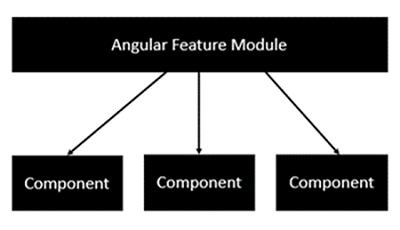
declarations: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

Each application is made up of feature modules where each module has a separate feature of the application. Each Angular feature module can then have multiple components to separate the functionality.



* **Angular 2 - Components**

Components are a logical piece of code for Angular JS application. A Component consists of the following −

* **Template** − This is used to render the view for the application. This contains the HTML that needs to be rendered in the application. This part also includes the binding and directives.
* **Class** − This is like a class defined in any language such as C. This contains properties and methods. This has the code which is used to support the view. It is defined in TypeScript.
* **Metadata** − This has the extra data defined for the Angular class. It is defined with a decorator.

## Class

The class decorator. The class is defined in TypeScript. The class normally has the following syntax in TypeScript.

### **Syntax**

class classname {

Propertyname: PropertyType = Value

}

### **Parameters**

* **Classname** − This is the name to be given to the class.
* **Propertyname** − This is the name to be given to the property.
* **PropertyType** − Since TypeScript is strongly typed, you need to give a type to the property.
* **Value** − This is the value to be given to the property.

### **Example**

export class AppComponent {

appTitle: string = 'Welcome';

}

In the example, the following things need to be noted −

* We are defining a class called AppComponent.
* The export keyword is used so that the component can be used in other modules in the Angular JS application.
* appTitle is the name of the property.
* The property is given the type of string.
* The property is given a value of ‘Welcome’.

## Template

This is the view which needs to be rendered in the application.

### **Syntax**

Template: '

<HTML code>

class properties'

### **Parameters**

* **HTML Code** − This is the HTML code which needs to be rendered in the application.
* **Class properties** − These are the properties of the class which can be referenced in the template.

### **Example**

template: '

<div>

<h1>{{appTitle}}</h1>

<div>To Tutorials Point</div>

</div>

'

In the example, the following things need to be noted −

* We are defining the HTML code which will be rendered in our application
* We are also referencing the appTitle property from our class.

### **Metadata**

This is used to decorate Angular JS class with additional information.

### **Example**

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

@Component ({

selector: 'my-app',

template: ` <div>

<h1>{{appTitle}}</h1>

<div>To Tutorials Point</div>

</div> `,

})

export class AppComponent {

appTitle: string = 'Welcome';

}

In the above example, the following things need to be noted −

* We are using the import keyword to import the ‘Component’ decorator from the angular/core module.
* We are then using the decorator to define a component.
* The component has a selector called ‘my-app’. This is nothing but our custom html tag which can be used in our main html page.

Let’s make sure that the body tag now contains a reference to our custom tag in the component. Thus in the above case, we need to make sure that the body tag contains the following code −

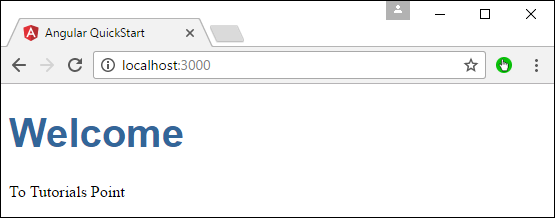
<body>

<my-app></my-app>

</body>

### **Output**

Now if we go to the browser and see the output, we will see that the output is rendered as it is in the component.



* **Angular 2 – Templates**

We can define a template using templateURL command.

## Syntax

templateURL:

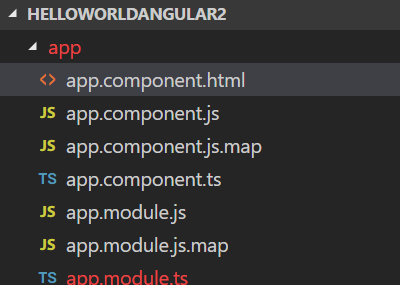
viewname.component.html

## Parameters

* **viewname** − This is the name of the app component module.

After the viewname, the component needs to be added to the file name

**Step 1 :** Create a file called app.component.html inside app folder. This will contain the html code for the view.



**Step 2:** Add below code inside app.component.html

<div>{{appTitle}} Tutorialspoint </div>

**Step 3 :** In the app.component.ts file, add the following code.

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

@Component ({

selector: 'my-app',

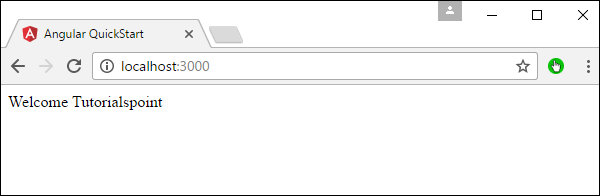
templateUrl: 'app/app.component.html'

})

export class AppComponent {

appTitle: string = 'Welcome';

}



## Data Binding

## Data Binding is a process where data is passed from Angular Component to view (Template) and vice versa. The Data Binding is used to bind DOM Elements to Component properties. Binding can be used display component class property values to the user, change element styles, respond to a user event etc.

1. Interpolation
2. Property Binding
3. Event Binding
4. Two Way Binding
   1. **Interpolation**

Inside app.component.html

<h1> {{title}} </h1>

Open the app.component.ts file and copy the following code

export class AppComponent {

title= 'Data Binding in Angular';

}

The title property, which is defined in the component class bound to the template using double curly braces in the template. This is called Interpolation.

*Interpolation provides the data-binding from component to the View. Interpolation is a one-way binding*. It binds from Component Class to the Template.

The Template expression (double curly braces ) used for interpolation in Angular. The Angular evaluates the Template expression and replaces it with the result.

In the above example, we got the data from the class Property called title.

You can use interpolation to invoke a method in the component, Concatenate two string, perform some mathematical operations or change the property of the DOM element like colour etc.

**Invoke a method in the component**

We can invoke the components methods using interpolation. Open the app.component.ts and the following function

app.component.ts

getTitle(): string {

return this.title;

}

app.component.html

<p>{{getTitle()}}</p>

* 1. **Property Binding**

Property binding allows us to bind Property of a view element to the value of template expression

**Syntax :**

[Property]=”expression”

The Property Binding uses the [] brackets. The Binding Target is placed inside the square brackets. The Binding source is enclosed in quotes.

Property binding is one way from Component to the Target in the template.

You can add these codes to the app.component.html and see the result

<p [innerText]="title"></p>

<p [innerText]="getTitle()"></p>

<p [innerText]="'Hello & Welcome to '+ ' Angular Data binding '"></p>

<p [innerText]="100\*80"></p>

<p [style.color]="color">This is red</p>

* 1. **Event Binding**
* Event Binding is one way from View to Component
* Binding source is a Template statement

app.component.html

<button (click)='buttonClicked()'>Click Me</button>

<p> Button Clicked {{count}} Times </p>

app.component.ts

count: number=0;

buttonClicked() : void {

this.count=this.count+1;

console.log("Button Clicked")

}

* 1. **Two Way Handling**

Two-way binding means that changes made in the component data are propagated to the view and that any changes made in the view are immediately updated in the underlying component data.

Two-way binding is used mainly in data entry forms. Whenever user makes changes in the data, we would like to update our model in the component with the new data and if the model changes, we would like to update the view as well.

The Angular uses the combination of Property binding (from component to view) and event binding (from view to component) to achieve the Two-way data binding. This is done so by using the ngModel directive.

import {Component} from 'angular2/core';

@Component({

selector: 'my-app',

template: `

Enter your Name : <input type='text' [(ngModel)]='name' />

You have entered {{name}}`

})

export class AppComponent {

name: string='';

}

## Angular Directive

## The Angular directive helps us to manipulate the DOM. You can change the appearance, behaviour or a layout of a DOM element using the Directives. They help you to extend HTML

## There are three kinds of directives in Angular:

## Component Directive

## Structural Directives

## Attribute Directives

## b) Structural Directives

## Structural directives can change the DOM layout by adding and removing DOM elements. All structural Directives are preceded by Asterix symbol

## ngFor:

## The ngFor is an Angular 2 structural directive, which repeats a portion of HTML template once per each item from an iterable list (Collection). The ngFor is similar to ngRepeat in AngularJS

## Example:

import {Component} from 'angular2/core';

@Component({

selector: 'courses',

template : `<h2>Courses</h2>

{{ title }}

<ol>

<li \*ngFor="#course of courses">

{{ course }}

</li>

</ol>`

})

export class CoursesComponent {

courses :string [] = ["Course1","Course2","Course3","Course4","Course5"];

}

## ngIf

The **ngif** element is used to add elements to the HTML code if it evaluates to true, else it will not add the elements to the HTML code.

### **Syntax**

\*ngIf = 'expression'

If the expression evaluates to true then the corresponding gets added, else the elements are not added.

**app.component.ts:**

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

@Component({

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

selector: 'my-app'

})

export class AppComponent {

appTitle: string = 'Welcome';

appStatus: boolean = true;

}

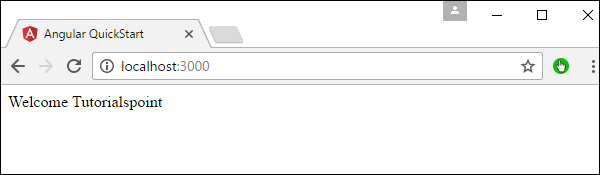
**app.component.html:**

<div \*ngIf = 'appStatus'>{{appTitle}} Tutorialspoint </div>

In the above code, we now have the \*ngIf directive. In the directive we are evaluating the value of the appStatus property. Since the value of the property should evaluate to true, it means the div tag should be displayed in the browser.

Once we add the above code, we will get the following output in the browser.

