



Angular – Course Contents

Rakesh Singh

RAKESHSOFTNET TECHNOLOGIES Hyderabad

🌐 <http://www.rakeshsoftnet.com>

FACEBOOK <https://www.facebook.com/RakeshSoftNet>

FACEBOOK <https://www.facebook.com/RakeshSoftNetTech>

TWITTER <https://twitter.com/RakeshSoftNet>

WhatsApp +91 40 4200 8807 Telegram +91 89191 36822

Angular 2, 4, 5, 6, 7 & 8

Summary:

Angular (commonly referred to as "Angular 2+" or "Angular v2 and above including Angular 8") is a TypeScript-based (which is a superset of JavaScript) open-source front-end web application framework led by the Angular Team at Google and by a community of individuals and corporations. Angular is a complete rewrite from the same team that built AngularJS. Angular was a ground-up rewrite of AngularJS. Originally, the rewrite of AngularJS was called "Angular 2" by the team, but this led to confusion among developers. To clarify, the team announced that separate terms should be used for each framework with "AngularJS" referring to the 1.X versions and "Angular" without the "JS" referring to versions 2 and up.

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

Angular is a platform and framework for building client applications in HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.

Course Objective:

Developing modern, complex, responsive and scalable web applications with Angular.

Use their gained, deep understanding of the Angular fundamentals to quickly establish themselves as frontend developers.

Fully understand the architecture behind an Angular 2+ application and how to use it.

How to use Angular 8 (or earlier versions) to facilitate the development of single-page web applications using modern design patterns and best practices.

You will:

- Learn how to migrate to Angular 7/8 from a previous version of the framework
- Explore what's new in Angular CLI 7/8
- Learn which projects were launched alongside Angular 7/8, such as Angular Console
- Become familiar with the component CDK, which has drag and drop as well as infinite scroll support in Angular

Prerequisites of Course:

- HTML, CSS & JavaScript
- Basics of TypeScript

Course Contents:

Introduction to Angular:

- What is Angular?
- Why use Angular?
- Features & Benefits of Angular
- Angular JS (1.x) Vs Angular (2+)
- Features of Angular JS (1.x) and Angular (2+)
- What are the major differences between Angular 1.X, Angular 2 Angular 4, Angular 5, Angular 6 and Angular 7?
- Why Angular4 and Not Angular3 after Angular 2?
- jQuery Vs Angular
- Angular 2 Features
- Angular 4 Features
- Angular 5 Features
- Angular 6 Features
- Angular 7 Features
- Angular 8 Features

Working with TypeScript:

TypeScript is an open-source programming language developed and maintained by Microsoft. TypeScript lets you write JavaScript the way you really want to. TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. TypeScript is pure object oriented with classes, interfaces and statically typed like C# or Java. The popular JavaScript framework Angular 2+ is written in TypeScript.

- **Introduction to TypeScript**
- **Overview**
- **Features of TypeScript**
- **Environment Setup**
- **Basic Syntax**
- **Types**
- **Variables**
- **Operators**
- **Decision Making**
- **Loops**
- **Functions**
- **Numbers**
- **Strings**
- **Arrays**
- **Tuples**
- **Union**
- **Interfaces**
- **Classes**
- **Objects**
- **Namespaces**
- **Modules**

Angular Environment Setup:

- Node / NPM
- Git
- Yarn
- Angular CLI
- Code Editors
- IDE for writing your code
- Steps to Setup for local development environment
- Executing First Angular program using Nodejs and NPM
- Executing First Angular program using Visual Studio Code
- Getting Started with Angular 7/8

Angular Architecture: Basic Building Blocks of Angular Applications

Components: Major part of the development with Angular is done in the components. Components are basically classes that interact with the .html file of the component, which gets displayed on the browser.

Component Life Cycle

- Life Cycle Hooks

Module

- Introduction to Module
- Why use Modules
- NgModule
- Declarations
- Providers
- Imports
- Bootstrapping
- The Core Module
- Shared Modules

Data Binding:

- Binding properties and Interpolation
- One-way Binding / Property Binding
- Two-way Binding
- Two-way binding with NgModel
- Attribute Binding
- Style and Class Binding

Event Binding: When a user interacts with an application in the form of a keyboard movement, a mouse click, or a mouse over, it generates an event. These events need to be handled to perform some kind of action. This is where event binding comes into the picture.

Template:

Directives:

- Directives Introduction
- Types of Directive
- Built-In Directive
- Component Directive
- Structural Directive
- Attribute Directive
- Custom Directive

Pipes: Pipes were earlier called filters in Angular1.x and called pipes in Angular 2+.

- Predefined Pipes
 - Lowercasepipe
 - Uppercasepipe
 - Datepipe
 - Currencypipe
 - Jsonpipe
 - Percentpipe
 - Decimalpipe
 - Slicepipe
- Custom Pipes

Routing [Single Page Application]: Routing basically means navigating between pages. You have seen many sites with links that direct you to a new page. This can be achieved using routing.

- Basic Routing
- Nested Routing
- Passing Parameters
- Route Guards

Services: We might come across a situation where we need some code to be used everywhere on the page. It can be for data connection that needs to be shared across components, etc. Services help us achieve that. With services, we can access methods and properties across other components in the entire project.

- Predefined Services
- Custom Services
- Building and Injecting Custom Services
- Service using another Service
- Built-In \$http Service
- Communicating with the Server using the Http Service
- How to make Web API AJAX calls with the \$http service
- .NET Integration
- Java Integration
- PHP Integration
- Series & Parallel Service Calls
- Node Introduction

- Static Data Interaction
- MySQL/SQL Server CRUD Operations
- MongoDB CRUD Operations
- JSON Server
- Observables
- Promises
- rxjs package
- ngrx package

Dependency Injection:

- Understanding Dependency Injection
- Understanding DI in Angular Framework
- Reflective Injector
- Exploring Provider
- Types of Tokens
- Types of Dependencies
- Configuring DI using Providers
- Implementing DI in Angular
- Optional Dependencies

Forms:

- Template Driven Forms(TDF)
- Model Driven Forms(MDF)
- Form Validation

Communication between Components

- @Input()
- @Output()
- @ViewChild()
- @ViewChildren()

Angular Cookies

Angular Animations

Angular Materials

Angular CLI

Real Time Case Study Examples

Interview Q & A Discussion

Mean Stack Development

Microsoft .Net Solution

Develop Your **Microsoft** Skills and Knowledge through great Training



Hyderabad's Best Institute for .NET

Angular



Rakesh Singh

RAKESHSOFTNET TECHNOLOGIES Hyderabad

🌐 <http://www.rakeshsoftnet.com>

FACEBOOK <https://www.facebook.com/RakeshSoftNet>

FACEBOOK <https://www.facebook.com/RakeshSoftNetTech>

TWITTER <https://twitter.com/RakeshSoftNet>

CALL +91 40 4200 8807 MOBILE +91 89191 36822



What is Angular?

The Angular is the newest form of the AngularJS, developed and maintained by the angular team at Google and the Father of Angular is Misko Hevery, which is an open-source front-end development platform used for building mobile and desktop web applications.

Angular is an open source JavaScript framework for building web applications and apps in JavaScript, html, and Typescript which is a superset of JavaScript and makes you able to create reactive **Single Page Applications** (SPAs). This is a leading front-end development framework which is regularly updated by Angular team of Google. The Angular now comes with every latest feature you need to build a complex and sophisticated web or mobile application. Angular provides built-in features for Component, Directives, Data Binding, Event Binding, Property Binding, Modules, Templates, Forms, Pipes, Http Services, Dependency Injection, Routing, Animations and Materials which in turn have features such as auto-complete, navigation, toolbar, menus, etc. The code is written in Typescript, which compiles to JavaScript and displays the same in the browser. Angular is completely based on components. It consists of several components forming a tree structure with parent and child components. Angular's versions beyond 2+ are generally known as **Angular** only. The very first version Angular 1.0 is known as **AngularJS**. Angular is rewritten by the same team that built AngularJS.

Angular is the most popular web development framework for developing web, mobile web, native mobile and native desktop applications. The angular framework is also utilized in the cross-platform mobile development called IONIC and so it is not limited to web apps only.

Angular is written in TypeScript and so it comes with all the capabilities that typescript offers.

Conclusion Definition:

Angular (commonly referred to as "**Angular 2+**" or "**Angular v2 and above**") is a TypeScript-based open-source web application framework led by the Angular Team at Google and by a community of individuals and corporations. Angular is a complete rewrite from the same team that built AngularJS.



One framework. Mobile & desktop.

Developer: Google

Initial release: 2.0 (14 September 2016)

Stable release: 8.0.0 (28 May 2019)

Repository: <https://github.com/angular/angular>

Written in: TypeScript

Platform: Web platform

Type: Web framework

Website: <https://angular.io/>

Latest Stable Version Angular is 8.2.8 (25th September 2019)

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



What is Single Page Application (SPA)?

A single page application is a web application or a website which provides users a very fluid, reactive and fast experience similar to a desktop application. It contains menu, buttons and blocks on a single page and when a user clicks on any of them; it dynamically rewrites the current page rather than loading entire new pages from a server. That's the reason behind its reactive fast speed.

Why Angular?

There are many front-end JavaScript frameworks to choose from today, each with its own set of trade-offs. Many people were happy with the functionality that Angular 1.x afforded them. Angular 2+ improved on that functionality and made it faster, more scalable and more modern. Organizations that found value in Angular 1.x will find more value in Angular 2+.

The powerful features and capabilities of Angular permit you to build complex, customizable, modern, responsive and user-friendly web applications. It also enables you to create software quicker and with less effort.

As your application grows, structuring your code in a clean and maintainable and more importantly, testable way, becomes more complex. But your life becomes far easier using a framework like Angular.

Angular is faster than AngularJS and offers a much more flexible and modular development approach. Due to the drastic change between Angular 1.x and Angular 2+ you don't need to have knowledge about AngularJS.

Angular 2+ Is Easier

TypeScript based

Develop Across All Platforms

Speed & Performance

Incredible Tooling Support

Familiarity

Project Architecture and Maintenance

Loved By Millions

Prerequisites:

Before proceeding with Angular, you should have a basic understanding of following:

- HTML,
- Document Object Model (DOM),
- CSS,
- JavaScript, and
- Typescript



Difference between AngularJS and Angular:

"Angular was a ground-up rewrite of AngularJS."

AngularJS is common and popular name of the first version of Angular1.0.

Angular is common and popular name of the Angular's version beyond 2+

AngularJS is a JavaScript-based open-source front-end web framework.

Angular is a TypeScript-based open-source full-stack web application framework.

Angular JS is written in JavaScript

Angular uses Microsoft's TypeScript Language, which is a superset of JavaScript. TypeScript that have powerful "type checking" support and object-oriented features.

AngularJS is based on the model view controller.

The structure of Angular is based on the components/services architecture.

AngularJS uses the concept of scope or controller.

Instead of scope and controller, Angular uses hierarchy of components as its primary architectural characteristic.

AngularJS has a simple syntax and used on HTML pages along with the source location.

Angular uses the different expression syntax. It uses "[]" for property binding, and "()" for event binding.

AngularJS is a simple JavaScript file which is used with HTML pages and doesn't support the features of a server-side programming language.

Angular uses of Microsoft's TypeScript language, which provides Class-based Object Oriented Programming, Static Typing, Generics etc. which are the features of a server-side programming language.

AngularJS doesn't support dynamic loading of the page.

Angular supports dynamic loading of the page.

Angular has its own suite of modern UI components that work across the web, mobile and desktop, called Angular Material

Angular JS was not built with Mobile support in mind, but Angular has mobile support.

Angular JS uses \$routeProvider.when() to configure routing while Angular uses @RouteConfig{...} }

Filters (In Angular JS) have been now renamed with "pipes" in Angular.

Angular 2 is 5 times faster than Angular JS



TypeScript Language Used:

TypeScript is used as the programming language. If you have knowledge of Java or C# then you will find it very easy.

TypeScript is the key language used by the official Angular team and the language which you'll mostly see in Angular development. It's a superset to JavaScript and makes Angular apps coding easy and simple. For creating Angular apps it ensures that you will have the best possible development.

What you'll do?

Develop a modern, complex, responsive and scalable web application with Angular 2+ to 7/8.

Fully understand the architecture behind an Angular application and how to use it.

Use their gained, deep understanding of the Angular fundamentals to quickly establish themselves as frontend developers.

Create single-page applications with one of the most modern JavaScript frameworks.

Angular Features:

A list of most important features and benefits of Angular:

Angular supports multiple platforms:

Angular is a cross platform. It supports multiple platforms. You can build different types of apps by using Angular.

- **Desktop applications:** Angular facilitates you to create desktop installed apps on different types of operating systems i.e. Windows, Mac or Linux by using the same Angular methods which we use for creating web and native apps.
- **Native applications:** You can build native apps by using Angular with strategies from Cordova, Ionic, or NativeScript.
- **Progressive web applications:** Progressive web applications are the most common apps which are built with Angular. Angular provides modern web platform capabilities to deliver high performance, offline, and zero-step installation apps.

High Speed, Ultimate Performance:

Angular is amazingly fast and provides a great performance due to the following reasons:

- **Universal support:** Angular can be used as a front-end web development tool for the programming languages like Node.js, .Net, PHP, Java Struts and Spring and other servers for near-instant rendering in just HTML and CSS. It also optimizes the website for better SEO.
- **Code splitting:** Angular apps are fast and loads quickly with the new Component Router, which delivers automatic code-splitting so users only load code required to render the view they request.
- **Code generation:** Angular makes your templates in highly optimized code for today's JavaScript virtual machines which gives the benefits of hand-written code.

Productivity:

Angular provides a better productivity due to its simple and powerful template syntax, command line tools and popular editors and IDEs.

- **Powerful templates:** Angular provides simple and powerful template syntax to create UI view quickly.
- **IDEs:** Angular provides intelligent code completion, instant errors, and other feedback in popular editors and IDEs.
- **Angular CLI:** Angular CLI provides command line tools start building fast, add components and tests, and then instantly deploy.

Full Stack Development

Angular is a complete framework of JavaScript. It provides Testing, animation and Accessibility. It provides full stack development along with Node.js, Express.js, and MongoDB.

- **Testing:** Angular provides Karma and Jasmine for unit testing. By using it, you can check your broken things every time you save. Karma is a JavaScript test runner tool created by Angular team. Jasmine is the testing framework for unit testing in Angular apps, and Karma provides helpful tools that make it easier to us to call our Jasmine tests whilst we are writing code.
- **Animation Support:** Angular facilitates you to create high-performance, complex choreographies and animation timelines with very little code through Angular's intuitive API.
- **Accessibility:** In Angular, you can create accessible applications with ARIA (Accessible Rich Internet Applications)-enabled components, developer guides, and built-in a11y test infrastructure.

History:

Naming:

Originally, the rewrite of AngularJS was called "Angular 2" by the team, but this led to confusion among developers. To clarify, the team announced that separate terms should be used for each framework with "AngularJS" referring to the 1.X versions and "Angular" without the "JS" referring to versions 2 and up.

Different versions of Angular

The first version of Angular was Angular 1.0 (also known as AngularJS) which was released in 2010. But here, we are talking about Angular so; let's see history and different versions of Angular.

The Early version of the Angular was named as **Angular 2**. Then later it was renamed to just "**Angular**". Angular Team releases new versions of the Angular regularly and the latest version is available **Angular 8.2.2 (12th August 2019)**.

Angular2

Angular 2.0 was first introduced in October 2014. It was a complete rewrite of Angular so, the drastic changes in the 2.0 version created controversy among developers. On April 30, 2015, the Angular developers announced that Angular 2 moved from Alpha to Developer Preview and then Beta version was released in December 2015. Its first version was published in May 2016 and the final version was released on September 14, 2016.

The core differences and many more advantages on Angular 2 vs. Angular 1 as following,

- It is entirely component based.
- Better change detection
- Angular2 has better performance.
- Angular2 has more powerful template system.
- Angular2 provide simpler APIs, lazy loading and easier to application debugging.
- Angular2 is much more testable.
- Angular2 provides to nested level components.
- Ahead of Time compilation (AOT) improves rendering speed
- Angular2 execute run more than two programs at the same time.
- Angular1 is controllers and \$scope based but Angular2 is component based.
- The Angular2 structural directives syntax is changed like ng-repeat is replaced with *ngFor etc.
- In Angular2, local variables are defined using prefix (#) hash. You can see the below *ngFor loop Example.
- TypeScript can be used for developing Angular 2 applications
- Better syntax and application structure

There are more advantages over performance, template system, application debugging, testing, components and nested level components.

For Examples as,

Angular 1 Controller:-

```
var app = angular.module("userApp", []);
app.controller("productController", function($scope) {
  $scope.users = [{ name: "Anil Singh", Age:30, department :"IT" },
  { name: "Aradhyaa Singh", Age:3, department :"MGMT" }];
});
```

Angular 2 Components using TypeScript:-

Here the @Component annotation is used to add the metadata to the class.

```
import { Component } from 'angular2/core';
@Component({
  selector: 'usersdata',
  template: `<h3>{{users.name}}</h3>`
})

export class UsersComponent {
  users = [{ name: "Anil Singh", Age:30, department :"IT" },
  { name: "Aradhyaa Singh", Age:3, department :"MGMT" }];
}
```

Bootstrapping in Angular 1 using ng-app,

```
angular.element(document).ready(function() {
  angular.bootstrap(document, ['userApp']);
});
```

Bootstrapping in Angular 2,

```
import { bootstrap } from 'angular2/platform/browser';
import { UsersComponent } from './product.component';

bootstrap(UsersComponent);
```

The Angular2 structural directives syntax is changed like **ng-repeat** is replaced with ***ngFor** etc.

For example as,

```
//Angular 1,
<div ng-repeat="user in users">
  Name: {{user.name}}, Age : {{user.Age}}, Dept: {{user.Department}}
</div>
```

```
//Angular2,
<div *ngFor="let user of users">
  Name: {{user.name}}, Age : {{user.Age}}, Dept: {{user.Department}}
</div>
```

Angular4

Angular 4 version was announced on 13 December 2016. The developers skipped the version 3 to avoid a confusion due to the misalignment of the router package's version which was already distributed as v3.3.0. The final version was released on March 23, 2017. Angular 4 is backward compatible with Angular 2.

Angular 4 contains some additional enhancement and improvement. Consider the following enhancements.

- Smaller & Faster Apps
- View Engine Size Reduce
- Animation Package
- NgIf and ngFor Improvement
- Template
- NgIf with Else
- Use of AS keyword
- Pipes
- HTTP Request Simplified
- Apps Testing Simplified
- Introduce Meta Tags
- Added some Forms Validators Attributes
- Added Compare Select Options
- Enhancement in Router
- Added Optional Parameter
- Improvement Internationalization

1. **Smaller & Faster Apps** - Angular 4 applications is smaller & faster in comparison with Angular
2. **View Engine Size Reduce** - Some changes under to hood to what AOT generated code compilation that means in Angular 4, improved the compilation time. These changes reduce around 60% size in most cases.
3. **Animation Package**- Animations now have their own package i.e. @angular/platform-browser/animations
4. **Improvement** - Some Improvement on *ngIf and *ngFor.
5. **Template** - The template is now ng-template. You should use the “ng-template” tag instead of “template”. Now Angular has its own template tag that is called “ng-template”.
6. **NgIf with Else** – Now in Angular 4, possible to use an else syntax as,

```
<div *ngIf="user.length > 0; else empty"><h2>Users</h2></div>
<ng-template #empty><h2>No users.</h2></ng-template>
```

7. **AS Keyword** – A new addition to the template syntax is the “as keyword” is use to simplify to the “let” syntax.
Use of as keyword,

```
<div *ngFor="let user of users | slice:0:2 as total; index as = i">
  {{i+1}}/{{total.length}}: {{user.name}}
</div>
```

To subscribe only once to a pipe “|” with “**async**” and If a user is an observable, you can now use to write,

```
<div *ngIf="users | async as usersModel">
  <h2>{{ usersModel.name }}</h2> <small>{{ usersModel.age }}</small>
</div>
```

8. **Pipes** - Angular 4 introduced a new “**titlecase**” pipe “|” and use to changes the first letter of each word into the uppercase.

The example as,

```
<h2>{{ 'mahesh kumar' | titlecase }}</h2>
<!-- OUPPUT - It will display 'Mahesh Kumar' -->
```

9. **Http** - Adding search parameters to an “**HTTP request**” has been simplified as,

```
//Angular 4 -
http.get('${baseUrl}/api/users', { params: { sort: 'ascending' } });
//Angular 2-
const params = new URLSearchParams();
params.append('sort', 'ascending');
http.get('${baseUrl}/api/users', { search: params });
```

10. **Test**- Angular 4, overriding a template in a test has also been simplified as,

```
//Angular 4 -  
 TestBed.overrideTemplate(UsersComponent, '<h2>{{users.name}}</h2>');  
//Angular 2 -  
 TestBed.overrideComponent(UsersComponent, {  
  set: { template: '<h2>{{users.name}}</h2>' }  
});
```

11. **Service**- A new service has been introduced to easily get or update “**Meta Tags**” i.e.

```
@Component({  
  selector: 'users-app',  
  template: `<h1>Users</h1>  
`})  
export class UsersAppComponent {  
  constructor(meta: Meta) {  
    meta.addTag({ name: 'Blogger', content: David Smith' });  
  }  
}
```

12. **Forms Validators** - One new validator joins the existing “required”, “minLength”, “maxLength” and “pattern”. An email helps you validate that the input is a valid email.

13. **Compare Select Options** - A new “**compareWith**” directive has been added and it used to help you compare options from a select.

```
<select [compareWith]="byUid" [(ngModel)]="selectedUser">  
  <option *ngFor="let user of users" [ngValue]="user.UId">{{user.name}}</option>  
</select>
```

14. **Router** - A new interface “**paramMap**” and “**queryParamMap**” has been added and it introduced to represent the parameters of a URL.

```
const uid = this.route.snapshot.paramMap.get('UId');  
this.userService.get(uid).subscribe(user => this.name = name);
```

15. **CanDeactivate** - This “**CanDeactivate**” interface now has an extra (optional) parameter and it is containing the next state.

16. **I18n** - The internationalization is tiny improvement.

```
//Angular 4-  
<div [ngPlural]="value">  
  <ng-template ngPluralCase="0">there is nothing</ng-template>  
  <ng-template ngPluralCase="1">there is one</ng-template>  
</div>  
//Angular 2-  
<div [ngPlural]="value">  
  <ng-template ngPluralCase="=0">there is nothing</ng-template>  
  <ng-template ngPluralCase="=1">there is one</ng-template>  
</div>
```

Angular5

This version was released on 1 Nov, 2017. It provided some improvements to support for progressive web apps, also provides improvements related to Material Design.

It comes a few new features as well as a number of internal changes to make Angular apps smaller and faster to execute.

- Performance

< 20%

- Progressive Web Application Support

- Build Optimizer:

Projects built using Angular CLI will now apply build optimization by default leading to decreased size of JavaScript files with better performance.

- TypeScript Update:

TypeScript 2.4 support in Angular 5

- String-based enums are a new feature introduced in TypeScript 2.4.
- The newer version of TypeScript improves type checking with regards to generics.
- Weak-Type-Detection.

- Compiler Improvements

- Incremental builds:-

- In the previous versions of Angular every time we do a build it builds from scratch, you either change or do no change the code. From Angular 5 it will use typescript transform which was introduced in Typescript 2.3. This will ensure speedy builds of the project.

- Preserve Whitespaces:

- **Removing White spaces:** - Previously white spaces, tabs and so on were part of build output. That can now be removed from the project final build output by providing "preserveWhitespaces" as true in the tsconfig.json file.

- **Router Life cycle events:** - Routing is one of the important part of Angular to create SPA application. In routing we have something called as Route guards. In Angular 5 they have defined routing life cycle which helps to track when router moves from guards to completion state.

The routing life cycle events fire in the following sequence:-

- GuardsCheckStart
- ChildActivationStart
- ActivationStart
- GuardsCheckEnd
- ResolveStart
- ResolveEnd
- ActivationEnd
- ChildActivationEnd

- Multiple Export Alias

- HttpClient Update

- RxJS Update

- New Pipes

- Forms Update

- Animations Update

Angular6

This version was released on 4 may, 2018. It was a major release which provides some features like: ng update, ng add, Angular Elements, Angular Material + CDK Components, Angular Material Starter Components, CLI Workspaces, Library Support, Tree Shakeable Providers, Animations Performance Improvements, and RxJS v6.

Angular 6 being smaller, faster and easier to use and it will make developers life easier.

Added ng update - This CLI commands will update your angular project dependencies to their latest versions. The ng update is normal package manager tools to identify and update other dependencies.

`ng update`

Angular 6 uses RxJS 6 - this is the third-party library (RxJS) and introduces two important changes as compared to RxJS 5.

1. RxJS 6 introduces a new internal package structure
2. Operator concept

Both are requires you to update your existing code

To update to RxJS 6, you simply run -

`npm install --save rxjs@6`

Simply run the blow command and update your existing Angular project-

`npm install --save rxjs-compat`

Alternatively, you can use the command - `ng update rxjs` to update RxJS and install the rxjs-compat package automatically.

RxJS 6 Related import paths -

Instead of -

```
import { Observable } from 'rxjs/Observable';
import { Subject } from 'rxjs/Subject';
```

Use a single import -

```
import { Observable, Subject } from 'rxjs';
```

So all from `rxjs/something` imports become from one '`rxjs`'

Operator imports have to change -

Instead of

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/throttle';
```

Now you can use -

```
import { map, throttle } from 'rxjs/operators';
```

And



Instead of

```
import 'rxjs/add/observable/of';
```

Now you can use -

```
import { of } from 'rxjs';
```

RxJS 6 Changes - Changed Operator Usage

Instead of-

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/throttle';
yourObservable.map(data => data * 2)
.throttle(...)
.subscribe(...);
```

You can use the new pipe () method,

```
import { map, throttle } from 'rxjs/operators';
yourObservable
.pipe(map(data => data * 2), throttle(...))
.subscribe(...);
```

CLI update and added a new project config file - Instead of ".angular-cli.json" using "angular.json". Now in Angular 6 new projects use an "angular.json" file instead of ".angular-cli.json" file.

```
ng update @angular/cli --from=1 --migrate-only
```

The above command helps you to update your existing ".angular-cli.json" file to the new "angular.json" file.

The "angular.json" file contains the Properties -

1. **Version** - This is integer file format version and it is currently 1.
2. **newProjectRoot** - This is string path where new projects will be created.
3. **defaultProject** - This is default project name used in commands.
4. **CLI** - This is workspace configuration options for Angular CLI and it contains
 - defaultCollection
 - packageManager
 - warnings
 - And so on.
5. **Schematics** - This is configuration options for Schematics.
6. **Projects** - This is configuration options for each project in the workspace and it contains
 - root
 - sourceRoot
 - projectType
 - prefix
 - schematics
 - Architect - This is the project configuration for Architect targets.

The `<template>` deprecated, Now Angular 6 introduce `<ng-template>` –

Now in Angular 6, you should use `<ng-template>` instead of `<template>`

For example, previously you are using

```
<template [ngIf]="isAdmin">
  <p>This template renders only if isAdmin is true.</p>
</template>
```

Now in Angular 6, you should use `<ng-template>` instead of `<template>`

```
<ng-template [ngIf]="isAdmin">
  <p>This template renders only if isAdmin is true.</p>
</ng-template>
```

Service level changes (the way of marking a service as global) -

In the earlier versions, if you want to provide a service to the entire application – you should add it to `providers []` in the `AppModule` but in the Angular 6 released you should not add in the `providers []` in the `AppModule`.

Example for marking a service as global -

Instead of

```
//my.service.ts
export class MyService { }

//In app.module.ts
//JavaScript imports services
import { MyService } from './my-service.service';
//AppModule class with the @NgModule decorator
@NgModule({
  declarations: [],
  providers: [MyService] //My services instances are now available across the entire app.
})
export class AppModule {
  //exporting app module
}
```

Use with Angular 6 released-

```
//my.service.ts

@Injectable({providedIn: 'root'})

export class MyService { }

@NgModule({
  declarations: [],
  providers: [] // Service does not need to be added here
})

export class AppModule {}
```

The second one obviously saves you some lines of code as compare to previous code.

Angular 6 introduces Angular Elements -

The elements are a feature that allows you to compile Angular components to native web components which you can use in your Angular application.

An angular element is a package which is part of the Angular framework @angular/elements.

Angular 6 introduces new Ivy Renderer -

The new Ivy renders and it's not stable for now and it's only in beta version. It will be stable in future for production.

Ivy Renderer is new rendering engine which is designed to be backward compatible with existing render and focused to improve the speed of rendering and it optimizes the size of the final package.

The main goal of Ivy render is to speed up its loading time and reduce the bundle size of your applications. Also for uses a different approach for rendering Angular components.

For Angular, this will not be default renderer, but you can manually enable it in compiler options.

Bazel Compiler -

The Bazel Complier is a build system used for nearly all software built at Google.

From Angular 6 release, will start having the Bazel compiler support and when you compile the code with Bazel Compiler, you will recompile entire code base, but it compiles only with necessary code.

The Bazel Complier uses advanced local and distributed caching, optimized dependency analysis and parallel execution.

Replace Context, Record and Injectors -

Replace ngOutletContext with ngTemplateOutletContext

Replace CollectionChangeRecord with IterableChangeRecord

Now use Renderer2, Instead of Renderer

Now use StaticInjector, Instead of ReflectiveInjector,

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



Angular 6 Renamed Operators -

The lists of renamed operators are –

- do() => tap()
- catch() => catchError()
- finally() => finalize()
- switch() => switchAll()
- throw() => throwError()
- fromPromise() => from()

Angular 6 introduces multiple validators for array method of FormBuilder –

```
import { Component } from '@angular/core';
import { FormsModule, FormBuilder, FormGroup } from '@angular/forms';
constructor(private fb: FormBuilder) {}
myForm: FormGroup;
ngOnInit() {
  this.myForm = this.fb.group({
    text: ['', Validators.required],
    options: this.fb.array([], [MyValidators.minCount, MyValidators.maxCount])
  });
}
```

Addition of navigationSource and restoredState to NavigationStart -

These two properties help us to handle multiple use cases in routing.

NgModelChange - Now emitted after value and validity is updated on its control. Previously, it was emitted before updated.

As the updated value of the control is available, the handler will become more powerful

Previously -

```
<input [(ngModel)]="name" (ngModelChange)="onChange($event)">
```

And

```
onChange(value) {
  console.log(value); // would log the updated value, not old value
}
```

Now Use -

```
<input #modelDir="ngModel" [(ngModel)]="name" (ngModelChange)="onChange(modelDir)">
```

And

```
onChange(NgModel: NgModel) {
  console.log(NgModel.value); // would log old value, not updated value
}
```

Form Control statusChanges – Angular 6 emits an event of “PENDING” when we call Abstract Control markAsPending.

New optional generic type ElementRef – This optional generic type will help to get hold of the native element of given custom Element as ElementRef Type

Angular 7

Angular 7 being smaller, faster and easier to use and it will making developers life easier. Angular 7 was released on October 18, 2018. Updates regarding Application Performance, Angular Material & CDK, Virtual Scrolling, Drag and Drop, Improved Accessibility of Selects, now supports Content Projection using web standard for custom elements, and dependency updates regarding Typescript 3.1, RxJS 6.3, and Node 10 (still supporting Node 8). So, it consists of many extensive features:

- **Updates regarding Application Performance**
- **CLI Prompts:** Angular CLI has updated to v7.0.2 added some features like now it will prompt users while typing common commands like ng-add or ng-new, @angular/material to help you discover built-in features like routing or SCSS support. With Angular 7, while creating new projects it takes advantage of Bundle Budgets in CLI.
- **Angular Material & CDK:** The version of Angular Material/CDK is updated in Angular 7. Also there are 2 features added to CDK – **virtual scrolling, and drag and drop.**
 - **Virtual Scrolling:** Virtual scrolling feature shows up the visible DOM elements to the user, as the user scrolls, the next list is displayed. This gives faster experience as the full list is not loaded at one go and only loaded as per the visibility on the screen.
 - **Drag and Drop:** You can drag and drop elements from a list and place it wherever required within the list. The new feature is very smooth and fast.
- **Improved Accessibility of Selects**
- **Angular Elements**
 - **Supports Content Projection using web standard for custom elements**
- **Angular Do-Bootstrap:** In the past, Angular has used for bootstrapping modules that need to bootstrap a component. Angular 7 added a new life-cycle hook (**ngDoBootstrap**) and interface (**DoBootstrap**).

Example:

```
class AppModule implements DoBootstrap {
  ngDoBootstrap(appRef: ApplicationRef) {
    appRef.bootstrap(AppComponent);
  }
}
```

- **Updated Dependencies in Angular 7:** Angular 7 comes with support for various upgrades in dependencies.
 - **Typescript 3.1:** Angular 7 have updated TypeScript version from 2.7 to 3.1 which is it's the latest release. It's compulsory to use Typescript's latest version while working with Angular 7.
 - **RxJs 6.3:** The latest version of RxJs (version 6.3.3) was added in Angular 7, bringing with it new, exciting additions and changes.
 - **Support for Node v10:** Angular 7 now supports Node V10 with backward compatibility, as well.
 - **Better Error Handling:** Angular 7 improves error handling in an angular application. @Output in angular7.0 has an improved error handling feature.

- **Bundle Budget:** While developing applications on Angular 7, now the developers can set up a budget limit of their bundle size. A default setting of the bundled budget has 2 MB as the lower limit and 5 MB as the higher limit. When the initial bundle is more than 2MB, a new application will warn and will error at 5 MB. The developer can also change these settings as per need. Reduction in bundle size improves the performance of the application.
- **Native Script:** Before Angular 7 developers have to create separate projects for mobile and web versions of the application but now through a single project, users can build a web and mobile app too. A native script schematics collection provide this functionality. The codes for the web and mobile apps will maintain in such a way that the shareable part keep at one place and non-shareable ones can create separately but in a single project.
- **A New ng-compiler:** Angular 7 added a new compiler called the Angular Compatibility Compiler (ngcc). The ngcc Angular node_module compatibility compiler - The ngcc is a tool which "upgrades" node_module compiled with non-ivy ngc into ivy compliant format. This compiler will convert node_modules compiled with Angular Compatibility Compiler (ngcc), into node_modules which appear to have been compiled with TSC compiler transformer (ngtsc) and this compiler conversions will allow such "legacy" packages to be used by the Ivy rendering engine.
- **Ivy Renderer Progress:** According to the official information, Angular's new project and the next gen rendering pipeline Ivy, hasn't been introduced with the release of Angular 7 as the team is still working on it. The current status is that it is under the active deployment process.
- **Documentation Updates:** The team has been continuously working on improving the guidelines and reference materials to serve the developers better. The updates related to documentation on angular is one such step including the reference material for the Angular CLI.
- **How to update to Angular 7:** If you are already running your Angular App on Angular 6 & RXJS 6, just update your @angular cli/core and also update your angular material.
`> ng update @angularcli @angular/core`
`> ng update @angular/material`

Or Visit update.angular.io for detailed information and guidance on updating your application. Developers have reported that Angular 7 update is faster than ever, and many apps take less than 10 minutes to update.

Angular 8:

Angular 8 is an open-source, client-side TypeScript based JavaScript framework. It is written in TypeScript and complied into JavaScript. Angular 8 is used to create **dynamic web applications**. It is very similar to its previous versions except having some extensive features. Angular 8 being smaller, faster and easier to use and it will make Angular developers life easier. Angular 8 was released on May 28, 2019. Featuring Differential loading for all application code, Dynamic imports for lazy routes, Web workers, TypeScript 3.4 support, and Angular Ivy as an opt-in preview.

- Differential loading
- Dynamic imports for lazy routes
- Web workers
- TypeScript 3.4 Support
- Angular Ivy
- Bazel Support

What is a dynamic web application?

A dynamic web application is simply a dynamic website which has a tendency to change data/information with respect to 3 things:

- Time-to-time (e.g. News update webs applications)
- Location-to-location (e.g. Weather-report web applications)
- User-to-user (e.g. Gmail, Facebook type applications)

Future Releases:

Each version is expected to be backward-compatible with the prior releases of Angular.

Google pledged to do twice a year upgrades. So, in 2019 two version are about to release i.e. Angular 8 in April/May and Angular 9 in September/October Month

Angular 8 is already released on 28th May 2019.

How to install Angular 7/8?

Angular 7/8 Environment Setup

In this document, you will see how you can install the prerequisites needed to run your first Angular 7/8 app.

We will discuss the Environment Setup required for Angular 7/8. To install Angular 7/8, we require the following -

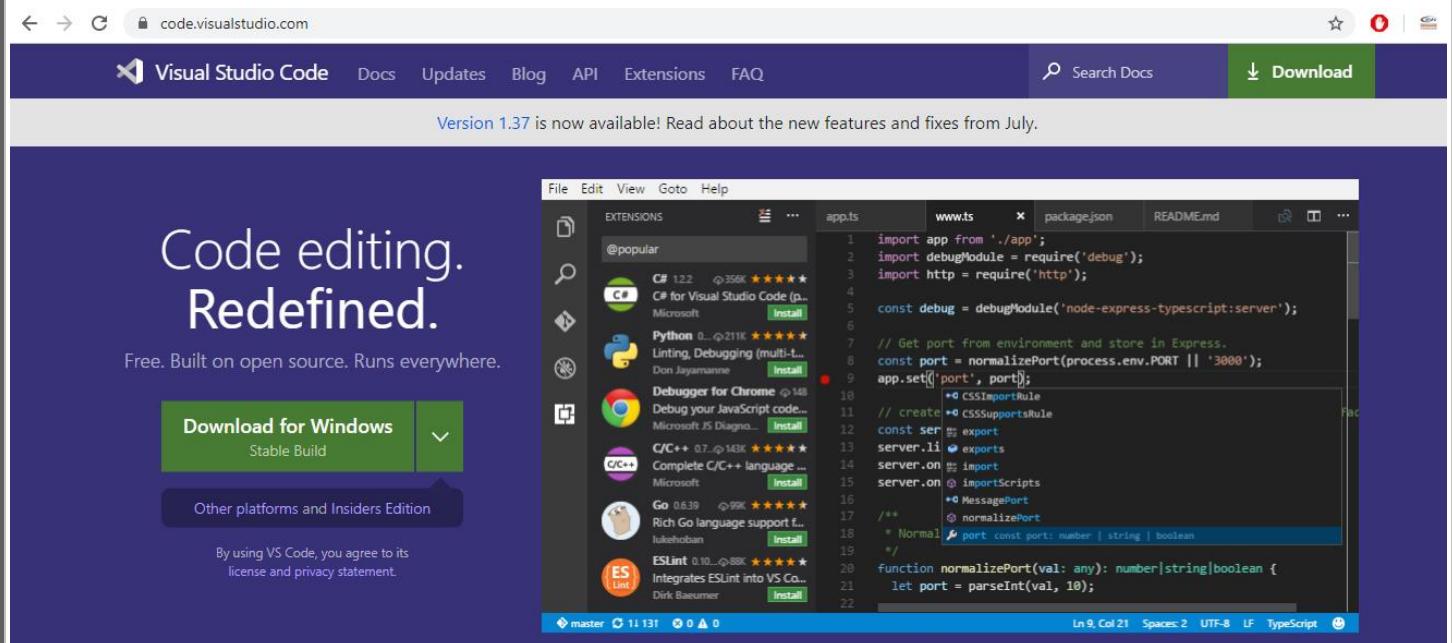
- IDE for writing your code
- Nodejs
- Npm
- Angular CLI

IDE for writing your code:

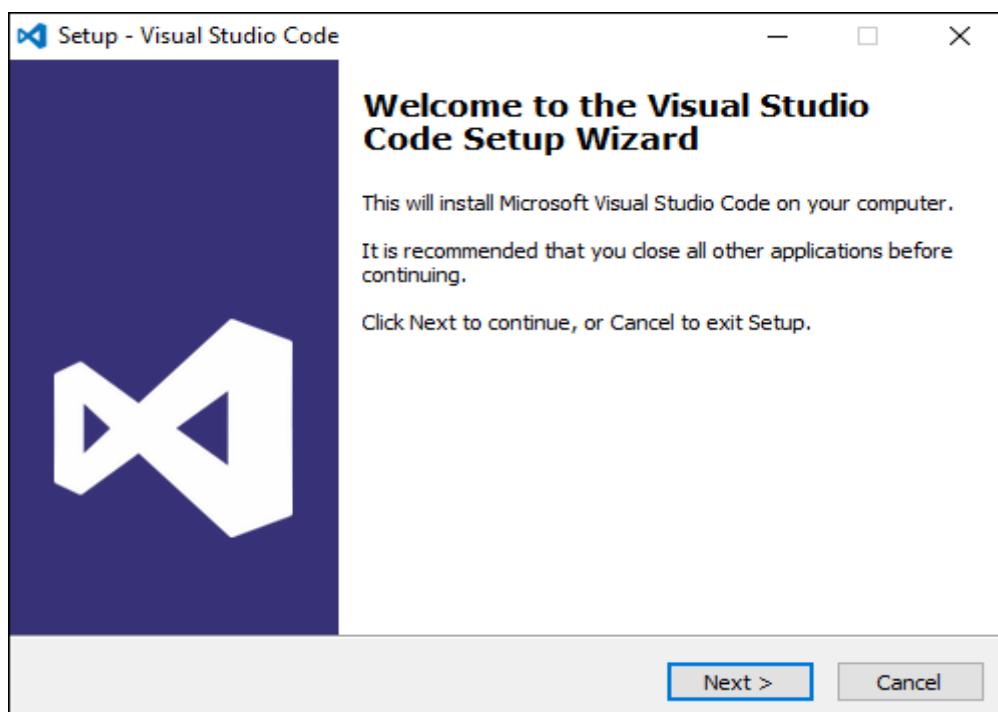
There are many IDE that can be used for Angular 7/8 development such as **Visual Studio Code** and **WebStorm**. In this document, we will use the Visual Studio Code, which is free from Microsoft.

VS Code is light and easy to setup, it has a great range of built-in code editing, formatting, and refactoring features. It is free to use. It also provides a huge number of extensions that will significantly increase your productivity.