### **Output**



## ngSwitch :

## The ngSwitch directive lets you add/remove HTML elements depending on a match expression. ngSwitch directive used along with NgSwitchCase and NgSwitchDefault

#### **ngSwitchCase**

The inner elements are placed inside the container element. The ngSwitchCase directive is applied to the inner elements with a match expression.Whenever the value of the match expression matches the value of the switch expression , the corresponding inner element is added to the DOM. All other inner elements are removed from the DOM.If there is more than one match, then all the matching elements are added to the DOM.

***ngSwitchDefault***

You can also apply the ngSwitchDefault directive. The element with ngSwitchDefault is displayed only if no match is found. The inner element with ngSwitchDefault can be placed anywhere inside the container element and not necessarily at the bottom. If you add more than one ngSwitchDefault directive, all of them are displayed.  
Any elements placed inside the container element, but outside the ngSwitchCase or ngSwitchDefault elements are displayed as it is.

import {Component} from 'angular2/core';

@Component({

selector: 'courses',

template : `

Input string : <input type='text' [(ngModel)] ="num"/>

<div [ngSwitch]="num">

<div \*ngSwitchCase="1">One</div>

<div \*ngSwitchCase="2">Two</div>

<div \*ngSwitchCase="3">Three</div>

<div \*ngSwitchCase="4">Four</div>

<div \*ngSwitchCase="5">Five</div>

<div \*ngSwitchDefault>This is Default</div>

</div>

`

})

export class CoursesComponent {

num: number= 0;

}

## c) Attribute Directives:

## An Attribute or style directive can change the appearance or behaviour of an element.

## 1)ngModel : The ngModel directive is used the achieve the two-way data binding.

import {Component} from 'angular2/core';

@Component({

selector: 'my-app',

template: `

Enter your Name : <input type='text' [(ngModel)]='name' />

You have entered {{name}}`

})

export class AppComponent {

name: string='';

}

**2)ngClass :** The ngClass is used to add or remove the CSS classes from an HTML element.

ngClass attribute is applied to an element of DOM. It is then bound to an expression. The ngClass attribute evaluates the expression and changes the class attribute of the element to which it is attached.

There are three different ways you can use the expression. The expression can be evaluated either to a string, array or an object.

* + 1. **Binding to a String**

You can use the String as expression and bind it to directly to the ngClass attribute. If you want to assign multiple classes, then separate each class with space as shown below.

<element [ngClass]="'cssClass1 cssClass2'">...</element>

Example :

**Style.css**

.red {

color: red;

}

.size20 {

font-size: 20px;

}

**app.template.html**

<div [ngClass]="'red size20'">

Red Text with Size 20px

</div>

You can also use the ngClass without a square bracket. In that case, the expression is not evaluated but assigned directly to the class attribute. We also need to remove the double quote around the expression as shown below.

<div ngClass=’red size20'>

Red Text with Size 20px

</div>

* + 1. **Binding to an Array:**

You can achieve the same result by using an array instead of a string as shown below. The syntax for ngClass array syntax is as shown below.

<element [ngClass]="['cssClass1', 'cssClass2']">...</element>

<div [ngClass]="['red','size20']">

Red Text with Size 20px

</div>

* + 1. **Binding to an object :**

You can also bind the ngClass to an object. Each property name of the object acts as a class name and is applied to the element if it is true. The syntax is as shown below.

<element [ngClass]="{'cssClass1': true, 'cssClass2': true}">...</element>

<div class="row">

<div [ngClass]="{'red':true,'size20':true}">

Red Text with Size 20px : as object

</div>

</div>

**3)ngStyle :**

The ngStyle directive allows you to modify the style of an HTML element using the expression.

<element [ngStyle]="{'styleName': styleExp}">...</element>

The ngStyle Directive is applied to an element on DOM. The styleName is name of the style ( ex: font-size, color etc). styleExp is the expression, which is evaluated and assigned to the StyleName.

**Example:**

import {Component} from 'angular2/core';

@Component({

selector: 'my-app',

template: `<input [(ngModel)]="color" />

<div [ngStyle]="{'color': color}">

Change my color

</div>

`

})

export class AppComponent {

color: string= 'red';

}

## Angular Pipes

## Angular 2 Pipes helps us to format or transform data to display in our template. They are similar to Filters in AngularJS. List of angular 2 built-in pipes like currency pipe, date pipe, number pipe, percent pipe, decimal pipe, & slice pipe.

**Syntax :** Expression | pipeOperator[:pipeArguments]

Expression : is the expression, which you want to transform

| : is the Pipe Character

pipeOperator : name of the Pipe

pipeArguments: arguments to the Pipe

1. **DatePipe**

date\_expression | date[:format]

date\_expression is a date object or a number

date is the name of the pipe

format is the date and time format string which indicates the format in which date/time components are displayed.

Some of the common format strings are

|  |  |  |
| --- | --- | --- |
| **COMPONENT** | **FORMAT** | **EXAMPLE** |
| Year | y | 2016 |
| Year | yy | 16 |
| Month | M | 9 |
| Month | M | 99 |
| Month | MMM | Nov |
| Month | MMMM | November |
| Day | d | 9 |
| Day | dd | 9 |
| hour | j | 9 |
| hour | jj | 9 |
| hour | h | 9:00 AM |
| hour | hh | 9:00 AM |
| hour24 | H | 13 |
| hour24 | HH | 13 |
| minute | m | 9 |
| minute | mm | 9 |
| second | s | 9 |
| second | ss | 99 |
| Time zone | z | Pacific Standard time |
| Time zone | Z | GMT-8:00 |
| Time zone | a | PM |
| era | G | AD |
| era | GGGG | Anno Domini |

Format argument also supports some predefined commonly used formats

|  |  |  |
| --- | --- | --- |
| **FORMAT NAME** | **EQUIVALENT FORMAT STRNG** | **EXAMPLE** |
| **(FOR EN-US)** |
| medium | yMMMdjms | Sep 3, 2010, 12:05:08 PM |
| short | yMdjm | 9/3/2010, 12:05 PM |
| fullDate | yMMMMEEEEd | Friday, September 3, 2010 |
| longDate | yMMMMd | 3-Sep-10 |
| mediumDate | yMMMd | 3-Sep-10 |
| shortDate | yMd | 9/3/2010 |
| mediumTime | jms | 12:05:08 PM |
| shortTime | jm | 12:05 PM |

**Example :**

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

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

@Component({

selector: 'app-root',

template:`<p>medium : {{toDate | date:'medium'}} </p>

<p>short : {{toDate | date:'short'}} </p>

<p>fullDate : {{toDate | date:'fullDate'}} </p>

<p>longDate : {{toDate | date:'longDate'}} </p>

<p>mediumDate : {{toDate | date:'mediumDate'}} </p>

<p>shortDate : {{toDate | date:'shortDate'}} </p>

<p>mediumTime : {{toDate | date:'mediumTime'}} </p>

<p>dd-MM-y : {{toDate | date:'dd-MM-y'}} </p>

<p>dd-MM-yy HH:mm : {{toDate | date:'dd-MM-yy HH:mm'}} </p>`

})

export class AppComponent

{

title: string = 'Angular 2 pipes Example' ;

toDate: Date = new Date();

}

1. **UpperCasePipe & LowerCasePipe**

These pipes transform the string to Uppercase or lowercase.

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

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

@Component({

selector: 'app-root',

template:`<p>Unformatted :{{msg}} </p>

<p>Uppercase :{{msg | uppercase}} </p>

<p>Lowercase :{{msg | lowercase}} </p>`

})

export class AppComponent

{

title: string = 'Angular 2 pipes Example' ;

msg: string= 'Welcome to Angular 2';

}

1. **SlicePipe**

Creates a new List or String containing a subset (slice) of the string or array. This Pipe uses the JavaScript API Array.prototype.slice() and String.prototype.slice().

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

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

@Component({

selector: 'app-root',

template:`<p>Complete String :{{msg}} </p>

<p>Example 1 :{{msg | slice:11:20}} </p>

<p>Example 2 :{{msg | slice:-9}} </p>`

})

export class AppComponent

{

title: string = 'Angular 2 pipes Example' ;

msg: string= 'Welcome to Angular 2';

}

1. **DecimalPipe / NumberPipe**

The Decimal Pipe is used to Format a number as Text. This pipe will format the number according to locale rules.

number\_expression | number[:digitInfo]

Where

number\_expression is the number you want to format

number is the name of the pipe

digitInfo is a string which has the following format

{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}

Where

minIntegerDigits is the minimum number of integer digits to use. Defaults to 1.

minFractionDigits is the minimum number of digits after fraction. Defaults to 0.

maxFractionDigits is the maximum number of digits after fraction. Defaults to 3.

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

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

@Component({

selector: 'app-root',

template: `<p> Unformatted :{{num}}</p>

<p> Formatted :{{num | number}}</p>

<p> Formatted :{{num | number:'3.1-2'}}</p>

<p> Formatted :{{num | number:'7.1-5'}} </p>`

})

export class AppComponent

{

title: string = 'Angular 2 pipes Example' ;

num: number= 9542.14554;

}

## Adding Child/Nested Components Using Angular

## The Angular follows component based Architecture, where each component manages a specific task or workflow. Each component is an independent block of the reusable unit.

## In real life angular application, we need to break our application into a small child or nested components. Then the task of root components is to just host these child components. These child components, in turn, can host the more child components creating a Tree like structure called Component Tree.