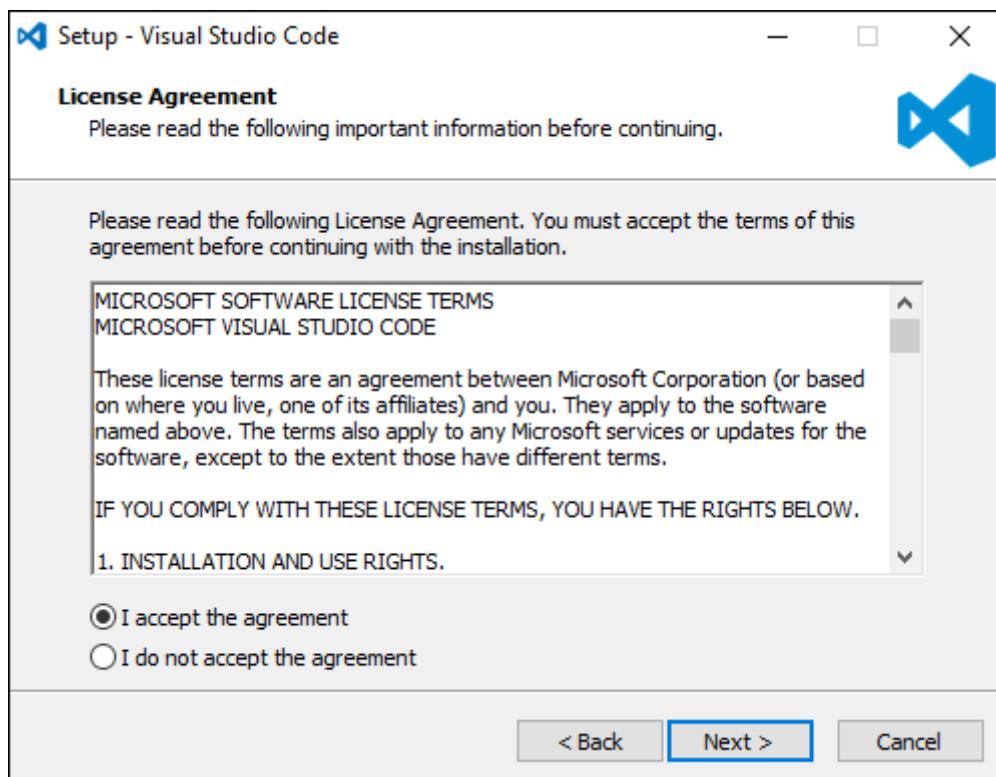
You can download VS Code from official site of Visual Studio Code: <https://code.visualstudio.com>



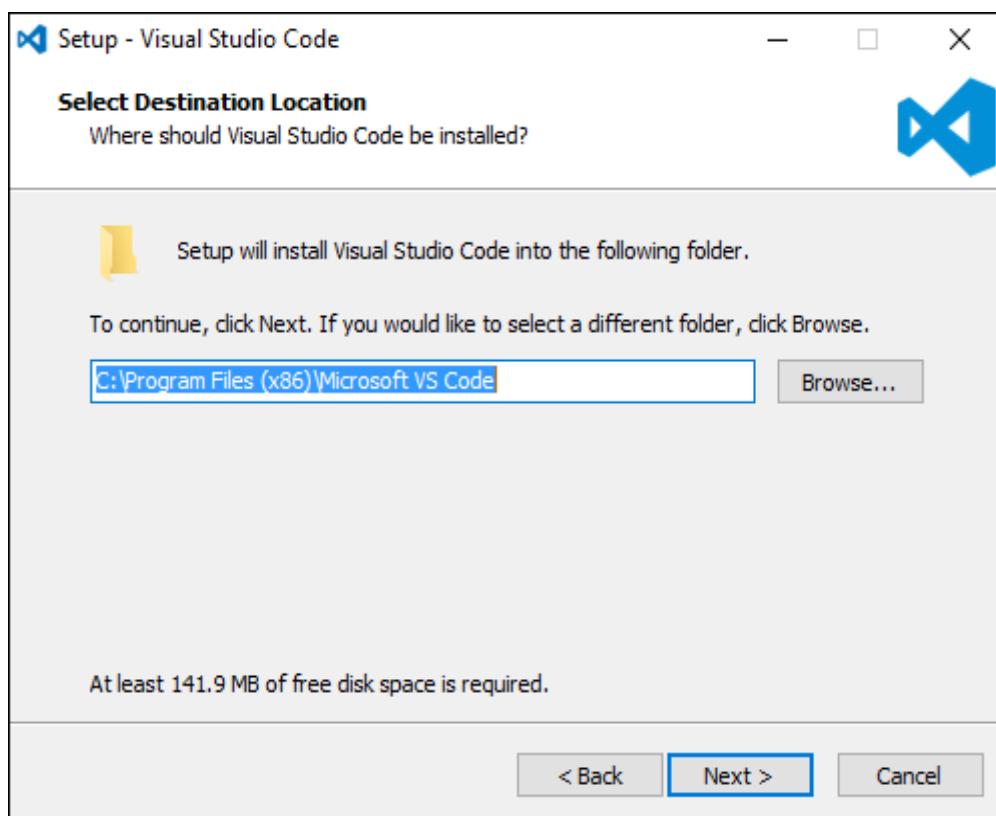
Step 1 – Follow the installation steps after the download is completed. In the initial screen, click the Next button.



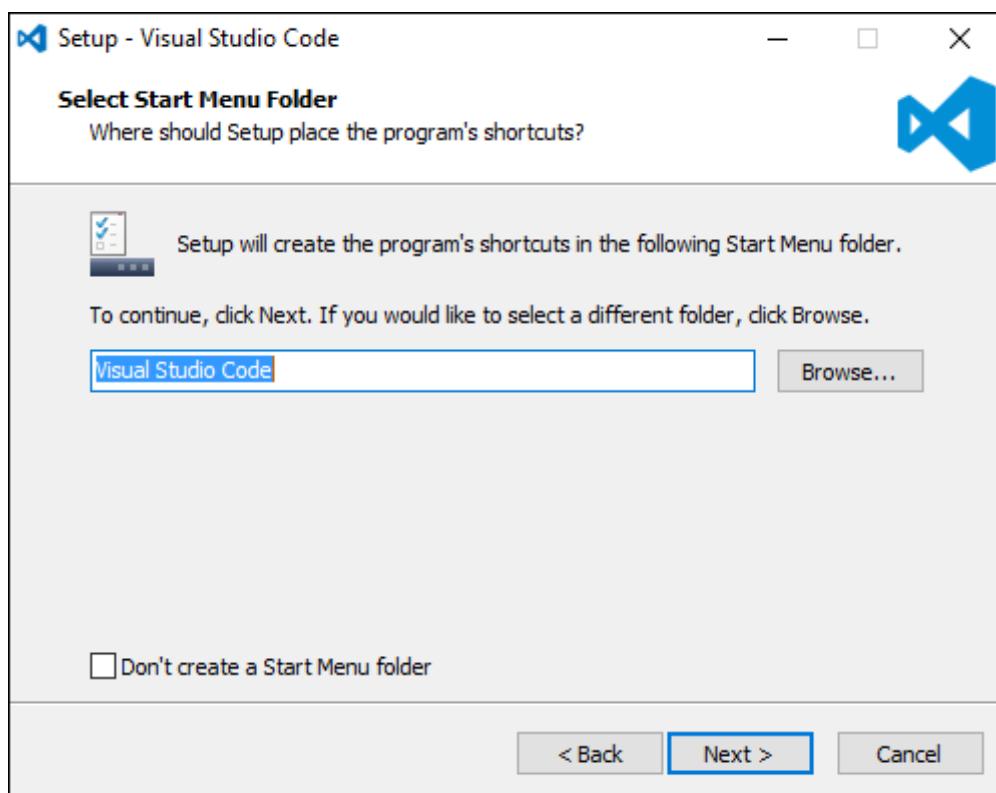
Step 2 – on the next screen, accept the license agreement and click on the Next button.



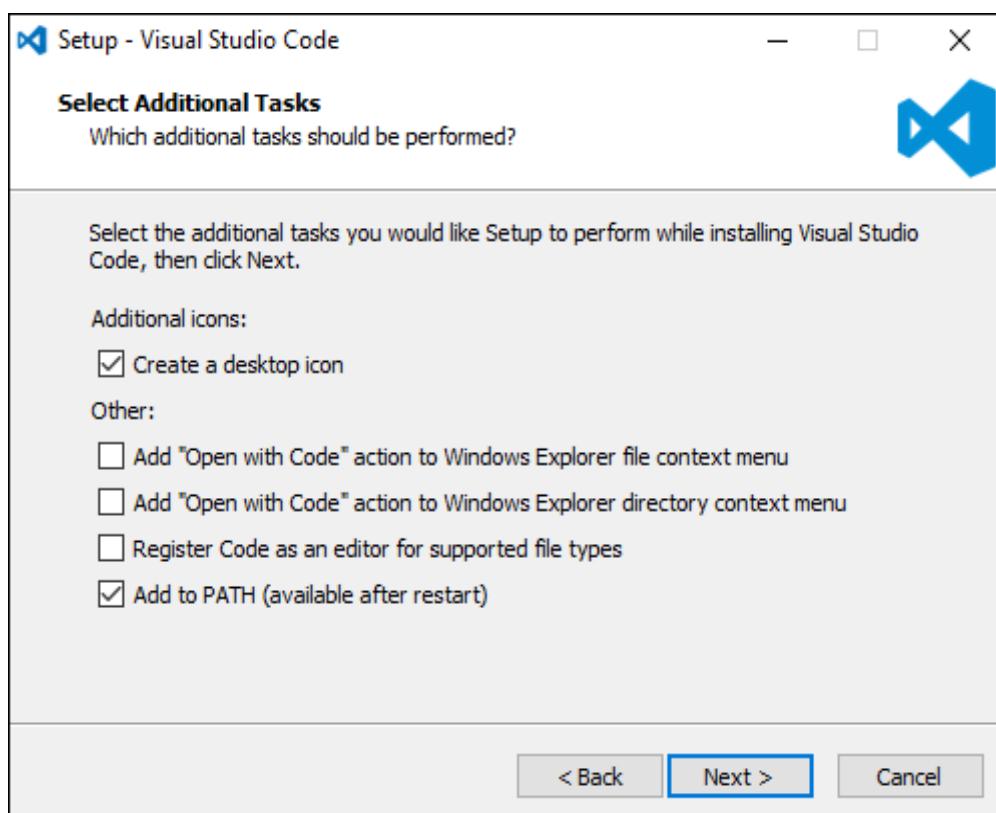
Step 3 – on the next screen, choose the destination location for the installation of Visual Studio Code and click on the next button.



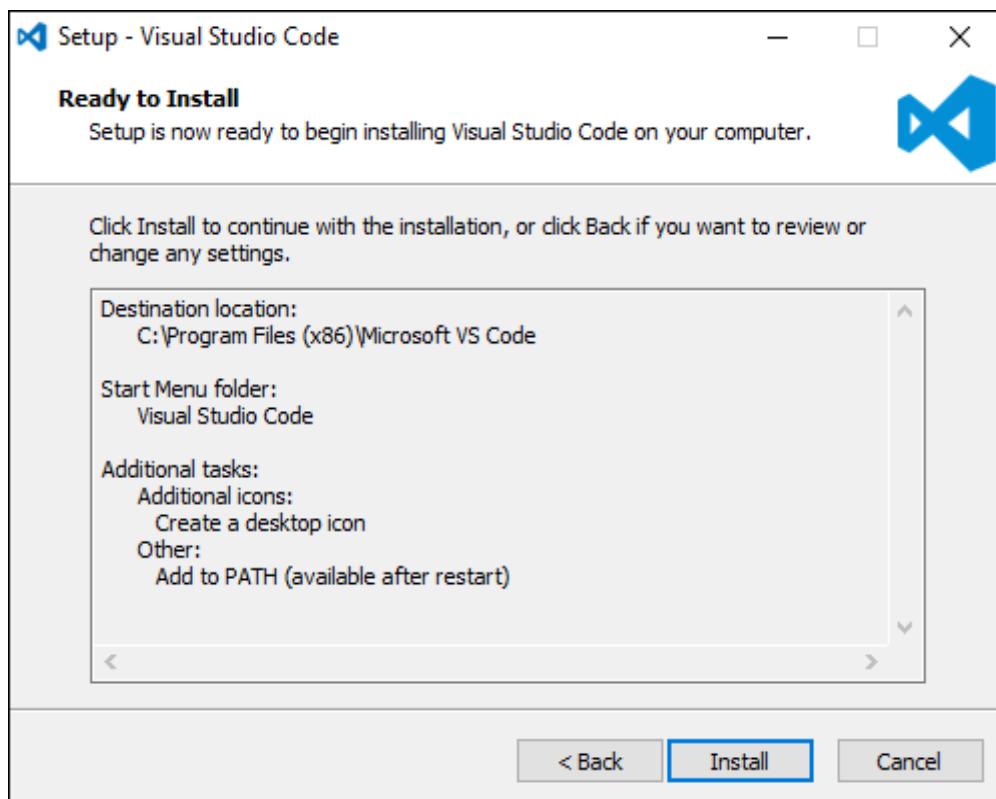
Step 4 – Choose the name of the program's shortcut and click the Next button.



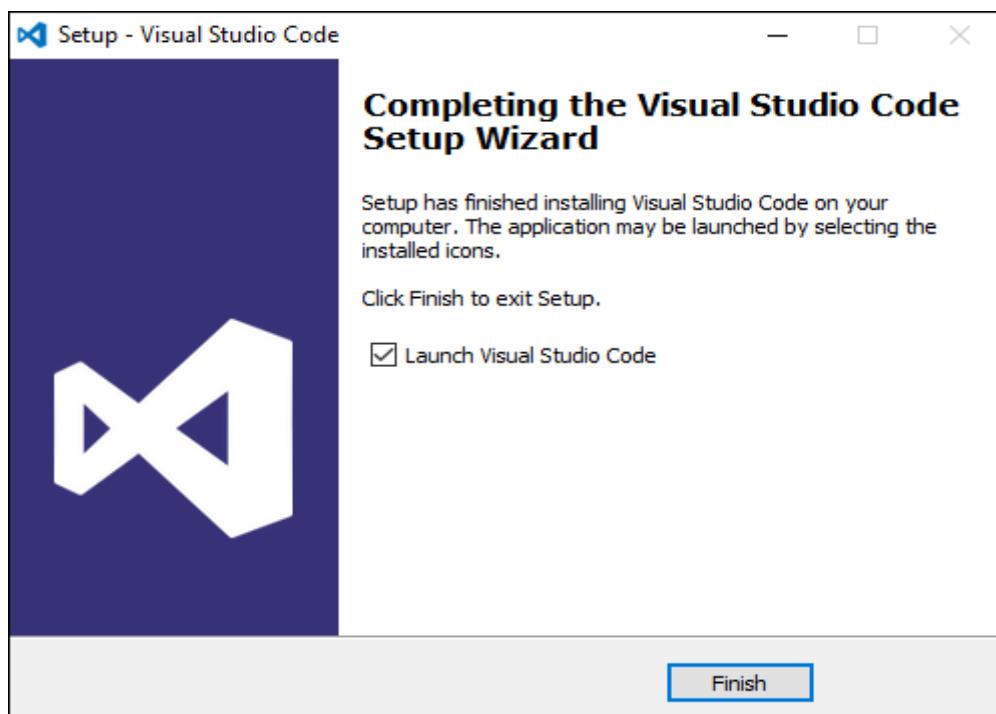
Step 5 – Accept the given default settings and click on the Next button.



Step 6 – Click on the Install button on the next screen.



Step 7 – and finally click on the Finish button to launch Visual Studio Code.



To verify the Visual Studio Code is installed, open the command prompt and type the following command:
> code –version or code -v

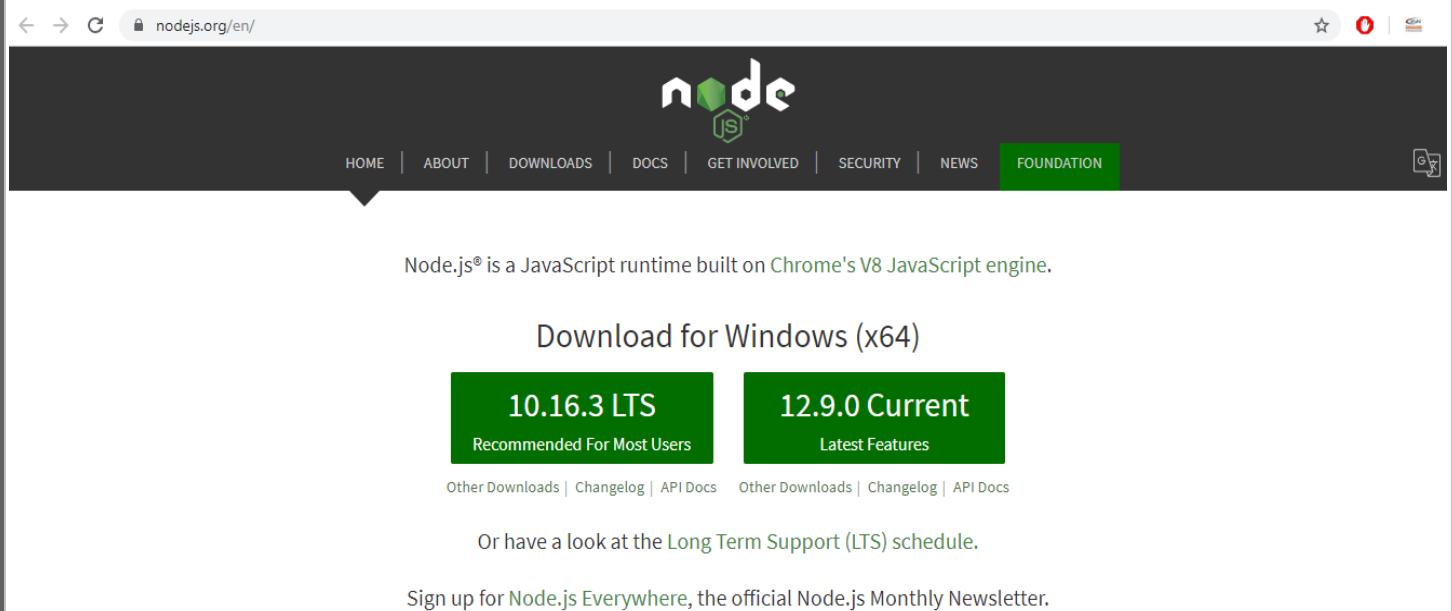
1.37.1

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Nodejs:

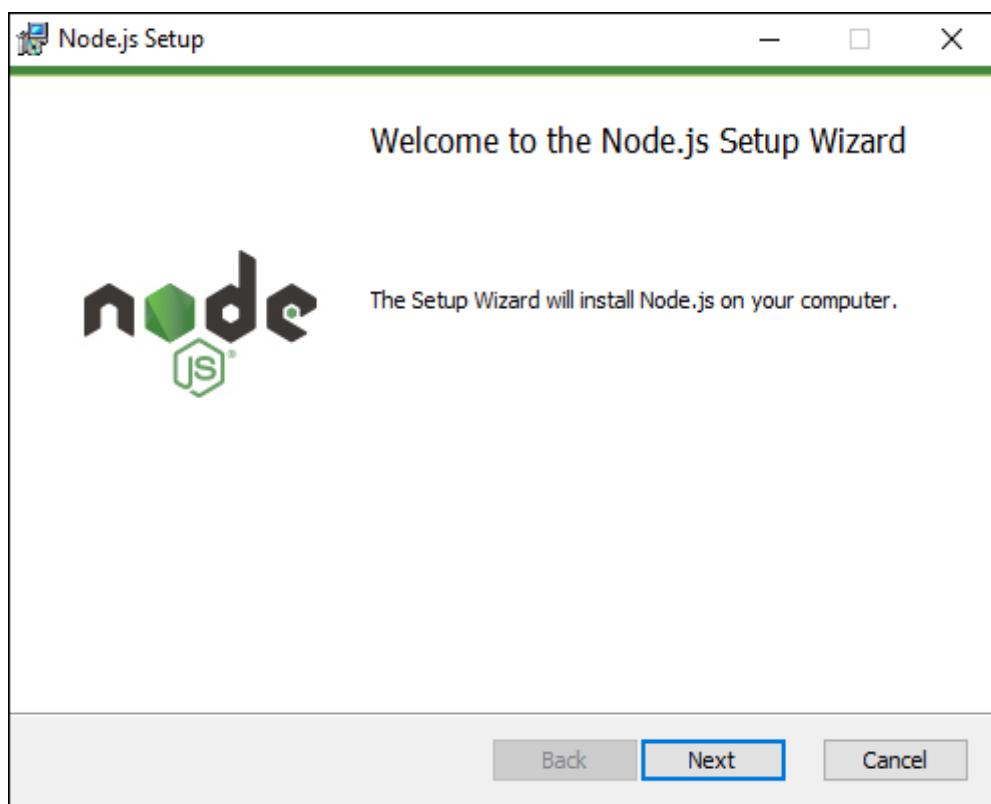
Angular 7/8 require Nodejs. You can download Nodejs from <https://nodejs.org/en/>, and click on the windows installer. You can see here both the **LTS** and the **current version** of the node from where you can download the recommended version or the current version.

Download and install latest version of Node.js. In our case, it is 12.9.0

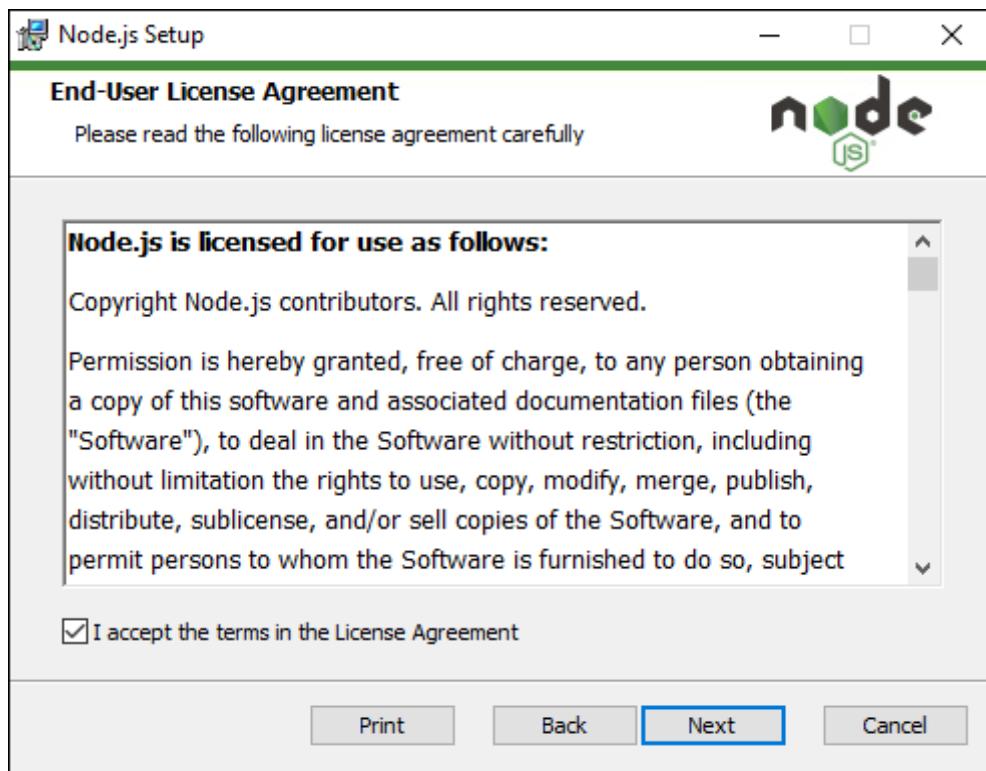


The screenshot shows the official Node.js website at nodejs.org/en/. The page features the Node.js logo at the top. Below it, a navigation bar includes links for HOME, ABOUT, DOWNLOADS, DOCS, GET INVOLVED, SECURITY, NEWS, and FOUNDATION. The FOUNDATION link is highlighted with a green background. A sub-header below the navigation states: "Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine." Two large green buttons are prominently displayed: "10.16.3 LTS" (labeled "Recommended For Most Users") and "12.9.0 Current" (labeled "Latest Features"). Below these buttons, smaller links for "Other Downloads", "Changelog", and "API Docs" are visible. A note below the buttons says, "Or have a look at the [Long Term Support \(LTS\) schedule](#)." Further down, there's a link to "Sign up for Node.js Everywhere, the official Node.js Monthly Newsletter."

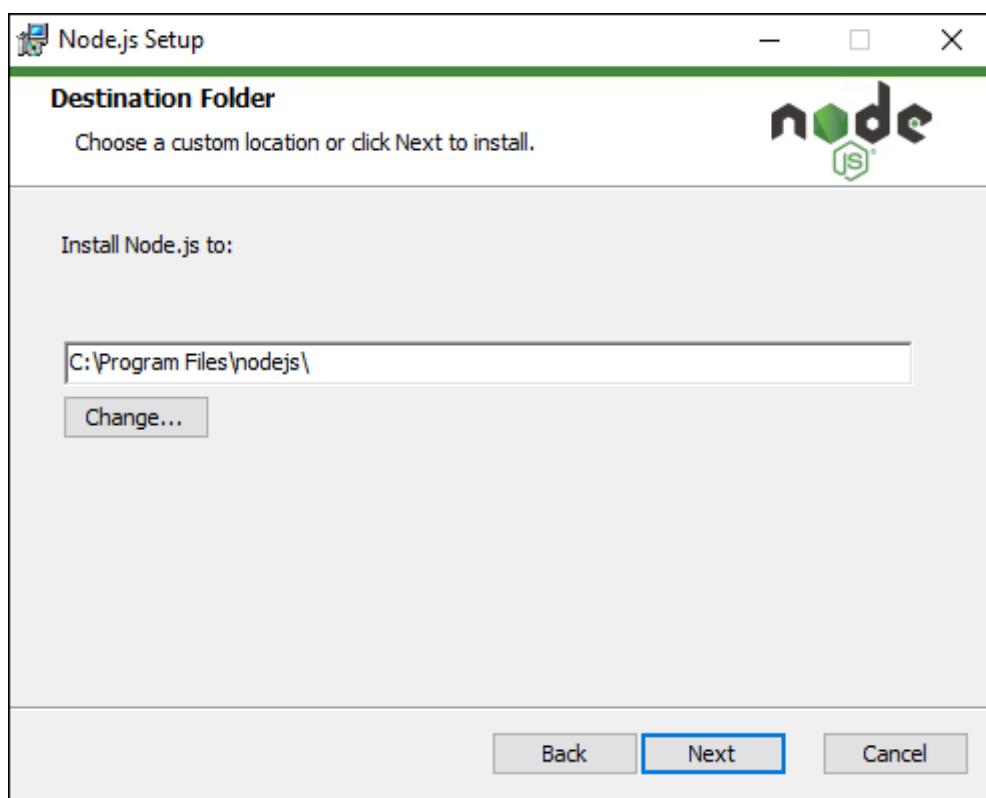
Step 1 – Launch the installer and in the initial screen, click the Next button.



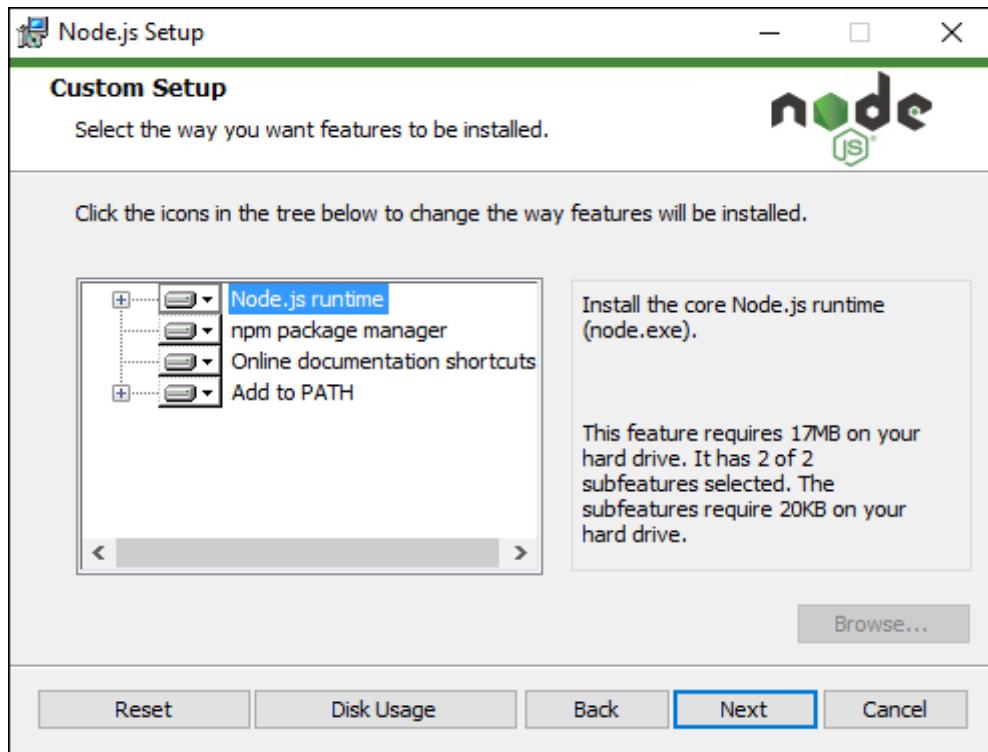
Step 2 – in the next screen, accept the license agreement and click on the next button.



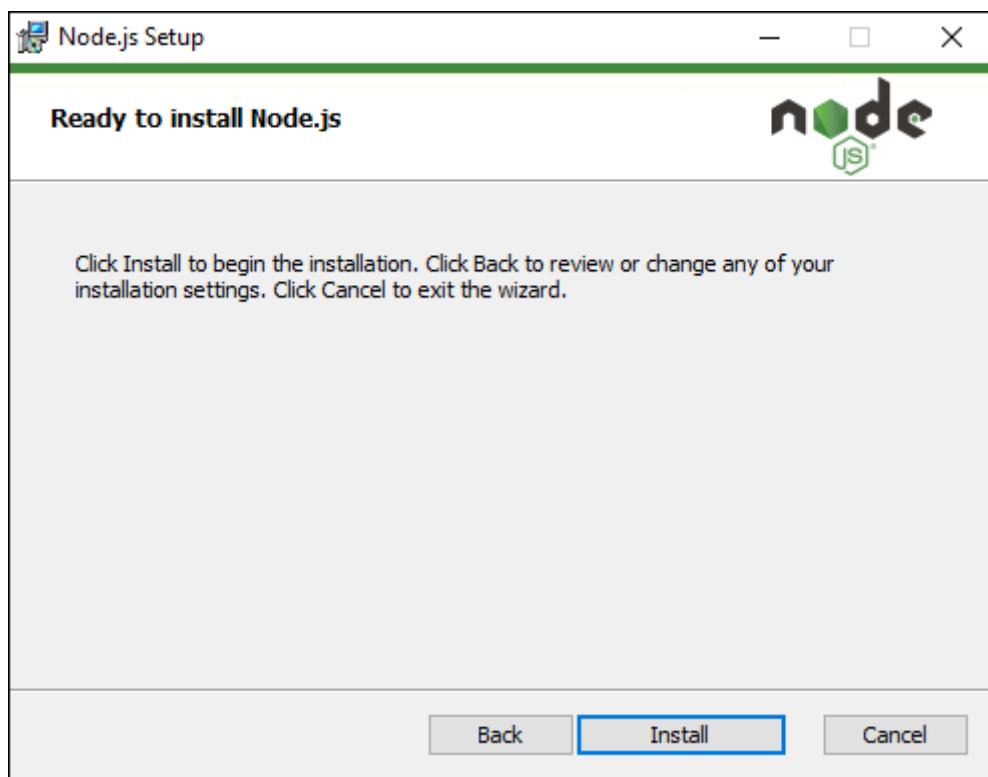
Step 3 – in the next screen, choose the destination folder for the installation and click on the Next button.



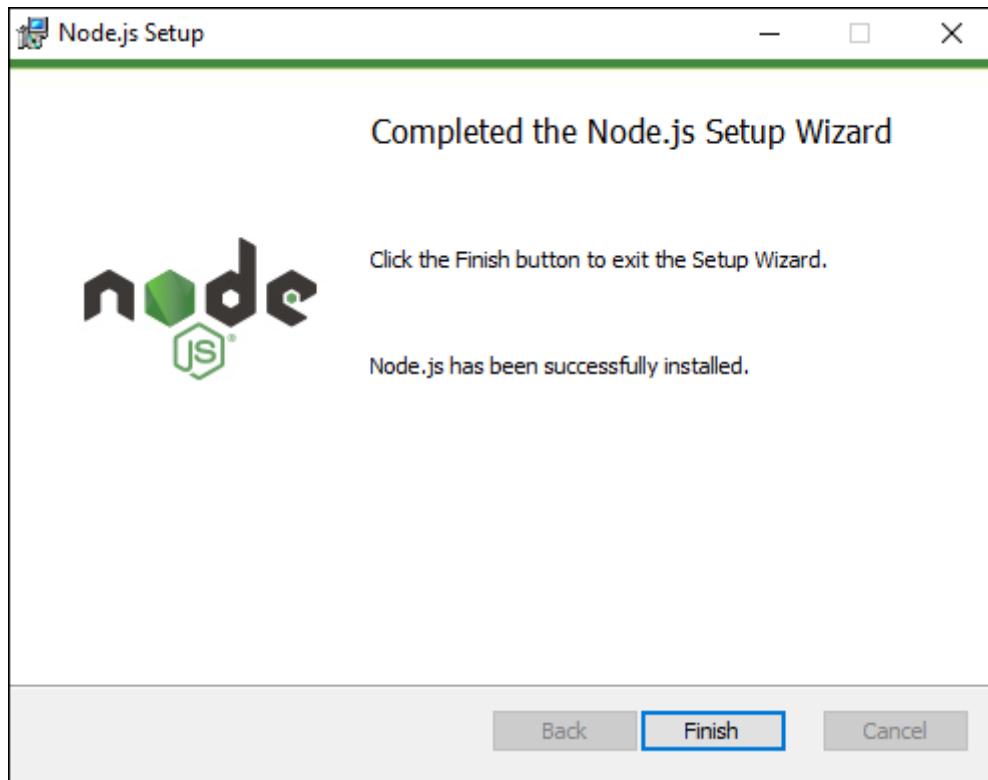
Step 4 – Choose the components in the next screen and click the Next button. For the default installation, you can accept all the components.



Step 5 – in the next screen, click on the Install button.



Step 6 – Once the installation is complete, click on the finish button.



After installing the node, open the command prompt window and type node -v to verify the installed version of the node. This will help you to see the version of nodejs currently installed on your system.

```
C:\Users\RakeshSoftNET>node -v
```

v12.9.0

Npm (Node Package Manager):

Npm (Node Package Manager) is a package manager for the JavaScript programming language. It is the default package manager for the JavaScript runtime environment Node.js. Depending on your operating system, install the required package. Once nodejs is installed, npm will also be installed, it is not necessary to install it separately. Type npm -v in the terminal to verify if npm is installed or not. It will display the version of the npm.

```
C:\Users\RakeshSoftNET>npm -v
```

6.10.2

Angular CLI (Command Line Interface):

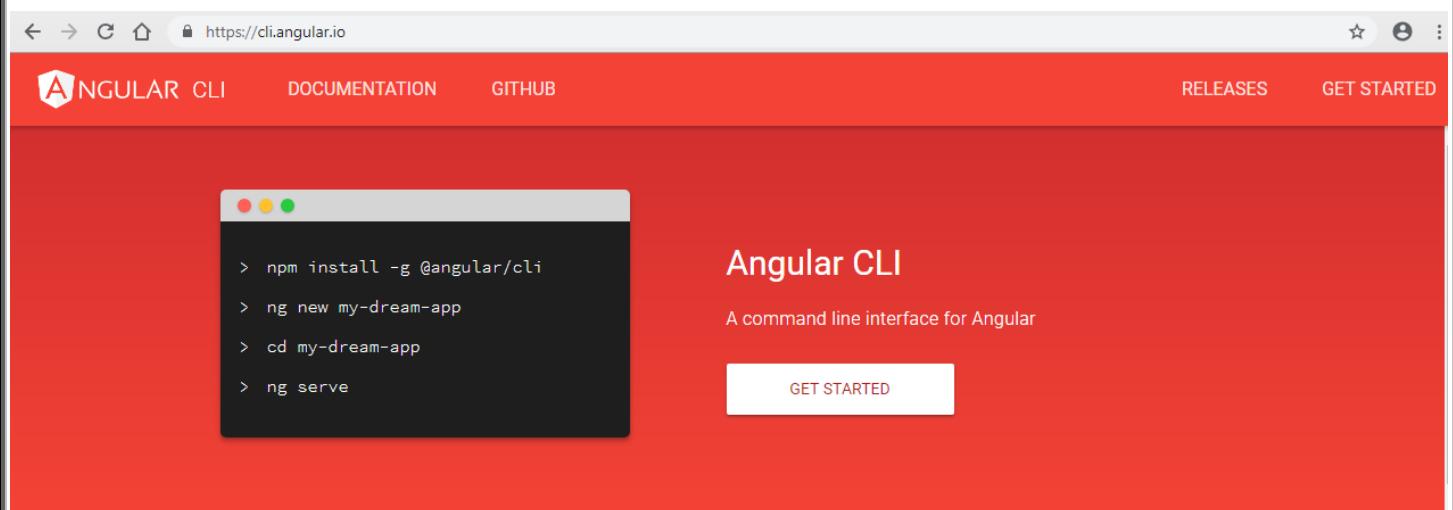
The Angular CLI is a command line interface tool that can create a project, add files, and perform a variety of ongoing development tasks such as testing, bundling, and deployment.

The Angular CLI is a tool to initialize, develop, scaffold and maintain Angular applications.

An Angular CLI project is the foundation for both quick experiments and enterprise solutions.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Angular CLI is very important in the setting of Angular, visit the homepage <https://cli.angular.io/> of angular to get the reference of the command.



The screenshot shows the Angular CLI homepage with a red header. The header includes links for ANGULAR CLI, DOCUMENTATION, GITHUB, RELEASES, and GET STARTED. Below the header is a large orange section featuring a terminal window with the following commands:

```
> npm install -g @angular/cli
> ng new my-dream-app
> cd my-dream-app
> ng serve
```

To the right of the terminal window, the text "Angular CLI" is displayed in bold, followed by "A command line interface for Angular". A "GET STARTED" button is located below this text.

To install angular cli globally on your system type **npm install -g @angular/cli**. It installs Angular CLI globally where g is referred to globally.

Install Globally:

C:\Users\RakeshSoftNET>npm install -g @angular/cli

OR

C:\Users\RakeshSoftNET>npm install -g @angular/cli@latest

Install Locally

C:\Users\RakeshSoftNET>npm install @angular/cli

OR

C:\Users\RakeshSoftNET>npm install @angular/cli@latest

If you want to make sure you have correctly installed the angular CLI, open the command prompt window and type **ng --version**. If you can see the cli version as shown below, then installation is complete.



A screenshot of a Windows Command Prompt window titled "Command Prompt". The window shows the following output:

```
C:\Users\RakeshSoftNET>ng --version
Angular CLI: 8.2.2
Node: 12.8.1
OS: win32 x64
Angular:
...
Package          Version
@angular-devkit/architect    0.802.2
@angular-devkit/core         8.2.2
@angular-devkit/schematics   8.2.2
@schematics/angular          8.2.2
@schematics/update           0.802.2
rxjs                      6.4.0

C:\Users\RakeshSoftNET>
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

To install specific Version (Example: 6.1.1)

```
C:\Users\RakeshSoftNET>npm install -g @angular/cli@6.1.1
```

If you want to uninstall angular-cli which is already installed, run the following command:

```
C:\Users\RakeshSoftNET>npm uninstall -g @angular/cli
```

Then run npm cache clean - it will clean your npm cache from app data folder under your username.

```
C:\Users\RakeshSoftNET>npm cache clean
```

Then use npm cache verify - It will verify your cache whether it is corrupted or not.

```
C:\Users\RakeshSoftNET>npm cache verify
```

Use npm cache verify --force -in order to clean your entire cache from your system.