## How to add Child Component

1. Create the Child Component. In the child Component, meta data specify the selector to be used
2. Import the Child Component in the module class and declare it in declaration Array
3. Use the CSS Selector to specify in the Parent Component Template, where you want to display the Child Component

## Creating the Child Component

## Customer.ts

export class Customer {

customerNo: number;

name:string ;

address:string;

city:string;

state:string;

country:string;

}

## customer-list.component.ts

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

import { Customer } from './customer';

@Component({

selector: 'customer-list',

templateUrl: 'app/customer-list.component.html'

})

export class CustomerListComponent

{

customers: Customer[] = [

{customerNo: 1, name: 'Rahuld Dravid', address: '', city: 'Banglaore', state: 'Karnataka', country: 'India'},

{customerNo: 2, name: 'Sachin Tendulkar', address: '', city: 'Mumbai', state: 'Maharastra', country: 'India'},

{customerNo: 3, name: 'Saurrav Ganguly', address: '', city: 'Kolkata', state: 'West Bengal', country: 'India'},

{customerNo: 4, name: 'Mahendra Singh Dhoni', address: '', city: 'Ranchi', state: 'Bihar', country: 'India'},

{customerNo: 5, name: 'Virat Kohli', address: '', city: 'Delhi', state: 'Delhi', country: 'India'},

]

}

## customer-list.component.html

## <div class='panel panel-primary'>

## <div class='panel-heading'>

## <p>List of Customers</p>

## </div>

## <div class='panel-body'>

## <div class='table-responsive'>

## <table class='table'>

## <thead>

## <tr>

## <th>No</th>

## <th>Name</th>

## <th>Address</th>

## <th>City</th>

## <th>State</th>

## </tr>

## </thead>

## <tbody>

## <tr \*ngFor="let customer of customers;">

## <td>{{customer.customerNo}}</td>

## <td>{{customer.name}}</td>

## <td>{{customer.address}}</td>

## <td>{{customer.city}}</td>

## <td>{{customer.state}}</td>

## </tr>

## </tbody>

## </table>

## </div>

## </div>

## </div>

## Import the Child Component in the Module

## app.module.ts

import { BrowserModule } from '@angular/platform-browser';

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

import {NgbModule} from '@ng-bootstrap/ng-bootstrap';

import { AppComponent } from './app.component';

import {CustomerListComponent} from './customer-list.component';

@NgModule({

declarations: [

AppComponent, CustomerListComponent

],

imports: [

BrowserModule,NgbModule.forRoot()

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

### **3. Tell angular where to display our component**

## Finally, we need to inform the Angular, where to display the child Component

## We want our child Component as the child of the App component. Open the app.component.html and add the following template.

## app.component.html

## <div>

## <h1>{{ title }}</h1>

## <customer-list></customer-list>

## </div>

## Angular Pass data to Child Component

## In Angular, the Parent Component can communicate with the child component by setting its Property. To do that the Child component must expose its properties to the parent component. The Child Component does this by using the @Input decorator

## In the Child Component

## Import the @Input module from @angular/Core Library

## Mark those property, which you need data from parent as input property using @Input decorator

## In the Parent Component

## Bind the Child component property in the Parent Component when instantiating the Child

## When you mark a property as input property, then the Angular injects values into the component property using Property Binding. The Property Binding uses the [] brackets. The Binding Target (Property of the child component) is placed inside the square brackets. The Binding source is enclosed in quotes. Property binding is one way from Component to the Target in the template

## Example:

## Create a Child Component and add it inside app.module.ts

## child.component.ts

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

## @Component({

## selector: 'child-component',

## template: `<h2>Child Component</h2>

## current count is {{ count }}`

## })

## export class ChildComponent {

## @Input() count: number;

## }

## Bind to Child Property in Parent Component

## app.component.ts

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

## @Component({

## selector: 'app-root',

## template: `

## <h1>Welcome to {{title}}!</h1>

## <button (click)="increment()">Increment</button>

## <button (click)="decrement()">decrement</button>

## <child-component [count]=Counter></child-component>` ,

## styleUrls: ['./app.component.css']

## })

## export class AppComponent {

## title = 'Component Interaction';

## Counter = 5;

## increment() {

## this.Counter++;

## }

## decrement() {

## this.Counter--;

## }

## }

## https://cdn.tektutorialshub.com/wp-content/uploads/2017/09/Passing-Data-to-Child-Component-in-Angular.png

## Detecting the Input changes

## Passing the data to child component is not sufficient, the child Component needs to know when the input changes so that it can act upon it.

## There are two ways of detecting when input changes in the child component in Angular

## Using OnChanges LifeCycle Hook

## Using Input Setter

## Using OnChanges LifeCycle Hook

## ngOnChanges is a lifecycle hook, which angular fires when it detects changes to data bound input property. This method receives a SimpeChanges object, which contains the current and previous property values. We can Intercept input property changes in the child component using this hook.

## How to use ngOnChanges for Change Detection

1. Import the OnChanges interface, SimpleChanges, SimpleChange from @angule/core library.
2. Implement the ngOnChanges() method. The method receives the SimpleChanges object containing the changes each input property.

## Example:

## Update child.component.ts as follows and open the console log to watch the logs as you click on increment and decrement buttons in parent component.

import { Component, Input, OnChanges, SimpleChanges, SimpleChange } from '@angular/core';

@**Component**({

selector: 'child-component',

template: `<h2>Child Component</h2>

current count is {{ count }}

`

})

export class **ChildComponent** implements **OnChanges** {

@**Input**() count: number;

**ngOnChanges**(changes: **SimpleChanges**) {

for (let property in changes) {

if (property === 'count') {

**console**.**log**('Previous:', changes[property].previousValue);

**console**.**log**('Current:', changes[property].currentValue);

*//console.log('firstChange:', changes[property].FirstChange);*

}

}

}

}

## Using Input Setter

## We can use the property getter and setter to detect the changes made to the input property as shown below

## Example: child.component.ts

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

@**Component**({

selector: 'child-component',

template: `<h2>Child Component</h2>

current count is {{ count }}

`

})

export class **ChildComponent** {

private \_count = 0;

@**Input**()

set **count**(count: number) {

this.\_count = count;

**console**.**log**(count);

}

get **count**(): number {

return this.\_count;

}

}

## Angular Component Life Cycle Hooks

## The life cycle hooks are the methods that angular invokes on directives and components as it creates, changes, and destroys them. Using life-cycle hooks we can fine-tune the behavior of our components during creation, update, and destruction.

## When the angular application starts it creates and renders the root component. It then creates and renders it’s Children. For each loaded component, it checks when its data bound properties change and updates them. It destroys the component and removes it from the DOM when no longer in use.

## Angular lets us know when these events happen using lifecycle hooks

## The Angular life cycle hooks are nothing but callback function, which angular invokes when a certain event occurs during the component’s life cycle.

## For example

## When component is initialized, Angular invokes ngOnInit

## When a component’s input properties change, Angular invokes ngOnChanges

## When a component is destroyed, Angular invokes ngOnDestroy

Here is the complete list of life cycle hooks provided by the Angular

1. ngOnChanges
2. ngOnInit
3. ngDoCheck
4. ngAfterContentInit
5. ngAfterContentChecked
6. ngAfterViewInit
7. ngAfterViewChecked
8. ngOnDestroy
9. **ngOnChanges**

The Angular invokes this life cycle hook whenever any data-bound property of the component or directive changes.

The parent component can communicate with the child component, if child component exposes a property decorated with @Input decorator. This hook is invoked in the child component, when the parent changes the Input properties. We use this to find out details about which input properties have changed and how they have changed.

We used this life cycle hook in tutorial Passing data to child component.

1. **ngOnInit**

This hook is called when the component is created for the first time. This hook is called after the constructor and first ngOnChanges hook.

This is a perfect place where you want to add any initialisation logic for your component.

Note that ngOnChanges hook is fired before ngOnInit. Which means all the input properties are available to use when the ngOnInit is hook is called

This hook is fired only once

This hook is fired before any of the child directive properties are initialized.

1. **ngDoCheck**

This event is called immediately after the ngOnInit. There after it is called on every change made to the component properties,  Whenever the event is raised or any operations that may result in change in value of properties.Basically this hook is executed for every change detection in cycles even there is no properties change

The Angular ngOnChanges hook does not detect all the changes made to the input properties. It Only detects when the Input Property is a primitive type or reference to the Input property changes

This hook is provided so as to implement custom change detection, whenever Angular fails to detect the changes made to Input properties

1. **ngAfterContentInit**

This Life cycle hook is called after the Component’s content has been fully initialized. This hook is called after the properties marked with ContentChild and ContentChildren are fully initialized.The Angular Component can include the external content from the child Components by transcluding them using the <ng-content></ng-content> element. This hook is fired after these projected contents are initialized.

This is a component only hook and called only once.

1. **ngAfterContentChecked**

This life cycle hook is called after the Components Content is checked by the Angular’s Change detection module. It is called immediately after ngAfterContentInit and after every subsequent ngDoCheck().

This is a component only hook

1. **ngAfterViewInit**

Similar to ngAfterContentInit, but invoked after Angular initializes the component’s views and all its child views. This is called once after the first ngAfterContentChecked.

A component-only hook.

1. **ngAfterViewChecked**

The Angular fires this hook after it checks the component’s views and child views. This event is fired after the ngAfterViewInit and after that for every subsequent ngAfterContentChecked hook.

This is a component only hook.

1. **ngOnDestroy**

This hook is called just before the Component/Directive instance is destroyed by Angular

You can perform any cleanup logic for the Component here. This is the correct place where you would like to Unsubscribe Observables and detach event handlers to avoid memory leaks.