```
C:\Users\RakeshSoftNET>npm cache verify --force
```

Create Project: Creating a First Angular 7/8 Application

In the earlier, we learned how to set up the Angular 7/8 environment setup. In this, we will learn how to create the project in Angular 7/8 using Angular CLI.

Generating and serving an Angular project via a development server.

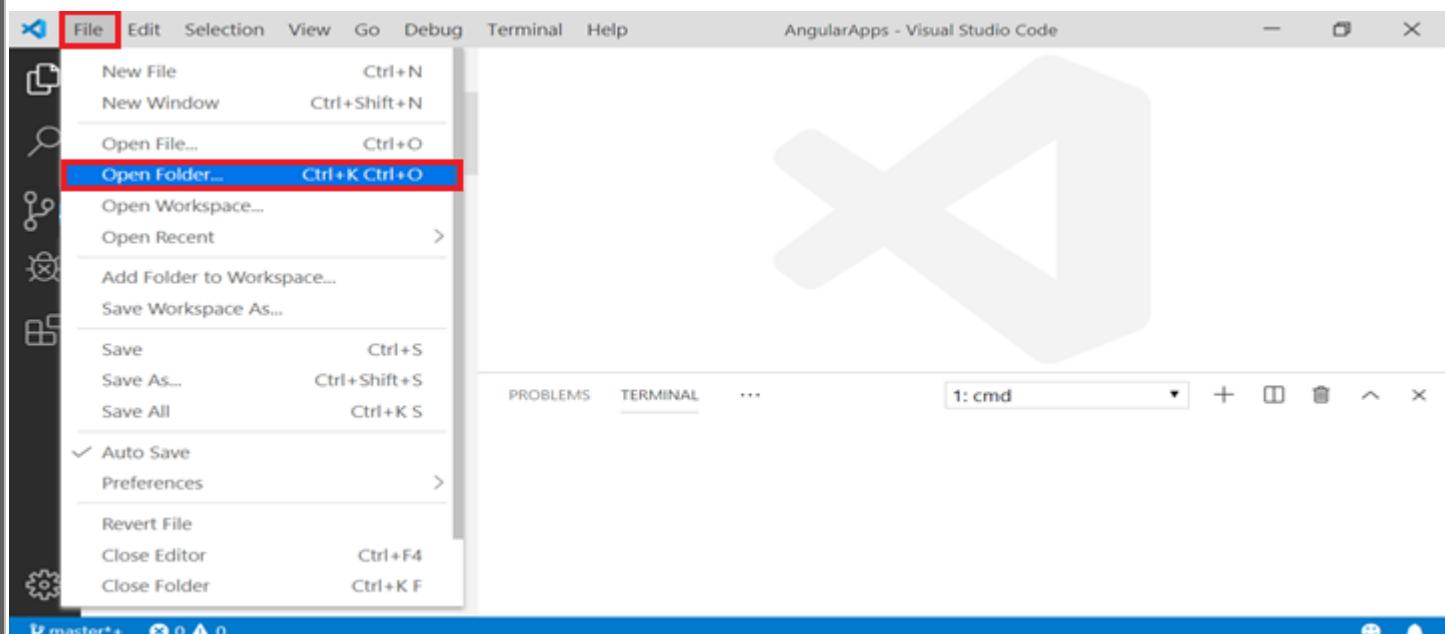
Now let's create our first project in Angular 7/8. To create a project in Angular 7/8, we will use the following command –

```
> ng new project-name
```

We will name the project as ng new DemoAngularApp1.

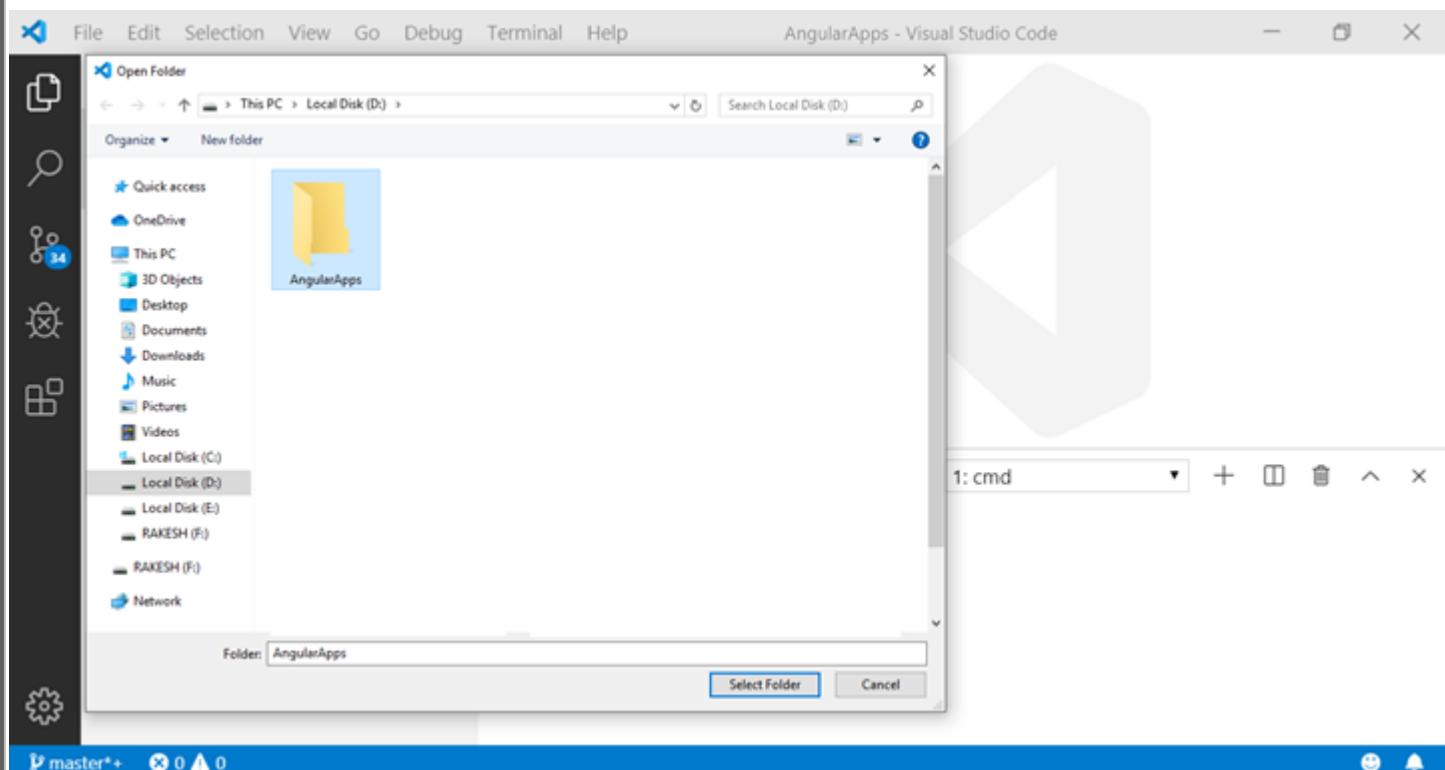
Step1:

First, create a folder name as **AngularApps**, on a desktop or wherever you want. Open Visual Studio Code, click on File, go to "Open Folder (Ctrl + O)" option and then click on it.



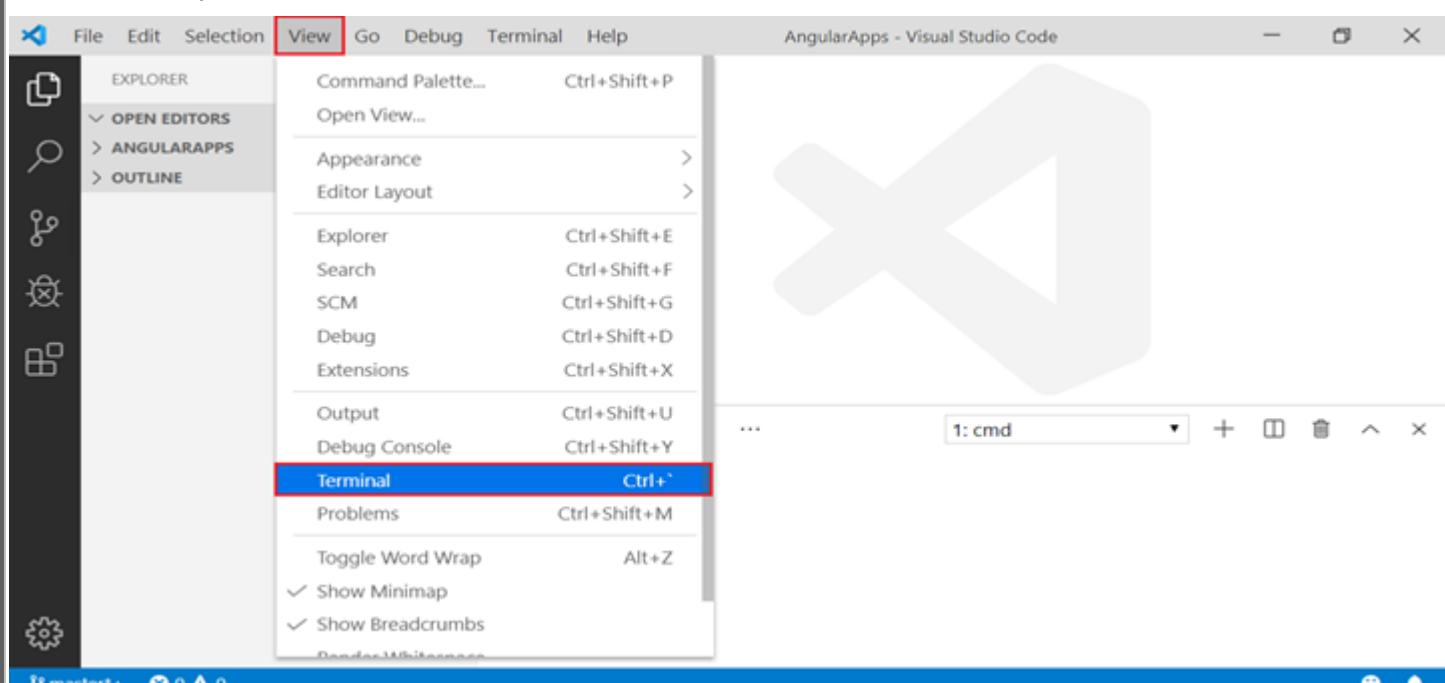
Step2:

After that, a window will appear. Select the created folder and click on Select Folder box as shown in the below image.

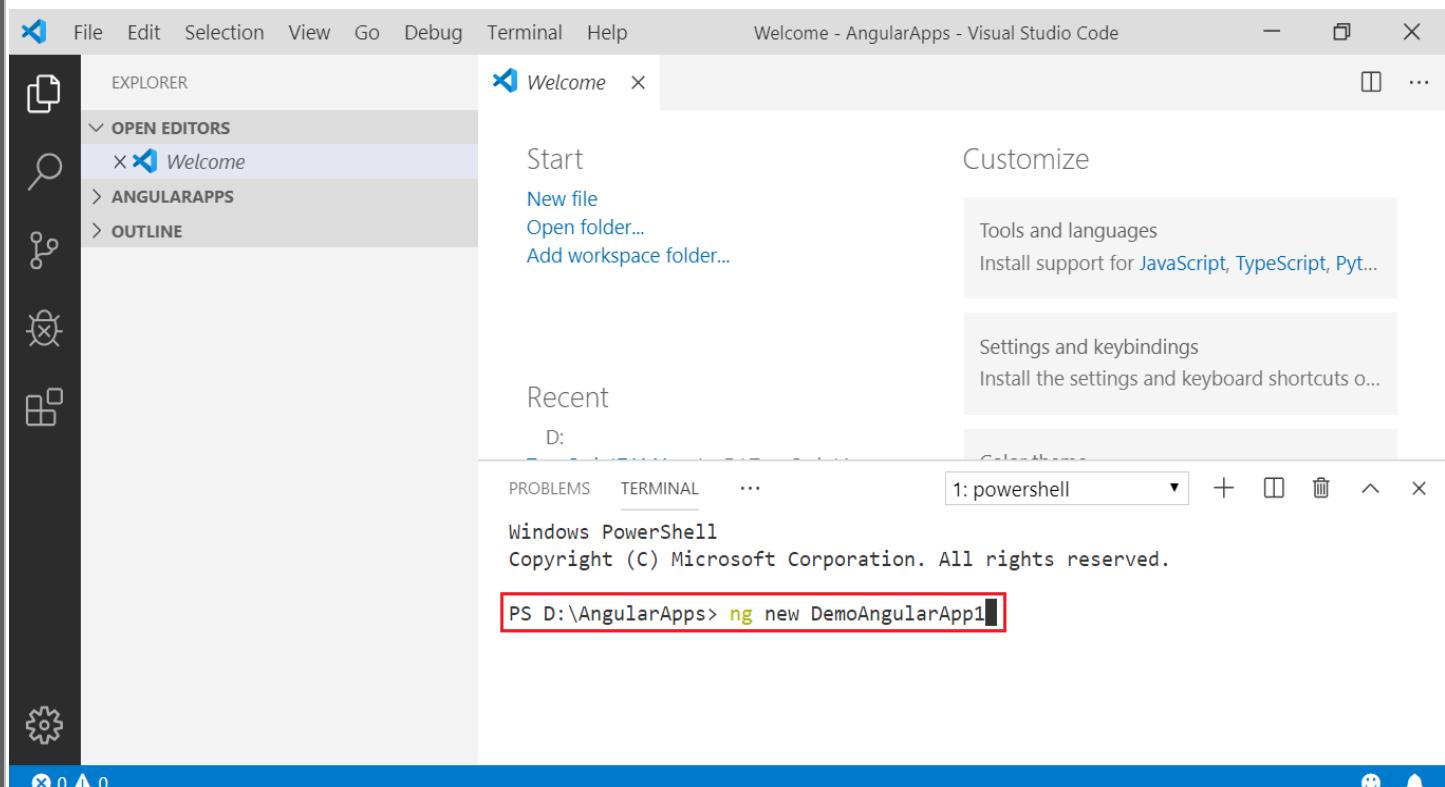


Step3:

To create the application, click on the view, in the list select “Terminal” and click on it. Visual Studio Code Console will open.



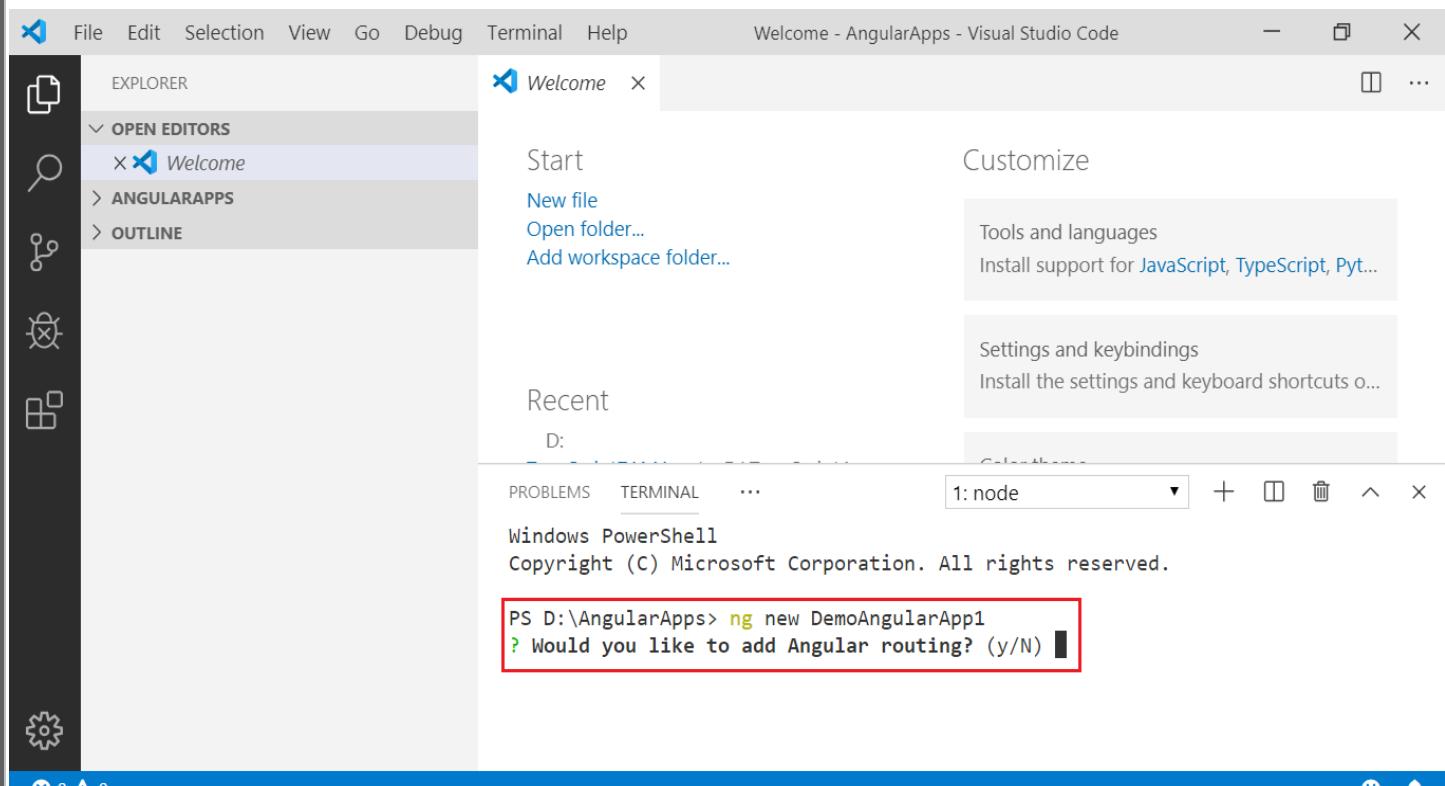
Now run the `ng new DemoAngularApp1` command in the command line.



The screenshot shows the Visual Studio Code interface. The terminal window at the bottom has the following content:

```
PS D:\AngularApps> ng new DemoAngularApp1
```

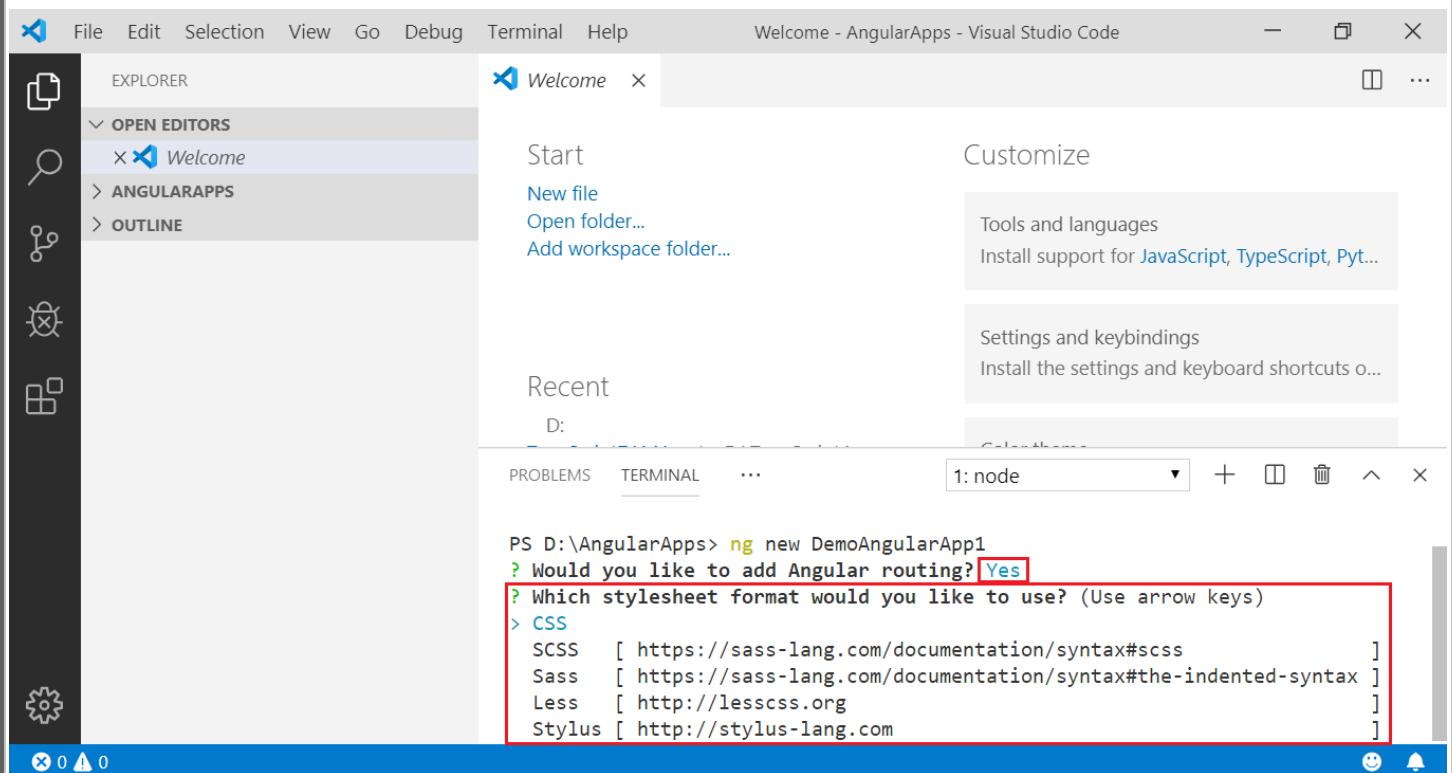
It's going to present you with a couple of questions before beginning: the Angular CLI will prompt you with two options. The first option will ask you whether you want to add routing support or not.



The screenshot shows the Visual Studio Code interface. The terminal window at the bottom has the following content:

```
PS D:\AngularApps> ng new DemoAngularApp1
? Would you like to add Angular routing? (y/N)
```

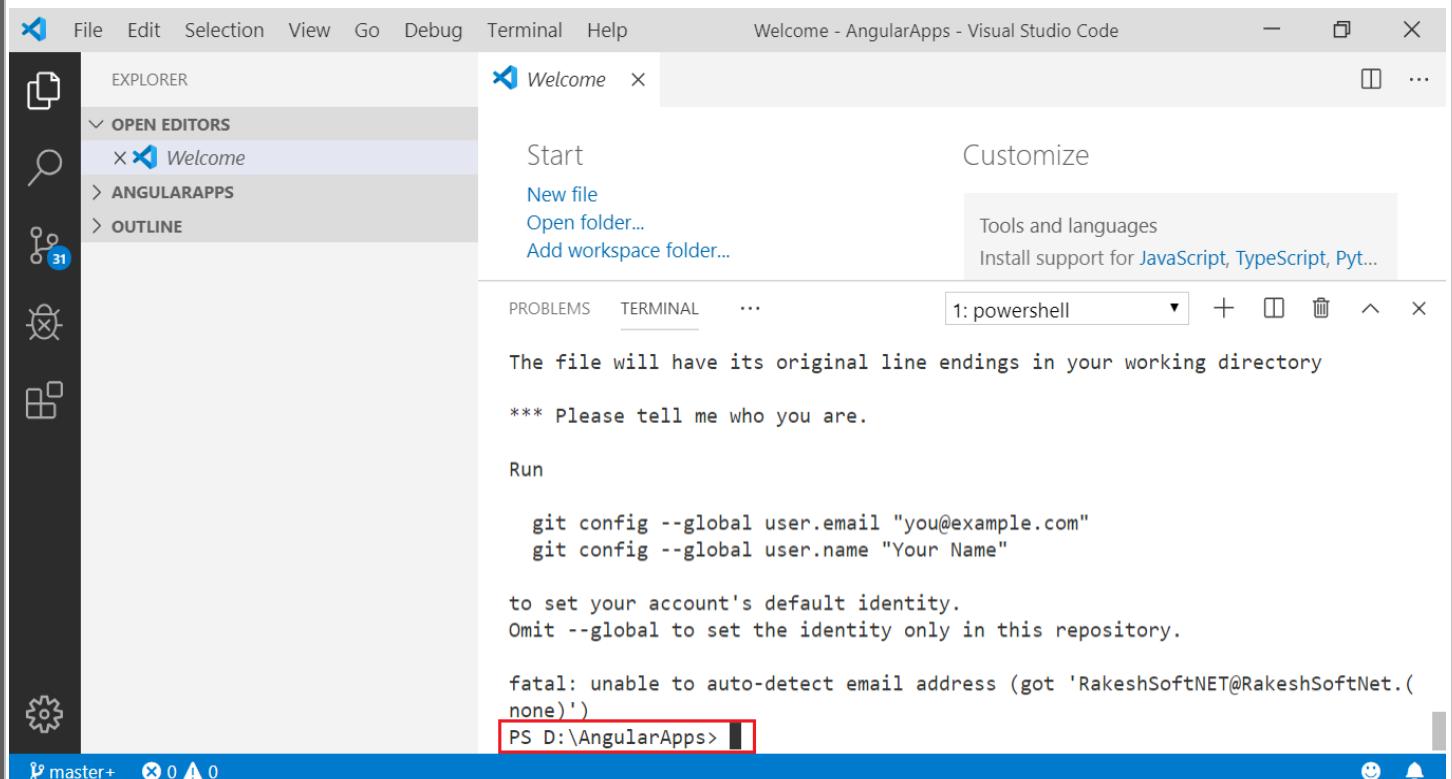
While the second option will ask you which stylesheet format (CSS, SCSS, SASS, Less, Stylus etc.) you would like to use.



The screenshot shows the Visual Studio Code interface. In the terminal, the command `ng new DemoAngularApp1` is run, followed by a question: "Would you like to add Angular routing?". The user types "Yes" and presses Enter. This triggers a series of questions about the stylesheet format. The options listed are CSS, SCSS, Sass, Less, and Stylus. The SCSS option is highlighted with a red box. The terminal output is as follows:

```
PS D:\AngularApps> ng new DemoAngularApp1
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? (Use arrow keys)
> CSS
SCSS [ https://sass-lang.com/documentation/syntax#scss ]
Sass [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
Less [ http://lesscss.org ]
Stylus [ http://stylus-lang.com ]
```

The project (DemoAngularApp1) is created successfully. It installs all the required packages necessary for our project to run in Angular 7/8.



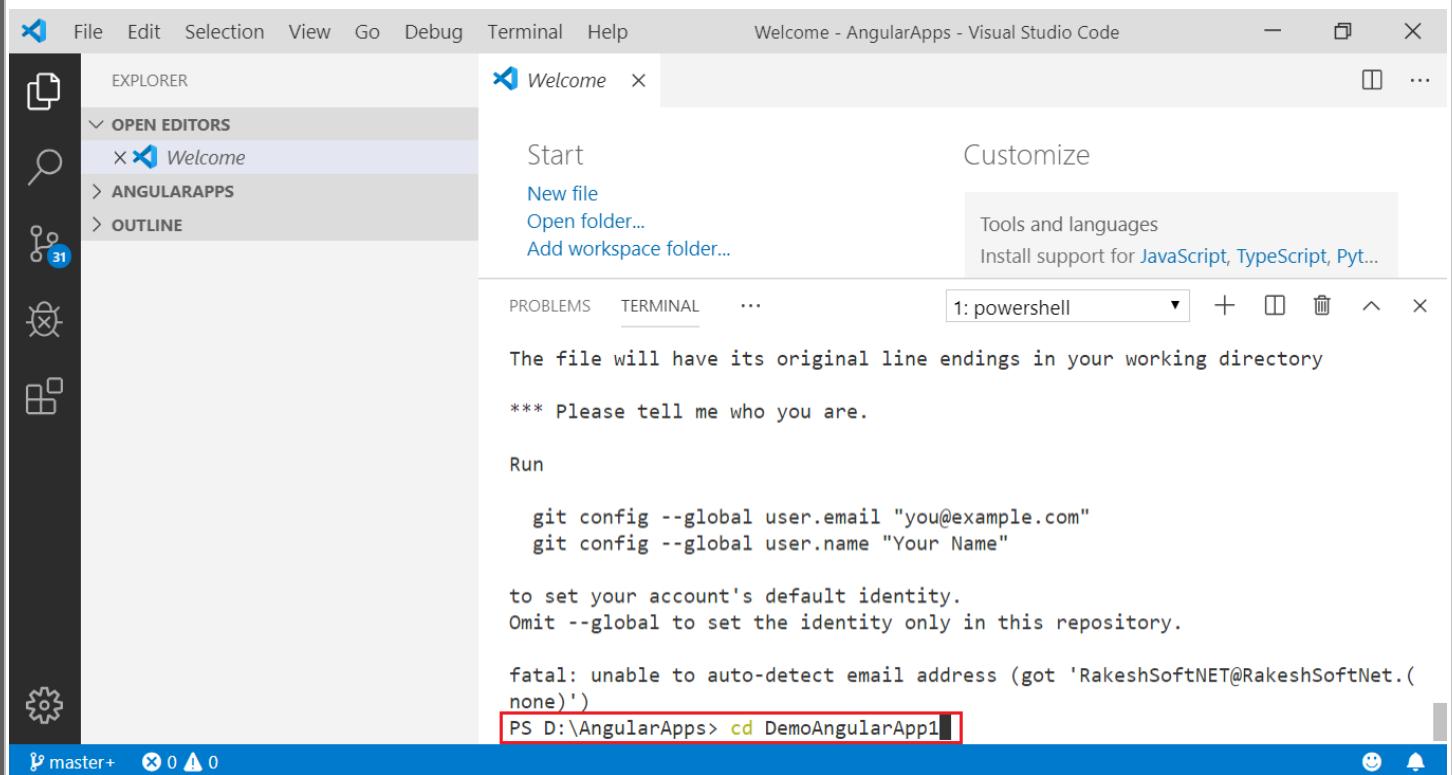
The screenshot shows the Visual Studio Code interface. In the terminal, the command `git config --global user.email "you@example.com"` and `git config --global user.name "Your Name"` are run. The terminal then asks the user to set their account's default identity. The user types "PS D:\AngularApps>" and presses Enter. The terminal output is as follows:

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'RakeshSoftNET@RakeshSoftNet.(none)')
PS D:\AngularApps>
```

Let us now switch to the project created, which is in the directory DemoAngularApp1. Change the directory in the command line - cd DemoAngularApp1.



```

File Edit Selection View Go Debug Terminal Help Welcome - AngularApps - Visual Studio Code
EXPLORER OPEN EDITORS Welcome Start Customize
  X Welcome
  > ANGULARAPPS
  > OUTLINE
New file Open folder... Add workspace folder...
PROBLEMS TERMINAL ...
1: powershell + - ×
The file will have its original line endings in your working directory
*** Please tell me who you are.

Run

git config --global user.email "you@example.com"
git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'RakeshSoftNET@RakeshSoftNet.(none)')
PS D:\AngularApps> cd DemoAngularApp1

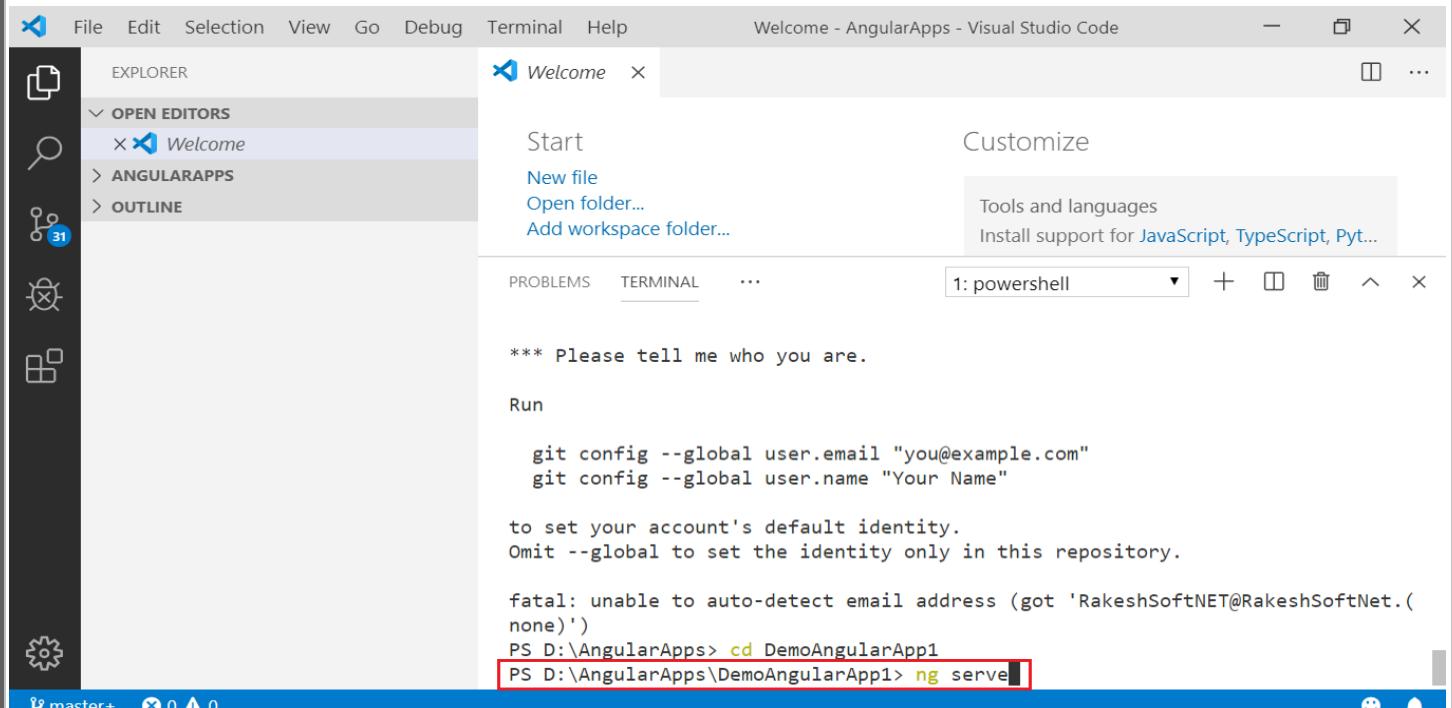
```

master+ 0 ▲ 0 1: powershell

Let's compile & run our project with the following command –

> ng serve

The ng serve command will compiles and builds the application, and starts the local web server.



```

File Edit Selection View Go Debug Terminal Help Welcome - AngularApps - Visual Studio Code
EXPLORER OPEN EDITORS Welcome Start Customize
  X Welcome
  > ANGULARAPPS
  > OUTLINE
New file Open folder... Add workspace folder...
PROBLEMS TERMINAL ...
1: powershell + - ×
*** Please tell me who you are.

Run

git config --global user.email "you@example.com"
git config --global user.name "Your Name"

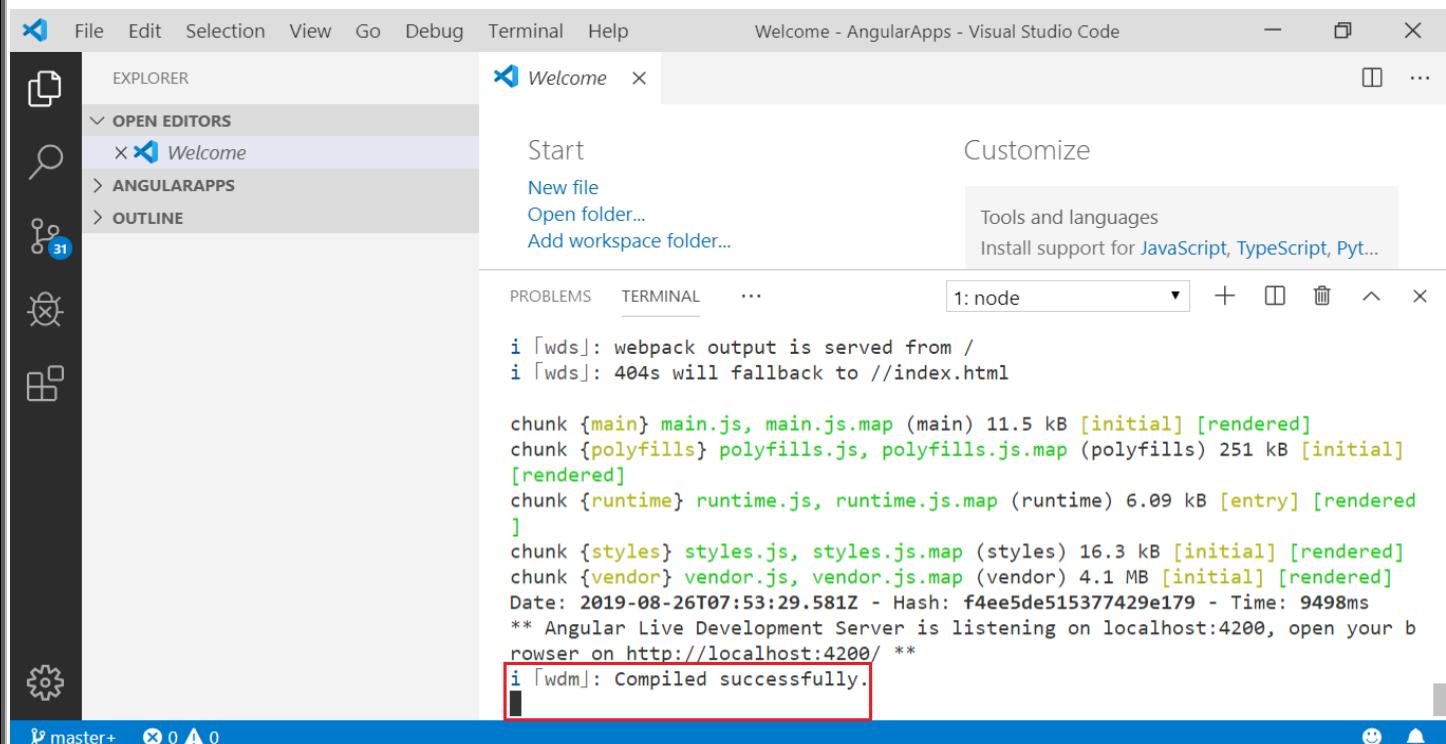
to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'RakeshSoftNET@RakeshSoftNet.(none)')
PS D:\AngularApps> cd DemoAngularApp1
PS D:\AngularApps\DemoAngularApp1> ng serve

```

master+ 0 ▲ 0 1: powershell

Once the project has been compiled, you will get the output that was Compiled Successfully, as shown in the image below:



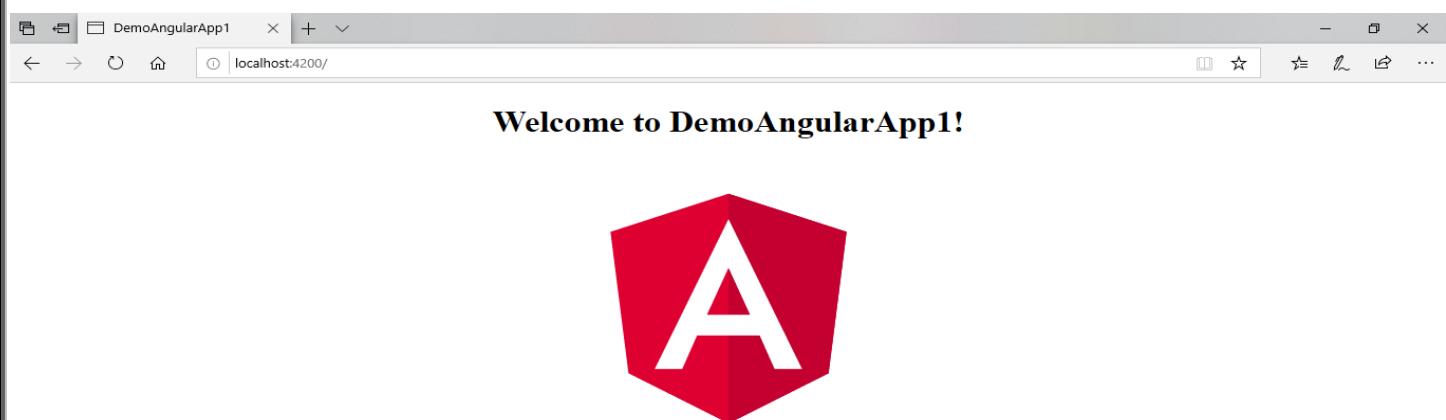
The screenshot shows the Visual Studio Code interface. The terminal window displays the following output:

```
i 「wds」: webpack output is served from /
i 「wds」: 404s will fallback to //index.html

chunk {main} main.js, main.js.map (main) 11.5 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 251 kB [initial]
[rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.09 kB [entry] [rendered]
]
chunk {styles} styles.js, styles.js.map (styles) 16.3 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 4.1 MB [initial] [rendered]
Date: 2019-08-26T07:53:29.581Z - Hash: f4ee5de515377429e179 - Time: 9498ms
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
i 「wdm」: Compiled successfully.
```

A red box highlights the final line: "i 「wdm」: Compiled successfully."

The local web server starts at localhost:4200. Enter the URL <http://localhost:4200/> in the browser and you will be redirected to the following screen (default page):



Here are some links to help you start:

- [Tour of Heroes](#)
- [CLI Documentation](#)
- [Angular blog](#)

Or directly compiles and serve in the browser automatically using following command:

> ng serve --open

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Project Structure:

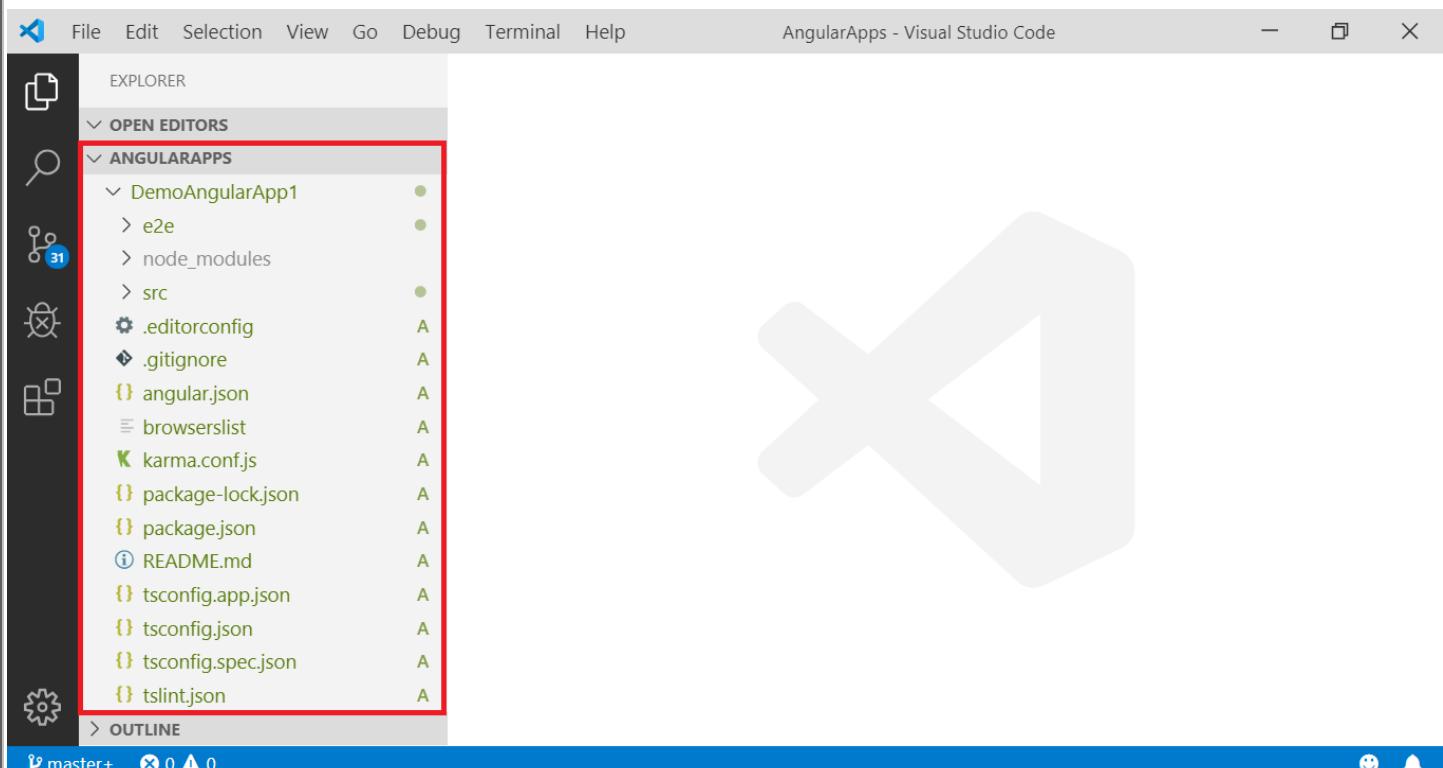
See the structure of the Angular 7/8 application on Visual Studio Code IDE. For Angular 7/8 development, you can use either Visual Studio Code IDE or WebStorm IDE. Both are good. Here, we are using Visual Studio Code IDE.