## How to Use Lifecycle Hooks

## Import Hook interfaces

## Declare that Component/directive Implements lifecycle hook interface

## Create the hook method (e.g. ngOnOnChanges)

## The Order of Execution of Life Cycle Hooks

The Angular hooks are executed in this order:

When the Component is created

1. OnChanges
2. OnInit
3. DoCheck
4. AfterContentInit
5. AfterContentChecked
6. AfterViewInit
7. AfterViewChecked

When the Component with Child Component is created

1. OnChanges
2. OnInit
3. DoCheck
4. AfterContentInit
5. AfterContentChecked
   1. Child Component -> OnChanges
   2. Child Component -> OnInit
   3. Child Component -> DoCheck
   4. Child Component -> AfterContentInit
   5. Child Component -> AfterContentChecked
   6. Child Component -> AfterViewInit
   7. Child Component -> AfterViewChecked
6. AfterViewInit
7. AfterViewChecked

## Verifying the Order of Execution

## You can verify the Order of execution by building a simple application

## Parent Component

## Update the app.component.ts with following code

import { Component, AfterContentChecked, AfterContentInit, AfterViewChecked, AfterViewInit ,

DoCheck, OnChanges, OnDestroy, OnInit, Input, SimpleChanges } from '@angular/core';

@**Component**({

selector: 'my-app',

template: `

<h2>Life Cycle Hook</h2>

<button (click)="toggle()">Hide/Show Child </button>

<child-component \*ngIf="displayChild" [message]="'Hello'" ></child-component>

`

})

export class **AppComponent** implements **OnChanges**, **OnInit**, **DoCheck**,

**AfterContentInit**, **AfterContentChecked**,**AfterViewInit**, **AfterViewChecked**,**OnDestroy** {

displayChild: boolean=false;

constructor() {

**console**.**log**("AppComponent:Constructor");

}

**toggle**() {

this.displayChild=!this.displayChild;

}

**ngOnChanges**() {

**console**.**log**("AppComponent:OnChanges");

}

**ngOnInit**() {

**console**.**log**("AppComponent:OnInit");

}

**ngDoCheck**() {

**console**.**log**("AppComponent:DoCheck");

}

**ngAfterContentInit**() {

**console**.**log**("AppComponent:AfterContentInit");

}

**ngAfterContentChecked**() {

**console**.**log**("AppComponent:AfterContentChecked");

}

**ngAfterViewInit**() {

**console**.**log**("AppComponent:AfterViewInit");

}

**ngAfterViewChecked**() {

**console**.**log**("AppComponent:AfterViewChecked");

}

**ngOnDestroy**() {

**console**.**log**("AppComponent:OnDestroy");

}

}

## We are listening to all the hooks.

## Child Component can be added/removed using the ngIf directive.

## We are passing data to child component by binding to @Input property of the child component

## Child Component

import { Component, AfterContentChecked, AfterContentInit, AfterViewChecked, AfterViewInit ,

DoCheck, OnChanges, OnDestroy, OnInit, Input, SimpleChanges } from '@angular/core';

@**Component**({

selector: 'child-component',

template: `

<h2>Child Component</h2>

`

})

export class **ChildComponent** implements **OnChanges**, **OnInit**, **DoCheck**,

**AfterContentInit**, **AfterContentChecked**,**AfterViewInit**, **AfterViewChecked**,**OnDestroy** {

@**Input**() message="";

constructor() {

**console**.**log**("ChildComponent:Constructor");

}

**ngOnChanges**() {

**console**.**log**("ChildComponent:OnChanges");

}

**ngOnInit**() {

**console**.**log**("ChildComponent:OnInit");

}

**ngDoCheck**() {

**console**.**log**("ChildComponent:DoCheck");

}

**ngAfterContentInit**() {

**console**.**log**("ChildComponent:AfterContentInit");

}

**ngAfterContentChecked**() {

**console**.**log**("ChildComponent:AfterContentChecked");

}

**ngAfterViewInit**() {

**console**.**log**("ChildComponent:AfterViewInit");

}

**ngAfterViewChecked**() {

**console**.**log**("ChildComponent:AfterViewChecked");

}

**ngOnDestroy**() {

**console**.**log**("ChildComponent:OnDestroy");

}

}

## We are listening to all the hooks

## @Input property message is defined

## Angular Forms

The data entry forms can be a very simple to very complex. It can contain large no of input fields, Spanning multiple tabs. Forms may also contain complex validation logic interdependent on multiple fields.

* Initialize the forms fields and present it to the user
* Capture the data from the user
* Track changes made to the fields
* Validate the inputs
* Display helpful errors to the user

## Angular 2 forms module provides all the above services out of the box. It uses the two-way data binding & event binding bind the form field to the Angular 2 component class. It tracks changes made to the form fields so that we can respond accordingly. The Angular forms provide the built-in validators to validate the inputs. You can create your own custom validator. It presents the validation errors to the user. Finally, it encapsulates all the input fields into an object structure when the user submits the form.

## There are 2 approaches for building the forms

## Template Driven Forms - Logic of the form is placed in the template

## Model Driven Forms - Logic of the form is placed in the component

## Building Blocks of Angular 2 Forms

## The Angular 2 Forms module consists of three Building blocks, irrespective of whether you are using Template driven or Forms Driven model.

1. FormControl
2. FormGroup
3. FormArray
4. **FormControl**

A FormControl represents a single input field in an Angular form.

Consider a simple Text input box

 First Name : <input type="text" name="firstname" />

As a developer, you would like to know the current value in the Text box. You would also be like to know if the value is valid or not. If the user has changed the value(dirty) or is it unchanged. You would like to be notified when the user changes value.

The FormControl is an object that encapsulates all these information related to the single input element. It Tracks the value and validation status of each of these control

The FormControl is just a class. A FormControl is created for each form field. You can then refer to them in your component class and inspect its properties and methods

You can use FormControl to set the value of the Form field. You can find the status of form field like (valid/invalid, pristine/dirty, touched/untouched ) etc. You can add validation rules to it.

The above input field is created using the FormControl as shown below

let firstname= new FormControl();

* You can retrieve the current value in the input field using the value property

firstname.value

* You can check the validation status of the First Name element as shown below

firstname.errors // returns the list of errors

firstname.dirty // true if the value has changed (dirty)

firstname.touched // true if input field is touched

firstname.valid // true if the input value has passed all the validation

1. **FormGroup**

FormGroup is a group of **FormControl** **instances**.

Often forms have more than one field. It is helpful to have a simple way to manage the Form controls together.

Consider the following Form. we have three input fields street, city & Pincode.

city : <input type="text" name="city" >

Street : <input type="text" name="street" >

PinCode : <input type="text" name="pincode" >

All of the above input fields are represented as the separate FormControl. If we wanted to check the validity of our form, we have to check the validity of each and every FormControl for validity. Imagine Form having large no of fields. It is cumbersome to loop over large no of FormControls and check for validity.

FormGroup solve’s this issue by providing a wrapper interface around a collection of FormControls A FormGroup tracks the status of each child FormControl and aggregates the values into one object. with each control name as the key

We can group these input fields under the group address as shown below

let address= new FormGroup({

street : new FormControl(""),

city : new FormControl(""),

pinCode : new FormControl("")

})

In the above example, the address is our FormGroup, consisting of 3 FormControl‘s city, street, and Pincode. Now we can check the validity of the entire group together. For example, if the state is invalid, then the address returns the invalid state.

* You can read the value of an address using the value method, which returns the JSON object as shown below

  address.value

* The Return value

 address {

        street :"",

        city:"",

        Pincode:""

    }

* You can access child control as  address.street
* Check the Validation status as follows

address.errors // returns the list of errors

address.dirty // true if the value of one of the child control has changed (dirty)

address.touched // true if one of the child control is touched

address.valid // true if all the child controls passed the validation

1. **FormArray**

The FormArray‘s are array of FormControl’s

The FormArray’s are very much similar to FormGroups. Here FormControls are represented as an array of objects.

The key difference is that its data gets serialized as an array. This might be especially useful when you don’t know how many controls will be present within the group, like dynamic forms.

The FormArray are defined as shown below

let address= new FormArray({

    street : new FormControl(""),

    city : new FormControl(""),

    pinCode : new FormControl("")

})

* **Angular Template Driven Forms**

In Template Driven Forms we specify behaviors/validations using directives and attributes in our template and let it work behind the scenes. All things happen in Templates hence very little code is required in the component class Enabling Template Driven Forms

**Example:**

1. Tell our @NgModule to use the FormsModule from @angular/forms. In the app.module.ts file. The forms module contains all the form directives and constructs for working with forms.

**app.module.ts**

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

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

FormsModule<strong>,</strong>

HttpModule

],

providers: [],

bootstrap: [AppComponent]

})

1. Create a simple template

**app.component.html**

<form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)" >

<div class="form-group">

<label for="firstname">First Name</label>

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

</div>

<div class="form-group">

<label for="lastname">Last Name</label>

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

</div>

<div class="form-group">

<button type="submit">Submit</button>

</div>

<pre>{{contactForm.value | json }} </pre>

</form>

1. Create a Root Component as

**app.component.ts**

Now we need to receive the data in Component class from our form. To do this we need to create the onSubmit method in our component class. The submit method receives the reference to the ngForm directive, which we named is as contactForm. The contactForm exposes the value method which returns the form fields as Json object.

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

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

@Component({

selector: 'my-app',

templateUrl:'app/app.component.html'

})

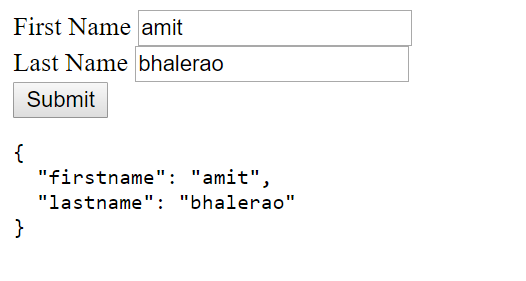
export class AppComponent {

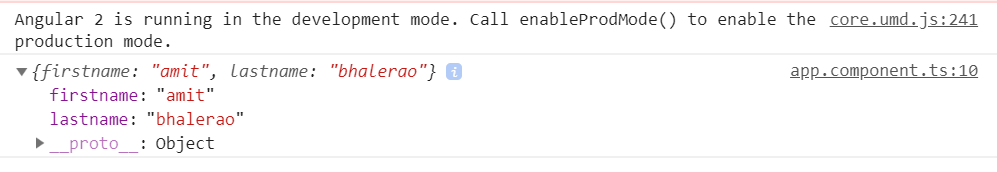
onSubmit(contactForm) {

console.log(contactForm.value);

}

}





* **Grouping FormControl’s using FormGroup**

Grouping is done using the directive ngModelGroup. All you need to do is to enclose the fields inside a div element with ngModelGroup directive applied on it as shown below

<div ngModelGroup="address">

<div class="form-group">

<label for="city">City</label>

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

</div>

<div class="form-group">

<label for="street">Street</label>

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

</div>

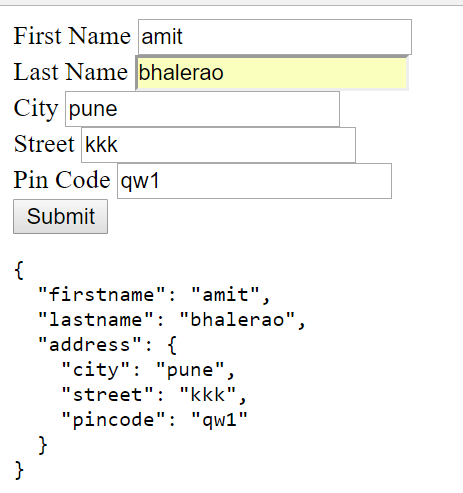
<div class="form-group">

<label for="pincode">Pin Code</label>

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

</div>

</div>



* **Initializing the Value**

The simplest way is to create a Variable in Component class and Set up one way or two way binding in the Template.

The correct way is to create an Angular Form is to build a model class. Let us build a model for our Form. Add it in app.component.ts

class Contact {

    firstname: string ;

    lastname: string ;

    address: {

       city:string

       street: string

       pincode: string

    }

}

And in our AppComponent Class create an instance of the Contact class and initialize it with the default values. In Real life application, you may initialize it with the data from the database.

**app.component.ts**

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

@Component({

selector: 'app-root',

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

})

export class AppComponent

{

submitted:boolean;

contact:Contact = {

firstname:"Sachin",

lastname:"Tendulkar",

address: { city:"Mumbai",street:"Perry Cross Rd", pincode:"400050"}

};

onSubmit(contactForm) {

console.log(contactForm.value);

this.submitted = true;

this.contact = contactForm.value;

}

}

class Contact {

firstname: string ;

lastname: string ;

address: {

city:string

street: string

pincode: string

}

}

**app.component.html**

<form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)" >

<div class="form-group">

<label for="firstname">First Name</label>

<input type="text" class="form-control" name="firstname" [ngModel]="contact.firstname" required>

</div>

<div class="form-group">

<label for="lastname">Last Name</label>

<input type="text" class="form-control" name="lastname" [ngModel]="contact.lastname" required>

</div>

<div ngModelGroup="address">

<div class="form-group">

<label for="city">City</label>

<input type="text" class="form-control" name="city" [ngModel]="contact.address.city" >

</div>

<div class="form-group">

<label for="street">Street</label>

<input type="text" class="form-control" name="street" [ngModel]="contact.address.street" >

</div>

<div class="form-group">

<label for="pincode">Pin Code</label>

<input type="text" class="form-control" name="pincode" [ngModel]="contact.address.pincode">

</div>

</div>

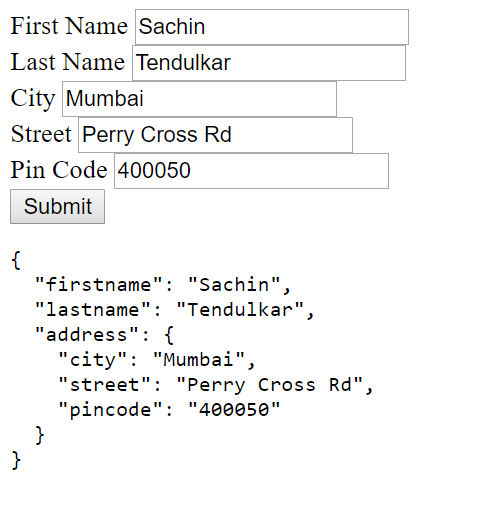
<div class="form-group">

<button type="submit">Submit</button>

</div>

<pre>{{contactForm.value | json }} </pre>

</form>



* **One Way Binding**

**Syntax:**

[ngModel]

**Example :**

<input type="text" class="form-control" name="firstname" [ngModel]="contact.firstname">

* **Two Way Binding**

**Syntax:**

[(ngModel)]

**Example :**

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

* **Validations:**
* **Built-in Validators**

The Angular Built-in validators use the HTML5 validation attributes like required, minlength, maxlength & pattern. Angular 2 interprets these validation attributes and add the validator functions to FormControl.

The list of HTML5 validation attributes supported in Angular

* **required**: There must be a value
* **minlength**: The number of characters must be more than the value of the attribute.
* **maxlength**: The number of characters must not exceed the value of the attribute.
* **pattern**: The value must match the pattern.

**Example :**

<input type="text" class="form-control" name="firstname" ngModel #firstname="ngModel" required minlength="10">

**Displaying the Validation/Error messages**

<div \*ngIf="!contactForm.controls.firstname?.valid && (contactForm.controls.firstname?.dirty || myForm.controls.firstname?.touched)" class="alert alert-danger">

<div [hidden]="!contactForm.controls.firstname.errors.required">

First Name is required

</div>

</div>

* **Angular Model(Reactive) Driven Forms**

The model driven Forms allows you to have an ability to setup validation rules in code rather than as directive in Template. You can subscribe to field value changes in your code. Since everything happens in Component class, you can easily write unit Tests.

**Enabling Model Driven Forms**

We need to import ReactiveFormsModule instead of FormModule to make use of Model Driven Forms. We should also add the ReactiveormsModule to the imports metadata property array.

**app.module.ts**

import { BrowserModule } from '@angular/platform-browser';

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

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

import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';

@NgModule({

    declarations: [

        AppComponent

    ],

    imports: [

        BrowserModule,

        ReactiveFormsModule,

        HttpModule

    ],

    providers: [],

        bootstrap: [AppComponent]

})

export class AppModule { }

## Angular Services

## Service is a piece of reusable code with a Focused Purpose. A code that you will use it in many components across your application.

## Our components need to access to data. You can write data access code in each component, but that is very inefficient. The Best way is to create a single reusable code and use it to retrieve data in every component that needs it. That is exactly where services come in.

## What services are used for

1. Features that are independent of components such a logging services
2. Share logic across components
3. Encapsulate external interactions like data access

## How to create a Service in Angular 2

An Angular 2 service is simply a JavaScript function. All we need to do is to create a class and add methods & properties. We can then create an instance of this class in our component and call its methods.

One of the best uses of services is to get the data from the data source. Let us create a simple service, which gets the product data and passes it to our component.

**1) Product.ts**

export class Product {

    productID:number ;

    name: string ;

    price:number;

    constructor(productID:number, name: string , price:number) {

        this.productID=productID;

        this.name=name;

        this.price=price;

    }

}

**2) Product.service.ts**

import {Product} from './Product'

export class ProductService{

  public getProducts() {

      let products:Product[];

        products=[

            new Product(1,'Memory Card',500),

            new Product(1,'Pen Drive',750),

            new Product(1,'Power Bank',100)

        ]

        return products;

    }

}

First, we have imported the Product

We have created the ProductService class and exported it.

We have created a getProducts method, which returns the collection of the products. In this example, we have hardcoded the products. In real life, you would send an HTTP request to your backend API to get the data

**3) app.component.ts**

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

import { ProductService } from './product.service';

import { Product } from './product';

@Component({

   selector: 'app-root',

   templateUrl: './app.component.html'

})

export class AppComponent{

    products:Product[];

    let productService;

    constructor(){

       this.productService=new ProductService();

    }

    getProducts() {

       this.products=this.productService.getProducts();

    }

}

First, we are imported both Product & ProductService

In the constructor of the AppComponet, we have created the instance of the ProductSerivce

The getProducts method calls the getProducts method of the ProductService. The returned list of Products are stored in the local variable products

**4) app.component.html**

<div class="container">

<h1 class="heading"><strong>Services </strong>Demo</h1>

<button type="button" (click)="getProducts()">Get Products</button>

<div class='table-responsive'>

<table class='table'>

<thead>

<tr>

<th>ID</th>

<th>Name</th>

<th>Price</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let product of products;">

<td>{{product.productID}}</td>

<td>{{product.name}}</td>

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

</tr>

</tbody>

</table>

</div>

</div>

We have added a button “Get Products”, which is hooked to getProducts method of the component class via [event binding](https://www.tektutorialshub.com/data-binding-angular-2/).

We are displaying the returned products via [ngFor directive](https://www.tektutorialshub.com/angular-2-ngfor-directive/).

Finally, run the code and click on Get Product button and see the Products are displayed.

## Angular Dependency Injection

## Dependency Injection (DI) is a technique in which we provide an instance of an object to another object, which depends on it. This is technique is also known as “Inversion of Control” (IoC)

We have instantiated the productService directly in our component as shown below.

export class AppComponent

{

    products:Product[];

    getProducts() {

        let productService=new ProductService();

        this.products=productService.getProducts();

    }

}

The ProductService Instance is local to the Component. The AppComponent is now tightly coupled to the ProductService, This tight coupling brings a lot of Issues.

What if ProductService requires a parameter. We need to change the AppComponent as shown below

this.productService=new ProductService(SomeParameters);

We need to do that at every component/service, where we have used ProductService.

It is hard to test this Component as it is difficult to provide the Mock for the ProductService. For Instance, what if we wanted to substitute out the implementation of ProductService for MockProductService during testing.

Our Component Class has now tied one particular implementation of ProductService. It will make it difficult to reuse our component.

We would also like to make our ProductService singleton so that we can use it across our application.

How to solve all these problems. Move the creation of ProductService to the constructor the AppComponent class as shown below.

export class AppComponent

{

    products:Product[];

    constructor(private productService:ProductService) {

    }

    getProducts() {

        this.products=this.productService.getProducts();

    }

}

Now our AppComponent does not create the instance of the ProductService. It just asks for it in its constructor.Now the responsibility of Creating the ProductService falls on the creator of the AppComponent.