In the earlier, we learned how to create a project in Angular 7/8. Here describes the project structure of Angular 7/8 application which Angular CLI has created for us.

We know that lots of files and folders are generated whenever we create a new project in Angular. So, in this, you will learn the folder structure of Angular 7/8.

We have already created **DemoAngularApp1** project in the previous section, and now we will use that project to get an idea about project structure.

When we open the Angular 7/8 project in the editor, we find three main folders **e2e**, **node_modules**, **src**, and different **configuration files**.



1. /e2e/ directory:

e2e stands for end-to-end and this is the place where we can write the end to end test cases. This folder contains test cases for testing the complete application along with its test-specific configuration files.

e2e/

src/ ([end-to-end tests for project](#))

app.e2e-spec.ts

app.po.ts

protractor.conf.js ([test-tool config](#))

tsconfig.json



2. /node_modules/:

All the libraries present in node_module folder. These libraries helps to develop and execute the angular applications. The npm packages installed in node_modules. You can open the folder and see the packages available. Packages specified in package.json file are installed into this node_modules folder.

3. /src/:

This folder contains the complete source code of the application for developer. This folder is where we will work on the project using Angular in development. The **src/** folder is the main folder, which internally has a different folders & files structure as following.

src	
> app	A
> assets	A
> environments	A
★ favicon.ico	A
↳ index.html	A
TS main.ts	A
TS polyfills.ts	A
# styles.css	A
TS test.ts	A

/app/: This is the place where we will keep our application source code. It contains the files described below. These files are installed by angular-cli by default. It contains all the modules and components of your application where every application has at least one module and one component.

app	
TS app-routing.module.ts	A
# app.component.css	A
↳ app.component.html	A
TS app.component.spec.ts	A
TS app.component.ts	A
TS app.module.ts	A

app.module.ts:

This is a typescript file which includes all the dependencies for the website. This file is used to define the needed modules to be imported, the components to be declared and the main component to be bootstrapped.

Defines the root module, named **AppModule** that tells Angular how to assemble the application. Initially declares only the **AppComponent**. As you add more components to the app, they must be declared here.

If you open the file, you will see that the code has reference to different libraries, which are imported. Angular-cli has used these default libraries for the import: angular/core, platform-browser.

The names itself explain the usage of the libraries. They are imported and saved into variables such as **declarations**, **imports**, **providers**, and **bootstrap**.

We can see **app-routing.module** is also added. This is because we had selected routing at the start of the installation. The module is added by @angular/cli.

Following is the structure of the file:

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

@NgModule is imported from @angular/core and it has object with following properties:

Declarations: In declarations, the reference to the components is stored. The **AppComponent** is the default component that is created whenever a new project is initiated.

Imports: This will have the modules imported as shown above. At present, **BrowserModule** is part of the imports which is imported from @angular/platform-browser. There is also routing module added **AppRoutingModule**.

Providers: This will have reference to the services created.

Bootstrap: This has reference to the default component created, i.e., **AppComponent**.

app.component.css: This file contains the cascading style sheets (css) code for your app component. You can write your css over here. Right now, we have added the one css class with background color property to apply an html element as shown below.

The structure of the file is as follows:

```
.divStyle{
  background-color: yellow;
}
```

app.component.html: This file contains the html file related to app component. This is the template file which is used by angular to do the data binding. The html code will be available in this file.

The structure of the file is as follows:

```
<!--The content below is only a placeholder and can be replaced.-->
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  
</div>
<h2>Here are some links to help you start: </h2>
<ul>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/tutorial">Tour of Heroes</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/cli">CLI Documentation</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h2>
  </li>
</ul>

<router-outlet></router-outlet>
```

This is the default html code currently available with the project creation.

app.component.spec.ts: This file is a unit testing file related to app component. This file is used along with other unit tests. It is run from Angular CLI by the command ng test.

app.component.ts: This is the most important typescript file which includes the logic for the app's root component, named AppComponent. The view associated with this root component becomes the root of the view hierarchy as you add components and services to your application.

The class for the component is defined over here. You can do the processing of the html structure in the .ts file. The processing will include activities such as connecting to the database, interacting with other components, routing, services, etc.

Technically, an Angular component is a **TypeScript class** decorated with the **@Component** decorator which is part of the Angular core.

A component has an associated view which is simply an HTML file (but can also contain some special Angular template syntax which helps display data and bind events from the controller component)

A component has also one or more associated stylesheet files for adding styles to the component view. These files can be in many formats like CSS, Stylus, Sass or Less.

The structure of the file is as follows:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'DemoAngularApp1';
}
```

In this file, we export the **AppComponent** class, and we decorate it with the **@Component** decorator, imported from the **@angular/core** package, which takes a few metadata, such as:

- **selector**: this allows you to invoke the component from an HTML template or file just like standard HTML tags i.e.: `<my-app></my-app>`,
- **templateUrl**: This is used to tell the component where to find the HTML view,
- **styleUrls**: This is an array of relative paths to where the component can find the styles used to style the HTML view.

AppComponent is the **root** component of our application. It's the base of the tree of components of our application and it's the first component that gets inserted in the browser DOM.

An Angular application is composed of a tree of components, in which each Angular component has a specific purpose and responsibility.

A component is comprised of three things:

- **A component class**, which handles data and functionality.
- **An HTML template**, which determines what is presented to the user.
- **Component-specific styles** that define the look and feel.

app.routing.module.ts: This file deals with the routing required for your project. It is connected with the main module i.e. **app.module.ts**

The structure of the file is as follows:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
const routes: Routes = [];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

/assets/: This directory is used to contain the static resources. In other words, this is the place where you can store static assets of your application for example images, icons, xml files, json files, css files etc...

/environments/: It contains build configuration environments of the application. This folder has details for the production and the development environment. The folder contains two files:

environment.prod.ts (Production Environment)

```
export const environment = {
  production: true
};
```

environment.ts (Development Environment)

```
export const environment = {
  production: false
};
```

Both the files have details of whether the final file should be compiled in the production environment or development environment.

Noted: The angular build system defaults to the dev environment so it uses environment.ts file. If we are doing a production build, it uses environment.prod.ts file. Environment and file mappings are specified in the angular cli configuration file i.e. angular.json

favicon.ico: This file specifies a small icon that appears next to the browser tab of a website.

index.html: This is the file which is displayed in the browser. This is entry file which holds the high level container for angular application. This is the main html page which is rendered when someone opens your website or application.

The structure of the file is as follows:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>DemoAngularApp1</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

The body has **<app-root></app-root>**. This is the selector which is used in **app.component.ts** file and will display the details from **app.component.html** file.

main.ts: main.ts is the file from where we start our project development. It starts with importing the basic module which we need. Right now if you see angular/core, angular/platform-browser-dynamic, app.module and environment is imported by default during angular-cli installation and project setup.

The platformBrowserDynamic.bootstrapModule(AppModule) has the parent module reference AppModule. Hence, when it executes in the browser, the file is called index.html. Index.html internally refers to main.ts which calls the parent module i.e. AppModule when the following code executes -

```
platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

When AppModule is called, it calls app.module.ts which further calls the AppComponent based on the bootstrap as follows:

bootstrap: [AppComponent]

In app.component.ts, there is a selector: app-root which is used in the index.html file. This will display the contents present in app.component.html.

In other words we can say this is the main entry point of the angular application which compiles the AppModule and renders the specified bootstrap component in the browser.

It is the starting point of your application or you can say that this is where our angular app bootstraps. If you open it you will find that there are no references to any stylesheet (CSS) nor JS files this is because all dependencies are injected during the build process.

polyfills.ts: This file is a set of code that can be used to provide compatibility support for older browsers. Angular 7/8 code is written mainly in ES6+ language specifications which is getting more adopted in front-end development, so since not all browsers support the full ES6+ specifications, polyfills can be used to cover whatever feature missing from a given browser.

styles.css: This is the style file required for the project. This is a global css file which is used by the angular application. Global stylesheet file means it is where we can add global styles for our applications. Note that each component has its own style component which applies styles only within the scope of the given component.

test.ts: This is the main test file that the Angular CLI command ng test will use to traverse all the unit tests within the application and run them.

tsconfig.json: This is a typescript compiler configuration file, ts stands for typescript. Since Angular 2 came out, typescripts are used for developing angular applications. This file contains the configurations for typescript. If there is a tsconfig file in a directory, that means that directory is a root directory of a typescript project, moreover, it is also used to define different typescript compilation related options.

tsconfig.app.json: This is used during compilation, it has the configuration details about how your application should compile.

tsconfig.spec.json: It is used for testing. This helps in maintaining the details for testing.

tslint.json: tslint is a tool useful for static analysis that checks our TypeScript code for readability, maintainability, and functionality errors.

browserslist: This file is used by the build system to adjust CSS and JS output to support the specified browsers. For additional information regarding the format and rule options, please see:

<https://github.com/browserslist/browserslist#queries>

karma.conf.js: It is used to store the setting of Karma i.e. test cases. It has a configuration for writing unit tests. Karma is the test runner and it uses jasmine as a framework. Both tester and framework are developed by the angular team for writing unit tests.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

.editorconfig: It is the config file for the editor which contains the setting of your editor. It has a parameter like style, size of the character, line length.

.gitignore: This file is related to the source control git. This file instructs git which files should be ignored when working with a git repository in order to share the ignore rules with any other users that clone the repository.

angular.json: Since Angular CLI v6-RC2, the angular-cli.json file has been replaced by angular.json. It is very important configuration file related to your angular application. It contains all the configuration of Angular Project. This file is mainly used for specifying configuration of CLI. It includes configuration of build, serve, test, lint, e2e commands which are used by @angular-cli.

package.json: This is npm configuration file. It includes details about your website's package dependencies. This file is mandatory for every npm project. It contains basic information regarding the **project, commands** which can be used, **dependencies** - these are packages required by the application to work correctly, and **devDependencies** - again the list of packages which are required for application however only during the development phase i.e. we need Angular CLI only during development to build a final package however for production we don't use it anymore.

The package.json is organized into two groups of packages:

- **Dependencies** are essential to *running* applications.
- **DevDependencies** are only necessary to *develop* applications.

package-lock.json: This is an auto-generated and modified file that gets updated whenever npm does an operation related to node_modules or package.json. It provides version information for all packages installed into node_modules by the npm client.

README.md: This file contains the description of the project. It contain information which we would like to provide to the users before they start using the app. It contains basic documentation for your project, pre-filled with CLI command information. Always make sure to enhance it with project documentation so that anyone checking out the reputation can build your application.

The structure of the file is as follows:

DemoAngularApp1

This project was generated with [[Angular CLI](#)](<https://github.com/angular/angular-cli>) version 8.2.2.

Development server

Run `ng serve` for a dev server. Navigate to `http://localhost:4200/`. The app will automatically reload if you change any of the source files.

Code scaffolding

Run `ng generate component component-name` to generate a new component. You can also use `ng generate directive|pipe|service|class|guard|interface|enum|module`.

Build

Run `ng build` to build the project. The build artifacts will be stored in the `dist/` directory. Use the `--prod` flag for a production build.

Running unit tests

Run `ng test` to execute the unit tests via [[Karma](#)](<https://karma-runner.github.io>).

Running end-to-end tests

Run `ng e2e` to execute the end-to-end tests via [[Protractor](#)](<http://www.protractortest.org/>).

Further help

To get more help on the Angular CLI use `ng help` or go check out the [[Angular CLI](#)

[README](#)](<https://github.com/angular/angular-cli/blob/master/README.md>).

Angular Architecture:

Angular is a platform and framework which is used to create client applications in HTML and TypeScript.

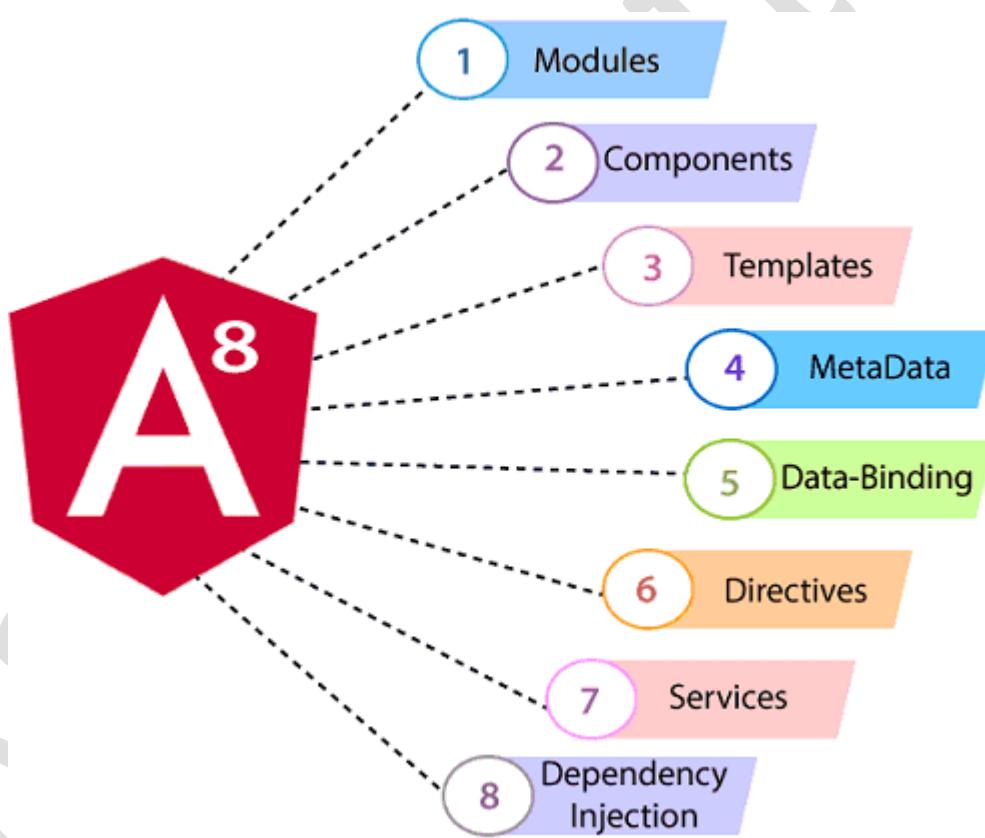
Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you can import into your apps.

NgModules are the basic building blocks of an Angular application. They provide a compilation context for components. An Angular app is defined by a set of NgModules and NgModules collect related code into functional sets.

An Angular app always has at least a root module which enables bootstrapping, and typically has many other feature modules.

- Components define views, which are the sets of screen elements that are chosen and modified according to your program logic and data by Angular.
- Components use services, which provide specific functionality not directly related to views. Service providers can be injected into components as dependencies, making your code modular, reusable, and efficient.

Key parts of Angular 8 Architecture:



Angular Components

In Angular, both components and services are simply classes, with *decorators* that mark their type and provide metadata that tells Angular how to use them.

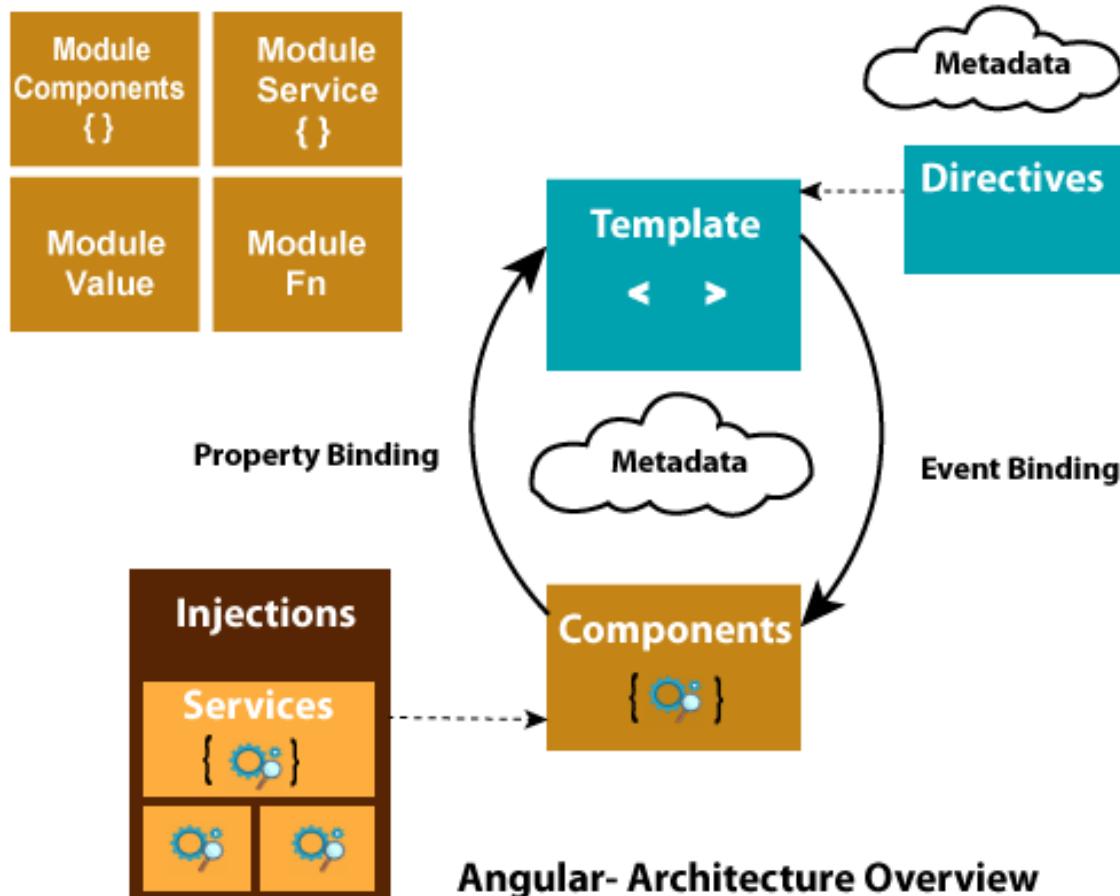
Every Angular application always has at least one component known as root component that connects a page hierarchy with page DOM. Each component defines a class that contains application data and logic, and is associated with an HTML template that defines a view to be displayed in a target environment.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Metadata of Component class

- The metadata for a component class associates it with a template that defines a view. A template combines ordinary HTML with Angular directives and binding markup that allow Angular to modify the HTML before rendering it for display.
- The metadata for a service class provides the information Angular needs to make it available to components through dependency injection (DI).

An app's components typically define many views, arranged hierarchically. Angular provides the **Router** service to help you define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.



Angular- Architecture Overview

Modules

Angular **NgModules** are different from other JavaScript modules. Every Angular app has a *root module*, conventionally named **AppModule**, which provides the bootstrap mechanism that launches the application.

Generally, every Angular app contains many functional modules.

Some important features of Angular Modules:

- Angular NgModules import the functionalities form other NgModules just like other JavaScript modules.
- NgModules allow their own functionality to be exported and used by other NgModules. For example, if you want to use the router service in your app, you can import the Router NgModule.



Template, Directives and Data Binding

In Angular, a template is used to combine HTML with Angular Markup and modify HTML elements before displaying them. Template directives provide program logic, and binding markup connects your application data and the DOM.

There are two types of data binding:

1. Event Binding: Event binding is used to bind events to your app and respond to user input in the target environment by updating your application data.

2. Property Binding: Property binding is used to pass data from component class and facilitates you to interpolate values that are computed from your application data into the HTML.

Before a view is displayed, Angular evaluates the directives and resolves the binding syntax in the template to modify the HTML elements and the DOM, according to your program data and logic. Angular supports *two-way data binding*, meaning that changes in the DOM, such as user choices, are also reflected in your program data.

Your templates can use *pipes* to improve the user experience by transforming values for display. For example, use pipes to display dates and currency values that are appropriate for a user's locale. Angular provides predefined pipes for common transformations, and you can also define your own pipes.

Services and Dependency Injection:

For data or logic that isn't associated with a specific view, and that you want to share across components, you create a service class.

A service class definition is immediately preceded by the `@Injectable()` decorator. The decorator provides the metadata that allows other providers to be injected as dependencies into your class.

Dependency injection (DI) lets you keep your component classes lean and efficient. They don't fetch data from the server, validate user input, or log directly to the console; they delegate such tasks to services.

Routing

In Angular, **Router** is an NgModule which provides a service that facilitates developers to define a navigation path among the different application states and view hierarchies in their app.

It works in the same way as a browser's navigation works. i.e.:

- Enter a URL in the address bar and the browser will navigate to that corresponding page.
- Click the link on a page and the browser will navigate to a new page.
- Click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages you've seen.