The above pattern is known as **Dependency Injection Pattern**.

## Why use Dependency Injection

Our Component is now loosely coupled to the ProductService. AppComponent does not bother how to create the ProductService. If the ProductService does require a Parameter then you do not have to change the AppComponent.

AppComponent is now easier to Test.Our AppComponent is not dependent on a Particular implementation of ProductService anymore. It will work with any implementation of ProductService that is passed on to it. You can just create a mockProductService Class and pass it while testing.

Reusing of the component is becomes easier. Our Component will now work with any ProductService as long as the interface is honoured.

Dependency injection pattern made our AppComponent reusable, Testable, Maintainable etc.

* ***Angular Dependency Injection Framework consists of four main parts***

**Consumer**: The Component that needs the Dependency

**Dependency**: The Service that is being injected.

**Provider**: Maintains the list of Dependencies. It provides the instance of dependencies to the injector

**Injector**: Responsible for injecting the instance of the Dependency to the Consumer

## How Dependency Injection works in Angular

1. The dependencies are registered with the Provider. This is done in the Providers metadata of the Consumer.
2. Angular Provides an instance of injector & provider to every Consumer.
3. Consumer when instantiated, It declares the Dependencies it needs in its constructor.
4. Injector reads the Dependencies from the constructor of the Consumer and looks for the dependency in the provider. The Provider provides the instance and injector, then injects it into the consumer. If the Dependency instance already exists, then it is reused making the dependency singleton.

## How to Use Dependency Injection

**1) Product.ts**

export class Product {

    productID:number ;

    name: string ;

    price:number;

    constructor(productID:number, name: string , price:number) {

        this.productID=productID;

        this.name=name;

        this.price=price;

     }

 }

**2) Product.service.ts**

import {Product} from './Product'

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

@Injectable()

export class ProductService{

  public getProducts() {

      let products:Product[];

        products=[

            new Product(1,'Memory Card',500),

            new Product(1,'Pen Drive',750),

            new Product(1,'Power Bank',100)

        ]

        return products;

    }

}

**3) app.component.ts**

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

import { ProductService } from './product.service';

import { Product } from './product';

@Component({

selector: 'app-root',

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

providers: [ProductService]

})

export class AppComponent{

products:Product[];

constructor(private productService:ProductService) { }

getProducts() {

this.products=this.productService.getProducts();

}

}

## Angular HTTP

## The HttpClient is a separate model in Angular and is available under the @angular/common/http package. All you need to do is to import it and inject it into our component/service. Then, Use HttpClient.Get method to send an HTTP Request and Subscribe to the response Asynchronously. And when the response arrives map it the desired object and displays the result.a

* 1. **Import HttpClient Module in Root Module**

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

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

@NgModule({

    declarations: [

        AppComponent

    ],

    imports: [

        HttpClientModule

    ],

    providers: [],

    bootstrap: [AppComponent]

})

export class AppModule { }

* 1. **Import Required Module in Component/Service**

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

import { Observable} from 'rxjs/Rx';

* 1. **Inject HttpClient service into your component/service**

constructor(public http: HttpClient) {

}

* 1. **Call the HttpClient.Get method**

We send the HttpClient.get request to our Service API endpoint. The HttpClient.Get method returns an Observable, which is subscribed. The Result arrives in JSON format which is displayed to the user.

public getData() {

  this.HttpClient.get<any[]>(this.baseUrl+'users/'+this.userName+'/repos')

           .subscribe(data => {

               this.data= data;

           },

           error => {

           }

  );

}

* **HttpClient.get**

The HttpClient.get sends the HTTP Get Request to the API endpoint and parses the returned result to the desired type. The default response type is JSON. If you want any other type, then you need to specify explicitly using the responseType parameter.

**Parameters of the HttpClient.get**

* **Headers** : Allows you to add HTTP headers to the outgoing requests.
* **Observe** : The HttpClient.get method returns the body of the response parsed as JSON (or type specified by the responseType). Sometimes you may need to read the entire response along with the headers and status codes. To do this you can set the observe property to **response**.The allowed options are
* response which returns the entire response
* body which returns only the body
* events which returns the response with events.

##### **Params** :Allows us to Add the URL parameters to the Get Request

* **reportProgress** :This is a boolean property. Set this to true, if you want to get notified of the progress of the Get Request. This is a pretty useful feature when you have a large amount of data to download (or upload) and you want the user to notify of the progress.
* **responseType**:Json is the default response received. In case you want a different type of response, then you need to use this parameter. The Allowed Options are **arraybuffer**, **blob**, **json** and **text**.
* **withCredentials:**It is of boolean type. If the value is true then HttpClient.get will request data with credentials

Notice that when we use Angular HttpClient Services to get data, it actually returns an Observables.

* **What is Observable?**

Observable help us to manage async data. You can think of Observables as an array of items, which arrive asynchronously over time.

The observables implement the observer design pattern, where observables maintains a list of dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods

Observer subscribes to an Observable. The observer reacts when the value of the Observable changes. An Observable can have multiple subscribers and all the subscribers are notified when the state of the Observable changes.

The Observables are used extensively in Angular. The new HTTP service and Event system are all Observable based.

The Observables are Proposed Feature for the next version of Javascript. The Angular Uses Third party library called Reactive Extensions or RxJs to implement the Observables.

* **Observables Operators :** Operators are methods that operate on an Observable and return an observable. Each Operator modifies the value it receives. These operators are applied one after the other in a chain.The RxJs Provides several Operators, which allows you to filter, select, transform, combine and compose Observables. Examples of Operators are map, filter, take, merge
* **How to use RxJs :** The RxJs is a very large library. Hence Angular exposes a stripped down version of Observables. You can import it using the following import statement

**Example :**

Let us query the GitHub repository to get the details of the repositories for a selected user.

1. **Repository Model**

Create repos.ts file and add the following code. This is a simplified model for GitHub repository.

export class repos {

    id: string;

    name: string;

    html\_url: string;

    description: string;

}

1. **Github.service.ts**

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

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

import { Observable} from 'rxjs/Rx';

import { repos} from './repos';

@Injectable()

export class GitHubService {

baseURL:string="https://api.github.com/";

constructor(private http:HttpClient){

}

getRepos(userName:string): Observable<repos[]> {

return this.http.get<repos[]>(this.baseURL + 'users/' + userName + '/repos')

}

}

* First, we import required HttpClient module from @angular/common/http package. We also need Injectable as we will be injecting HttpClient via Dependency Injection.
* The Observable is included from the RxJs library.
* Set the baseURL variable to GitHub API endpoint
* Inject HttpClient into the service.
* Next, is the GetRepos method, which takes the username as the argument and returns an Observable<repos[]> observable of Repos.
* Inside the getRepos method, we invoke httpClient.get method.
* The HttpClient.get method allows us to cast the returned response object to a type we require. We make use of that feature and supply the type for the returned value httpClient.get<repos[]>
* Finally, We pass the URL as the parameter to the get method.

1. **app.component.ts**

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

import { Observable} from 'rxjs/Rx';

import { GitHubService } from './github.service';

import { repos} from './repos';

@Component({

selector: 'app-root',

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

})

export class AppComponent

{

userName: string ="amitmca"

repos: repos[];

loading: boolean=false;

errorMessage;

constructor(private githubService: GitHubService) {

}

public getRepos() {

this.loading=true;

this.errorMessage="";

this.githubService.getRepos(this.userName)

.subscribe((response) => {this.repos=response;},

(error) => {this.errorMessage=error; this.loading=false; },

() => {this.loading=false;})

}

}

We are using Dependency injection to inject our newly created gitHubService into the component.

We have four variables.

**userName**: GitHub username

**Repos**: variable to hold the data received from GitHub

**Loading**: to display the loading indicator to the user

**errorMessage**: to display the user-friendly error message

GetRepos method

We make a call to GetRepos to retrieve the Repos. We start with initializing loading variable to true so that we can display a loading indicator to the user.

Next, Invoke the getRepos method of the gitHubService, with the userName as the argument.

This method returns Observable<repos>. We then subscribe to it.

**Subscribe method**

The Subscribe operator is the glue that connects an observer to an Observable

We need to Subscribe to an observable to see the results of observables.

The subscribe method has three arguments. Each specifies the action to be taken when a particular event occurs

.subscribe(success, failure, completed);

**Success:** This event is raised whenever observable returns a value. We use this event to assign the response to the repos

(response) => {this.repos=response;}

**Failure:** This event occurs, when observable is failed to generate the expected data or has encountered some other error. The error message is assigned to the local variable errorMessage so as to display the result to the user. The loading indicator is also disabled as the observable now terminates due to the error

(error) => {this.errorMessage=error; this.loading=false; },

**Completed**: This event fires, when the observables complete its task. We disable the loading indicator here.

() => {this.loading=false;})

1. **app.component.html**

<h1 class="heading"><strong>HTTP </strong>Demo</h1>

<div class="form-group">

<label for="userName">GitHub User Name</label>

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

</div>

<div class="form-group">

<button type="button" (click)="getRepos()">Get Repos</button>

</div>

<div \*ngIf="loading">loading...</div>

<div \*ngIf="errorMessage" class="alert alert-warning">

<strong>Warning!</strong> {{errorMessage}}

</div>

<div class='table-responsive'>

<table class='table'>

<thead>

<tr>

<th>ID</th>

<th>Name</th>

<th>HTML Url</th>

<th>description</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let repo of repos;">

<td>{{repo.id}}</td>

<td>{{repo.name}}</td>

<td>{{repo.html\_url}}</td>

<td>{{repo.description}}</td>

</tr>

</tbody>

</table>

</div>

<pre>{{repos | json}}</pre>

</div>

1. **app.module.ts**

import { BrowserModule } from '@angular/platform-browser';

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

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

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

import { AppComponent } from './app.component';

import { GitHubService } from './github.service';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

HttpClientModule,

FormsModule

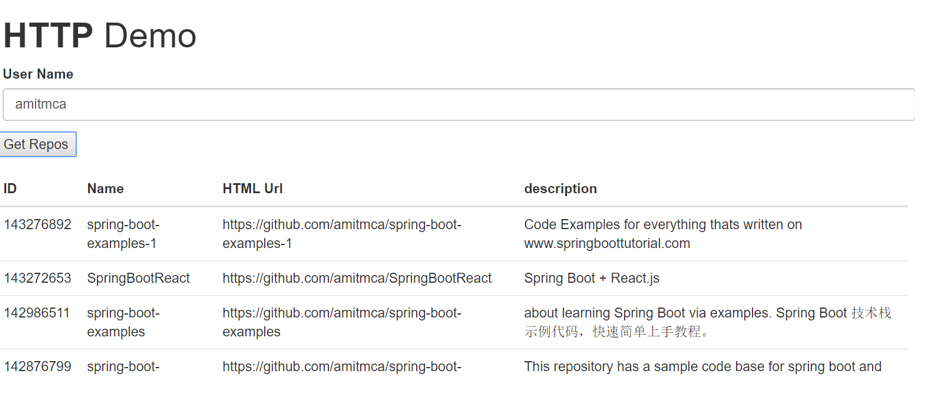
],

providers: [GitHubService],

bootstrap: [AppComponent]

})

export class AppModule { }



* **Angular HttpParams()**

We have created GitHubService. The Service issued GET Request to GitHub API Endpoint to retrieve the List of Repositories belonging to a particular User.

The GitHub API also have a set of parameters, which allows us to specify how we want to sort, which page to retrieve, No of Entries per page and type of the Repository to retrieve etc.

Ex: <https://api.github.com/users/tekTutorialsHub/repos?sort=description&page=2>

The Above query will return the result sorted on the description and retrieves only the second page. The string sort=description&page=2 after the question mark is called URL Parameter or Query strings /Query Parameters. The Question mark is used as a separator. The URL Parameters are also known as get params

**HttpParams()**

The Query parameters are added using the helper class HttpParams. The HttpParams is passed as one of the argument to HttpClient.get method.

To use HttpParams, you need to import it first as shown below.

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

Then create an instance of the HttpParams class.

const params = new HttpParams()

.set('page', PageNo)

.set('sort', SortOn);

And then call the httpClient.get method passing the params as the argument.

return this.httpClient.get<repos[]>(this.baseURL + 'users/' + userName + '/repos',{params})

* **HttpParams members**

1. **HttpParams.set**

**set(param: string, value: string): HttpParams**

Construct a new body with a new value for the given parameter name. If the parameter already exists then it is replaced else it is added.

params = new HttpParams()

    .set('page', '2')

    .set('page', '3')

    .set('sort', 'name');

console.log(params.toString()); //Returns page=3&sort=name

1. **HttpParams.append**

**append(param: string, value: string): HttpParams**

Construct a new body with an appended value for the given parameter name. Always appends the value irrespective whether the parameter exists. The page parameter is appended twice in the following example.

params = new HttpParams()

    .set('page', '2')

    .append('page', '3')

    .set('sort', 'name');

console.log(params.toString()); //Returns page=2&page=3&sort=name

1. **HttpParams.has**

**has(param: string): Boolean**

Returns true if the given parameter name already exists in the HttpParams

params = new HttpParams()

    .set('sort', 'name');

if (!params.has('page')) {

    params = params.set('page', PageNo)

}

1. **HttpParams.get**

**get(param: string): string | null**

Get the first value for the given parameter name, or null if it’s not present.

params = new HttpParams()

    .set('page', '2')

    .append('page', '3')

    .set('sort', 'name');

console.log(params.get('page')); // Returns 2 The First occurance of Page

1. **HttpParams.getAll**

**getAll(param: string): string[] | null**

Get all values as for the given parameter name, or null if it’s not present.

params = new HttpParams()

.set('page', '2')

.append('page', '3')

.set('sort', 'name');

console.log(params.getAll('page')); // Returns ["2", "3"] All occurance of Page

1. **HttpParams.keys**

**keys(): string[]**

Get all the parameter names for this body.

let params = new HttpParams()

    .set('page', '2')

    .set('sort', 'name');

console.log(params.keys()); //Returns ["page", "sort"]

1. **HttpParams.delete**

**delete(param: string, value?: string): HttpParams**

Deletes the parameter and returns the new parameter collection. You can delete using the parameter name or by using the name & value. The no argument is specified, then the all parameters are deleted.

params = new HttpParams()

    .set('page', '2')

    .Append('page', '3')

    .set('sort', 'name');

params = params.delete('page', '3'); //Deletes the parameter page with value 3

params = params.delete('page'); //Delete the all the parameter of page

params = params.delete(''); //Delete all parameters

1. **HttpParams.toString**

**toString(): string**

Serialize the body to an encoded string, where key-value pairs (separated by =) are separated by &s.

params = new HttpParams()

    .set('page', '2')

    .Append('page', '3')

    .set('sort', 'name');

console.log(params.toString()); //Returns page=2&page=3&sort=name

**Example :**

1. **Repository Model**

Create repos.ts file and add the following code. This is a simplified model for GitHub repository.

export class repos {

    id: string;

    name: string;

    html\_url: string;

    description: string;

}

1. **Github.service.ts**

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

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

import { Observable} from 'rxjs/Rx';

import { repos} from './repos';

@Injectable()

export class GitHubService {

baseURL:string="https://api.github.com/";

constructor(private httpClient:HttpClient){

}

getRepos(userName: string, PageNo: string, SortOn: string): Observable<repos[]> {

let params = new HttpParams()

.set('page', PageNo)

.set('sort', SortOn);

console.log(params.toString());

return this.httpClient.get<repos[]>(this.baseURL + 'users/' + userName + '/repos', {params});

}

}

1. **app.component.ts**

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

import { Observable} from 'rxjs/Rx';

import { GitHubService } from './github.service';

import { repos} from './repos';

@Component({

selector: 'app-root',

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

})

export class AppComponent

{

userName = 'amitmca';

pageNo = '1';

sortOn = 'description';

repos: repos[];

loading = false;

errorMessage = '';

constructor(private githubService: GitHubService) {

}

public getRepos() {

this.loading=true;

this.errorMessage="";

this.githubService.getRepos(this.userName)

.subscribe((response) => {this.repos=response;},

(error) => {this.errorMessage=error; this.loading=false; },

() => {this.loading=false;})

}

}

1. **app.component.html**

<div class="container">

    <h1 class="heading"><strong>HTTP </strong>Demo</h1>

       <div class="form-group">

       <label for="userName">User Name</label>

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

       <label for="pageNo">Page No</label>

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

       <label for="sortOn">Sorted On</label>

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

    </div>

    <div class="form-group">

        <button type="button" (click)="getRepos()">Get Repos</button>

    </div>

    <div \*ngIf="loading">loading...</div>

   <div \*ngIf="errorMessage" class="alert alert-warning">

       <strong>Warning!</strong> {{errorMessage}}

    </div>

    <div class='table-responsive'>

            <table class='table'>

                <thead>

                    <tr>

                        <th>ID</th>

                        <th>Name</th>

                        <th>HTML Url</th>

                        <th>description</th>

                    </tr>

                </thead>

                <tbody>

                    <tr \*ngFor="let repo of repos;">

                        <td>{{repo.id}}</td>

                        <td>{{repo.name}}</td>

                        <td>{{repo.html\_url}}</td>

                        <td>{{repo.description}}</td>

                    </tr>

                </tbody>

            </table>

        </div>

    <pre>{{repos | json}}</pre>

</div>

1. **app.module.ts**

import { BrowserModule } from '@angular/platform-browser';

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

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

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

import { AppComponent } from './app.component';

import { GitHubService } from './github.service';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

HttpClientModule,

FormsModule

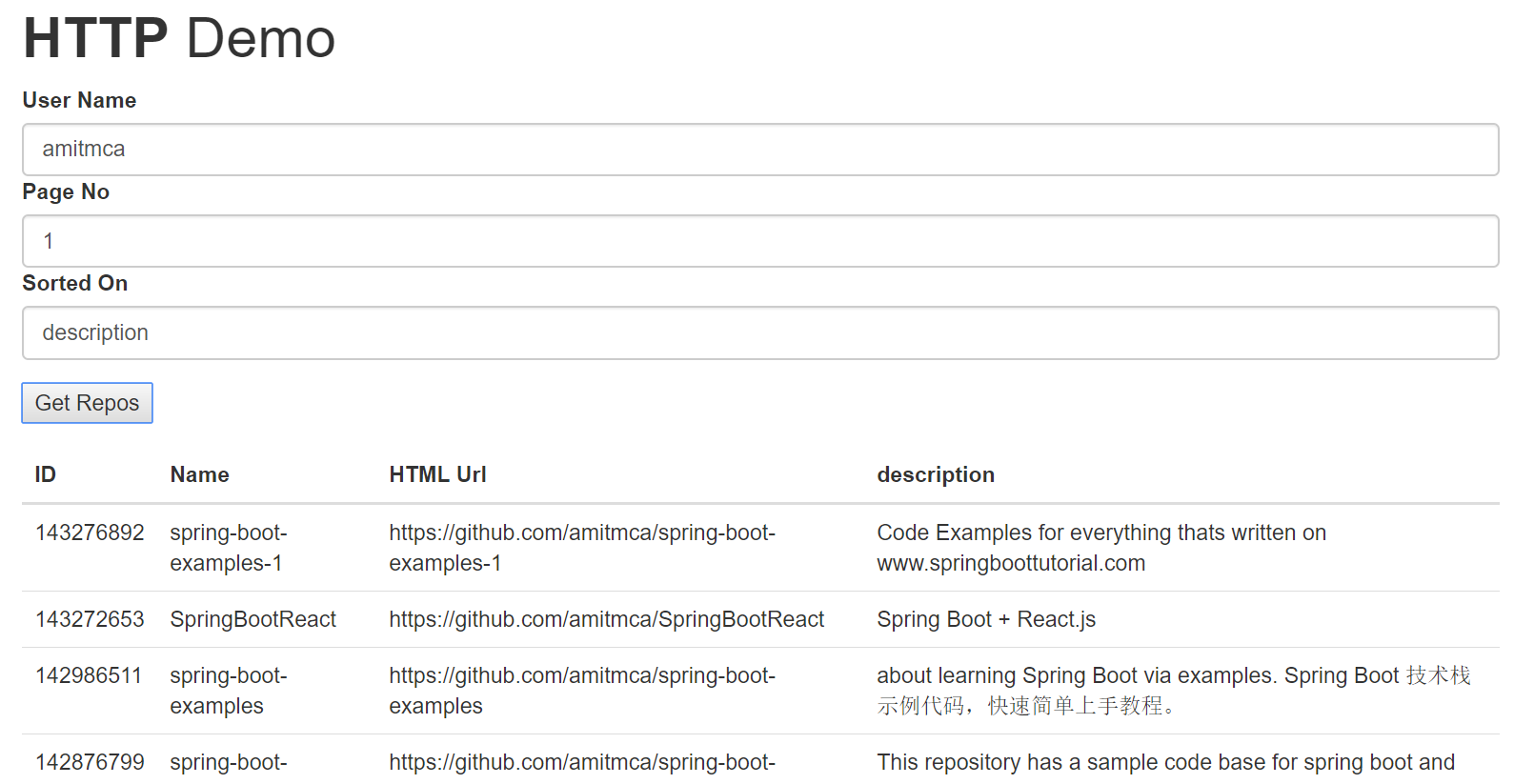
],

providers: [GitHubService],

bootstrap: [AppComponent]

})

export class AppModule { }



## Angular Routing

The Router is a separate module in Angular. It is in its own library package, @angular/router. The Router Module provides the necessary service providers and directives for navigating through application views. Using Angular Router you can

* Navigate to a specific view by typing a URL in the address bar
* Pass optional parameters to the View
* Bind the clickable elements to the View and load the view when user performs application tasks
* Handles back and forward buttons of the browser
* Allows you to dynamically load the view
* **Components of Router Module**

**1) Router**

The Angular Router is an object that enables navigation from one component to the next component as users perform application tasks like clicking on menus links, buttons or clicking on back/forward button on the browser. We can access the router object and use its methods like navigate() or navigateByUrl(), to navigate to a route

**2) Route**

Route tells the Angular Router which view to display when a user clicks a link or pastes a URL into the browser address bar. Every route consists of a path and a component it is mapped to. The Router object parses and builds the final URL using the Route

**3) Routes**

Routes is an array of Route objects our application supports

**4) RouterOutlet**

The RouterOutlet is a directive (<router-outlet>) that serves as a placeholder, where the router should display the view

**5) RouterLink**

The RouterLink is a directive that binds the HTML element to a Route. Clicking on the HTML element, which is bound to a RouterLink, will result in navigation to the Route. The RouterLink may contain parameters to be passed to the route’s component.

**6) RouterLinkActive**

RouterLinkActive is a directive for adding or removing classes from an HTML element that is bound to a RouterLink. Using this directive, we can toggle CSS classes for active RouterLinks based on the current RouterState

**7) ActivatedRoute**

The ActivatedRoute is an object that represents the currently activated route associated with the loaded Component.

**8) RouterState**

The current state of the router including a tree of the currently activated routes together with convenience methods for traversing the route tree.

**9) RouteLink Parameters array**

The Parameters or arguments to the Route. It is an array which you can bind to RouterLink directive or pass it as an argument to the Router.navigate method.

* **How to configure Angular Router**

To Configure the Router in Angular, you need to follow these steps

* Set the <base href>
* Define routes for the view
* Register the Router Service with Routes
* Map HTML Element actions to Route
* Choose where you want to display the view
* **Set the <base href>**

The HTML <base> element specifies the base URL to use for all relative URLs contained within a document. The Angular Router uses the HTML5 style of Routing (or PathLocationStrategy) as the default option. The router makes use of the browser’s history API for navigation and URL interaction.**<base href="/">.**

To make HTML5 routing to work, we need to set up the “**base href”** in the DOM. This is done in app’s *index.html* file immediately after the head tag.

* **Define routes for the view**

Next, create an array of route objects. Each route maps path (URL Segment) to the component.

const appRoutes={ path: 'product', component: ProductComponent }

**path**: The URL path segment of the route. We will use this value to refer to this route elsewhere in the app

**component**: The component to be loaded.

This route tells angular to render ProductComponent when the user navigate to the URL “/product”

* **Register the Routes**

Import the Router Module from @angular/router library in the root module of the application

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

Then, install the routes using the RouterModule.forRoot method, passing the routes as the argument in the imports array

imports: [RouterModule.forRoot(routes)],

* **Map Action to Routes**

Next, we need to bind the click event of the link, image or button to a route. This is done using the routerlink directive

<li><a [routerLink]="['product']">Product</a></li>

The routerLink directive accepts an array of route names along with parameters. This array is called as Link Parameters array.

When the application requests navigation to the route “product”, the router looks in the routes array and activates the instance of the component associated with the route “product”, which is ProductComponent. The browser address location & history is also updated to /product

* **Choose where you want to display**

Finally, we need to tell the angular where to display the view. This is done using the RouterOutlet directive as shown. We will add the following directive to the root component.

<router-outlet></router-outlet>

**Example :**

1. [**home.component.ts**](http://home.component.ts) **: HomeComponent**

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

@Component({

template: `<h1>Welcome!</h1>

<p>This is Home Component </p> `

})

export class HomeComponent { }

1. **contact.component.ts : ContactComponent**

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

@Component({

template: `<h1>Contact Us</h1>

<p>Pune </p>`

})

export class ContactComponent { }

* **product.ts**

export class Product {

constructor(productID:number, name: string , price:number) {

this.productID=productID;

this.name=name;

this.price=price;

}

productID:number ;

name: string ;

price:number;

}

* **product.service.ts**

import { Observable } from 'rxjs/Observable';

import {Product} from './Product'

export class ProductService{

public getProducts() {

let products:Product[];

products=[

new Product(1,'Memory Card',500),

new Product(2,'Pen Drive',750),

new Product(3,'Power Bank',100)

]

return products;

}

public getProduct(id) {

let products:Product[]=this.getProducts();

return products.find(p => p.productID==id);

}

}

* **product.component.ts**

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

import { ProductService } from './product.service';

import { Product } from './product';

@Component({

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

})

export class ProductComponent

{

products:Product[];

constructor(private productService:ProductService){

}

ngOnInit() {

this.products=this.productService.getProducts();

}

}

* **product.component.html**

<h1>Product List</h1>

<div class='table-responsive'>

<table class='table'>

<thead>

<tr>

<th>ID</th>

<th>Name</th>

<th>Price</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let product of products;">

<td>{{product.productID}}</td>

<td><a [routerLink]="['detail',product.productID]">{{product.name}} </a> </td>

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

</tr>

</tbody>

</table>

</div>

<router-outlet></router-outlet>

1. **error.component.ts : ErrorComponent**

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

@Component({

template: `<h1>Page not found</h1>

<p>This is a Error Page</p>

`

})

export class ErrorComponent {

}

1. **index.html**

<!doctype html>

<html>

<head>

<base href="/">

<meta charset="utf-8">

<title>Angular 2 Routing</title>

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="icon" type="image/x-icon" href="favicon.ico">

<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" rel="stylesheet">

</head>

<body>

<app-root>Loading...</app-root>

</body>

</html>

1. **app.routes.ts**

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

import { HomeComponent} from './home.component'

import { ContactComponent} from './contact.component'

import { ProductComponent} from './product.component'

import { ErrorComponent} from './error.component'

export const appRoutes: Routes = [

{ path: 'home', component: HomeComponent },

{ path: 'contact', component: ContactComponent },

{ path: 'product', component: ProductComponent },

{ path: '', redirectTo: 'home', pathMatch: 'full' },

{ path: '\*\*', component: ErrorComponent }

];

1. **app.module.ts**

import { BrowserModule } from '@angular/platform-browser';

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

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

import { HttpModule } from '@angular/http';

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

import { AppComponent } from './app.component';

import { HomeComponent} from './home.component'

import { ContactComponent} from './contact.component'

import { ProductComponent} from './product.component'

import { ErrorComponent} from './error.component'

import { ProductService } from './product.service';

import { appRoutes } from './app.routes';

@NgModule({

declarations: [

AppComponent,HomeComponent,ContactComponent,ProductComponent,ErrorComponent

],

imports: [

BrowserModule,

FormsModule,

HttpModule,

RouterModule.forRoot(appRoutes) /\*path location strategy \*/

],

providers: [ProductService],

bootstrap: [AppComponent]

})

export class AppModule { }

1. **app.component.ts**

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

@Component({

selector: 'my-app',

templateUrl: 'app/app.component.html'

})

export class AppComponent {

title = 'Routing Module Demo';

}

1. **app.component.html**

<div class="container">

<nav class="navbar navbar-default">

<div class="container-fluid">

<div class="navbar-header">

<a class="navbar-brand" [routerLink]="['/']"><strong> {{title}} </strong></a>

</div>

<ul class="nav navbar-nav">

<li><a [routerLink]="['home']">Home</a></li>

<li><a [routerLink]="['product']">Product</a></li>

<li><a [routerLink]="['contact']">Contact us</a></li>

</ul>

</div>

</nav>

<router-outlet></router-outlet>

</div>