How does Router work?

The router maps URL-like paths to views instead of pages. When a user performs an action, such as clicking a link that would load a new page in the browser, the router intercepts the browser's behaviour, and shows or hides view hierarchies.

If the router determines that the current application state requires particular functionality, and the module that defines it hasn't been loaded, the router can *lazy-load* the module on demand.

The router interprets a link URL according to your app's view navigation rules and data state. You can navigate to new views when the user clicks a button or selects from a drop box, or in response to some other stimulus from any source. The router logs activity in the browser's history, so the back and forward buttons work as well.

To define navigation rules, you associate *navigation paths* with your components. A path uses a URL-like syntax that integrates your program data, in much the same way that template syntax integrates your views with your program data. You can then apply program logic to choose which views to show or to hide, in response to user input and your own access rules.



Bootstrapping an Angular Application:

Seeing the complexity of the Angular file structure, the first question that comes to the mind is how does it actually works?

"Bootstrapping is the technique or process of initializing or loading the whole Angular Application."

Understanding the bootstrapping process gives you a great insight of the Angular architectural design as well as the connection of various parts of angular like modules, components, directives, services, dependency injection, pipes with each other.

It also gives you an understanding of how Angular segregate their functionalities, still make them work together as a whole application.

If you are the one who gets scared of the long file structure; believe it it's a very straightforward procedure, so let's get started.

The file structure of the Angular application is as follows:

▽ DemoAngularApp1	●
> e2e	●
> node_modules	
▽ src	●
> app	●
> assets	●
> environments	●
★ favicon.ico	A
↳ index.html	A
TS main.ts	A
TS polyfills.ts	A
# styles.css	A
TS test.ts	A
⚙ .editorconfig	A
❖ .gitignore	A
{ angular.json	A
≡ browserslist	A
K karma.conf.js	A
{ package-lock.json	A
{ package.json	A
ⓘ README.md	A
{ tsconfig.app.json	A
TS tsconfig.json	A
{ tsconfig.spec.json	A
{ tslint.json	A

The Angular takes the following steps to load our first view.

1. Loading the Angular Core Libraries
2. Loading the Main entry point
3. Loading the Root Module
4. Loading the Root Component
5. Loading the Template

First, our application needs to load the Angular Components. This is done from the index.html.

Loading Index.html

The index.html is the first page that is served by our web host. Let us look at index.html, which was created by Angular CLI in our Angular Application.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>DemoAngularApp1</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

There are no JavaScript files included in the index.html. Neither can you see a stylesheet file. The body of the files has the following HTML

```
<app-root></app-root>
```

How do Angular loads? The Answer lies in the Module bundler Webpack. To find out let us build our application

Building Application

We run our application using the (NPM Package manager) command **npm start**. This command actually translates into **ng serve**.

ng serve does build our application but does not save the compiled application to the disk. It saves it in memory and starts the development server.

To Build & create a distribution copy of our application, open the command prompt and run the following command

ng build

The **ng build** is Angular CLI command. This command builds our application and copies the output files to the **dist folder**. Now open the **dist folder** and open the **index.html**.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

```

<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>DemoAngularApp1</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
  <script src="polyfills-es5.js" nomodule defer></script>
  <script src="polyfills-es2015.js" type="module"></script>
  <script src="styles-es2015.js" type="module"></script>
  <script src="styles-es5.js" nomodule defer></script>
  <script src="runtime-es2015.js" type="module"></script>
  <script src="vendor-es2015.js" type="module"></script>
  <script src="main-es2015.js" type="module"></script>
  <script src="runtime-es5.js" nomodule defer></script>
  <script src="vendor-es5.js" nomodule defer></script>
  <script src="main-es5.js" nomodule defer></script></body>
</html>

```

The above is our index.html after our application is built. You can see that the many **script files** are added to our original **index.html**.

These files are added by the **Webpack module loader**.

What is Webpack?

Webpack is a bundler. Webpack scans our application looking for JavaScript files and merges them into one (or more) big file. Webpack has the ability to bundle any kind of file like JavaScript, CSS, SASS, LESS, images, HTML, & fonts etc.

The Angular CLI uses Webpack as module bundler. The Webpack needs a lots configuration option to work correctly. The Angular CLI sets up all these options behind the scene.

The Webpack traverses through our application looking for JavaScript and other files and merges all of them into one or more bundles. In our example application, it has created few additional files. The Angular core files along with all other files are now merged and is part of these additional files.

So when index.html is loaded, the Angular loaded.

Loading the Entry point

The Angular is now loaded via our index.html page. The next step is to find out the entry point of our application. The entry point of our application is **main.ts**

angular.json

The Angular finds out about the entry point of our application from the configuration file angular.json. This file is located in root folder of our application. The relevant part of the angular.json is shown below:

```
"main": "src/main.ts"
```

Note the main property point towards to **main.ts**. The main.ts file is located in the src folder of the root folder.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Loading the Root module

Every application in Angular must have one root module. The root module must be loaded first. The Angular loads the main.ts file to locate the root module. Our main.ts file is as shown below:

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

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

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

Let us look the above code in detail.

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

This line of the code import the module **platformBrowserDynamic** from the path '@angular/platform-browser-dynamic'.

What is platformBrowserDynamic

platformBrowserDynamic is the module, which is responsible loading the Angular application in the desktop browser.

Angular Applications can be bootstrapped in many ways. Our Application can be loaded in many different environments. For example, we can load our application in a Desktop Browser or in a mobile device with ionic or NativeScript.

platformBrowserDynamic is the library this is used to **bootstrap angular application in the Desktop browser**

If you are using the NativeScript, then you will be using platformNativeScriptDynamic from "nativescript-angular/platform" library and will be calling **platformNativeScriptDynamic().bootstrapModule(AppModule)**.

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

This line of code imports AppModule. The AppModule is our Root Angular Module.

```
platformBrowserDynamic().bootstrapModule(AppModule)
```

Finally, we instruct the platformBrowserDynamic to load the root module by invoking the bootstrapModule and giving it the reference to our Angular Root module 'AppModule'. This is how the AppModule acts as the root module and loads at the start of the application.

Every application has at least one Angular module, the root module that you bootstrap to launch the application. By convention, it is usually called AppModule.



Loading the Root Component

The angular bootstrapping loads our root module **AppModule**. The AppModule is located under the folder **src/app**. The Code of our Root module is shown below:

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

The Angular applications are organized as modules. Every application built in Angular must have at least one module. The module, which is loaded first when the application is loaded is called as root module.

The Every Angular module must have at least one root component. The root component is loaded, when the module is loaded by the Angular.

In our example, **AppComponent** is our root component. Hence first we import it.

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

We use **@NgModule** class decorator to define a Module and provide metadata about the Modules.

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

The @NgModule takes a metadata object with the properties:

declarations

The Declarations array contains the list of components, directives, pipes & services that belong to this Angular Module. We have only one component in our application AppComponent.

imports

We need to list all the external modules required including other Angular modules, that is used by this Angular Module

providers:

A provider is an instruction to the Dependency Injection system on how to obtain a value for a dependency. Most of the time, these dependencies are services that you create and provide. (The service providers)

bootstrap

The component that angular should load, when this Angular Module loads. The component must be part of this module. We want AppComponent load when AppModule loads, hence we list it here.

The Angular reads the bootstrap metadata and loads the AppComponent.

Loading the Template

Finally, we arrive at AppComponent, which is the root component of the AppModule. The code of our AppComponent is shown below:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title:string = 'DemoAngularApp1';
}
```

The Class AppComponent is decorated with **@Component Class Decorator**.

The @Component class decorator provides the metadata about the class to the Angular. It has 3 properties in the above code. **Selector, templateUrl & styleUrls**

templateUrl

This property contains HTML template, which is going to be displayed in the browser.

styleUrls

The External styles define CSS in a separate file and refer to this file in styleUrls.

selector

This property specifies the CSS Selector, where our template will be inserted into the HTML.

Now if we have a look of **AppComponent** we will see the **selector** name in **@component** decorator as ‘app-root’.

The **index.html** file has **HTML tag <app-root>** with the same name as the selector name of the **AppComponent**.

index.html:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>DemoAngularApp1</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

The Angular locates ‘app-root’ in our **index.html** and renders our template between those tags.

This is how **AppComponent** is the first component that gets rendered on bootstrap into the view. As Angular has hierarchical component structure you can embed other components into the template of the **AppComponent**.

This helps in keeping the **index.html** file clean.

Hence, the bootstrap process loads **main.ts** which is the **main entry point of the application**.

The **AppModule** operates as the **root module** of our application. The module is configured to use **AppComponent** as the component to bootstrap, and will be rendered on any **app-root** HTML element encountered.



Angular – Components:

Components are the key features of Angular. The whole application is built by using different components. The core idea behind Angular is to build components. They make your complex application into reusable parts which you can reuse very easily.

Components are the most basic building block of a UI in Angular applications and it controls views (HTML/CSS). They also communicate with other components and services to bring functionality to your applications. An angular application contains a tree of angular components.

Technically components are basically TypeScript classes in which you can create your own properties and methods according to your needs that interact with the HTML files of the components, which get displayed on the browsers.

The most important feature of any Angular application is the component that controls View or the template that we use. In general, we write all the application logic for the view that is assigned to this component.

Therefore, an Angular application is just a tree of such Components, when each Component is processed, it recursively processes its children Components. At the root of this tree is the top level component, the main component.

When we bootstrap an Angular application, we tell the browser to render that top level root Component which renders its child component and so on.

So major part of the development with Angular is done in the components.

The file structure of the app component and it consists of the following files which are created by default whenever we create a new angular project using angular cli:

- app.component.css
- app.component.html
- app.component.spec.ts
- app.component.ts
- app.module.ts

And if you have selected angular routing during your project setup, files related to routing will also get added and the files are as follows:

- app-routing.module.ts

If you open up the **app.module.ts** file, it has some libraries which are imported and also a declarative which is assigned the app component as follows:

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Angular apps are modular and Angular has its own modularity system called **NgModules**. They can contain components, service providers, and other code files whose scope is defined by the containing **NgModule**. They can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules. Every Angular app has at least one NgModule class, the root module, which is conventionally named **AppModule** and resides in a file named **app.module.ts**. You launch your app by **bootstrapping** the root **NgModule**.

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

NgModule metadata

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

- declarations: The components, *directives*, and *pipes* that belong to this NgModule.
- imports: Other modules whose exported classes are needed by component templates declared in *this* NgModule.
- providers: Creators of services that this NgModule contributes to the global collection of services; they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)
- bootstrap: The main application view, called the root component, which hosts all other app views. Only the root NgModule should set the bootstrap property.

The declarations include the `AppComponent` variable, which we have already imported. This becomes the parent component.

Now, angular-cli has a command to create your own component. However, the app component which is created by default will always remain the parent and the next components created will form the child components.

Let's see the `AppComponent` code snippet in the application.

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'DemoAngularApp1';
}
```

Let's take a look at each of these 3 definition sections in detail.

1. Component Imports

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

The way you make a class component is by importing the `Component` member from `@angular/core` library: some components may have dozens of imports based on the component's needs.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

2. The Component Decorator

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

The **@Component** decorator identifies the class immediately it as a component class, and specifies its metadata. It's not a component until you mark it as one with the **@Component** decorator.

The **@Component** is using the component that was imported from the above import line. This is what makes this class a component. Within the component, it has a variety of configuration properties (metadata) that help to define this component.

- **selector:** this is the name of the label to which the component is applied. For example: <app-root> Loading ... </app-root> in index.html.
- **templateUrl:** It defines the HTML template associated with this component. You can also use the **template** property to define inline HTML.
- **styleUrls:** This is an array of relative paths to where the component can find the styles used to style the HTML view. You can also use the **styles** property to define inline CSS.

There are other properties that can be defined within the component decorator based on the component's needs.

3. The Component Class

```
export class AppComponent {
  title = 'DemoAngularApp1';
}
```

Lastly, we have the core of the component that is the component class.

This is where the various properties and methods are defined. All properties and methods defined here are accessible from the template. Likewise, events occurring in the template are accessible within the component.

Become a component when we add **@Component** metadata or we can say that when you decorate the simple class with the **@Component** decorator, it becomes the component.

To make this class an angular component, we need to decorate the above class using the **@Component** decorator that is present in **@angular/core** library. That's why we have to import **@angular/core** even using the following piece of code.

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

Note: The normal TypeScript class will become a component class once decorated with **@component** decorator.

Finally, a component must belong to an **NgModule** to be available for another component or application. To specify that a component is a member of an **NgModule**, it must be included in the declaration field of that **@NgModule** metadata (**NgModule**).

In the **app.module.ts** file, you have some imported libraries and also a declarative to which the **AppComponent** is assigned. So, declarations include the **AppComponent** variable, which we have already imported. This becomes the parent component.

AppComponent is the **root** component of our angular application. It's the base of the tree of components of our application and it's the first component that gets inserted in the browser DOM.

A component is comprised of three things:

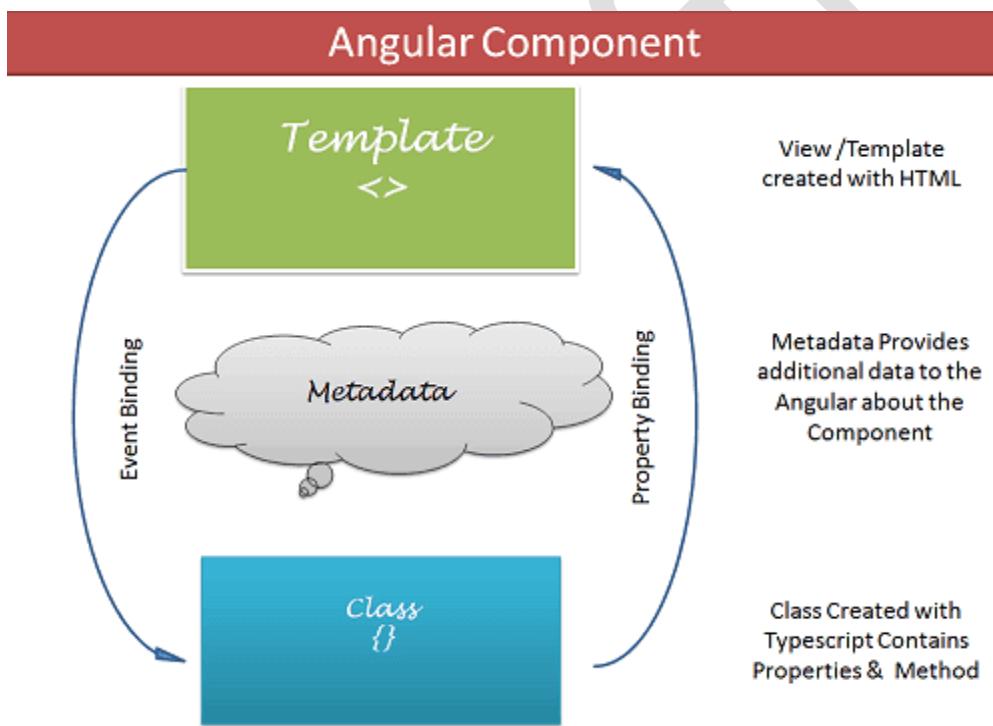
- A **component class**, which handles data and functionality.
- An **HTML template**, which determines what is presented to the user.
- **Component-specific styles** that define the look and feel of HTML View.

The Angular applications will have lots of components. Each component handles a small part of UI. These components work together to produce the complete user interface of the application

How to create a new component?

While creating a new component you need to understand the main building blocks of the components. The Components consists of three main building blocks as follows:

- Class
- Metadata
- Template



Building blocks of the Angular Component:

Template (View)

The template defines the Layout of the View and defines what is rendered on the page. Without the template, there is nothing for Angular to render to the DOM.

The Templates are created with HTML. You can add Angular directives and bindings on the template.

There are two ways you can specify the Template in Angular.

1. Defining the Template Inline
2. Provide an external Template



Class

The class is the code associated with Template (View). The Class is created with the Typescript. Class Contains the Properties & Methods. The Properties of a class can be bind to the view using Data Binding.

The simple Angular Class

```
export class AppComponent
{
  title: string = "app"
}
```

The Component classes in Angular are suffixed with the name “Component”, to easily identify them.

Metadata

Metadata Provides additional information about the component to the Angular. Angular uses this information to process the class. The Metadata is defined with a **decorator**.

A decorator is a function that adds metadata to class, its methods & to its properties. The Components are defined with a @Component class decorator. It is @Component decorator, which defines the class as Component to the Angular

@Component decorator

A class becomes a Component when Component Decorator is used. A Decorator is always prefixed with @. The Decorator must be positioned immediately before the class definition.

Important Component metadata properties

selector:

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

styles/styleUrls

The CSS Styles or style sheets that this component needs. Here we can use either external stylesheet (using styleUrls) or inline styles (using styles). The styles used here are specific to the component.

template/templateUrl

The HTML template that defines our View. It tells angular how to render the Component's view. The templates can be inline (using a template) or we can use an external template (using a templateUrl). The Component can have only one template. You can either use inline template or external template and not both.

providers

An array of providers for services that the component requires. The providers are the services, that our component going to use. The Services provide service to the Components or to the other Services.

Creating the Component:

The creation of the Angular component requires you to follow these steps:

1. Create the Component file
2. Import the required external Classes/Functions
3. Create the Component class and export it
4. Add @Component decorator
5. Add metadata to @Component decorator
6. Create the Template
7. Create the CSS Styles
8. Register the Component in Angular Module

Creating the Component File:

Open Visual Studio Code >> Go to your project source folder i.e. src >> Expand the **app** folder and create a new folder named "**user**". Right click on **app** folder then click "**New Folder**" option and named as "**user**".

Now, create the component within **user** directory. Right click on the **user** directory and create a new file named as "**user.component.ts**". It is the newly created component.

Import the Angular Component Object:

Before we use any Angular functions or classes, we need to tell Angular how and where to find it. This is done using the Import statement. The Import statement is similar to the using statement in C#, which allows us to use the external namespaces in our class

To define the Component class, we need to use the **@Component** decorator. This function is found in the Angular core library. So we import it in our class as shown below:

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

Create the Component Class and export it:

The third step is to create the Component class using the export keyword. The export keyword allows this component class to be used in other components by importing. The UserComponent class is shown below:

```
export class UserComponent{
  title: string = "Welcome to User Section";
  public ButtonClick(){
    alert("You have clicked Button !!!");
  }
}
```

Note we are using Pascal case naming conventions for the class name. In the above class defines a property named **title** with a default value "**Welcome to User Section**" and a function named "**ButtonClick**" to display an alert message when user clicks a button in view page.

Add **@Component** decorator:

The next step is to inform the Angular that this is a Component class. This is done by adding the **@Component** decorator to the above class. The decorator must be added immediately above the class as shown below:

```
@Component({
})
export class UserComponent{
  title: string = "Welcome to User Section";
  public ButtonClick(){
    alert("You have clicked Button !!!");
  }
}
```

Add metadata to **@Component** decorator:

The next step is to add the metadata to the **@component decorator**. Add the following to the component metadata:

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

```
@Component({
  selector: 'user-section', // selector to be used inside .html file.
  templateUrl: './user.component.html', // reference to the html file created in the new component.
  styleUrls: ['./user.component.css'] // reference to the style file.
})
```

selector:

The angular places the view (template) inside the selector user-section

templateUrl:

In the above example, we have used external template using templateUrl metadata. The templateUrl points to the external HTML file **user.component.html**.

styleUrls:

Defines the styles for our template. The metadata points to the external CSS file **user.component.css**. The Component specific CSS styles can be specified here.

Create the Template (View):

Template is nothing but an HTML file, which component must display it to the user. The Angular knows which template display, using the templateUrl metadata, which points to **user.component.html**.

```
<h1>{{title}}</h1>
<div align="center">
<input type="button" (click)="ButtonClick()" value="Click Me" />
</div>
```

Note that **title inside the double curly bracket**. When rendering the view, the Angular looks for **title property** in our component and binds the property to our view. This is called **data binding**. And button's **click event inside the parenthesis** is associated with a function called **ButtonClick()**. When rendering the view, the Angular looks for **ButtonClick() function** in our component and binds the function to a click event of button in our view. This is called **event binding**.

Add the Styles:

The next step is to add the CSS Styles. The styleUrls metadata tells Angular, where to find the CSS File. This property points to external file **user.component.css**

Note that styleUrls metadata can accept multiple CSS Files.

```
h1{
  text-align: center;
  background-color: yellow;
  color: red;
  font-weight: bold;
  border: 5px solid blue;
}
```

Register the Angular Component in Angular Module:

We have created the Angular Component. The next step is to register it with the Angular Module.

The Angular Module organizes the components, directives, pipes and services that are related and arrange them into cohesive blocks of functionality.

It is a class that is decorated with **@NgModule** decorator in root module file i.e. **app.module.ts** in **app** folder.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.module.ts:

```

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';          Default Root Component in app folder
import {UserComponent} from './user/user.component'      Newly Created Component in user folder

@NgModule({
  declarations: [
    AppComponent,           Here it is added in declarations and will behave as a parent component
    UserComponent          Here it is added in declarations and will behave as a child component
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]          for bootstrap, the main component i.e. AppComponent is given.
})
export class AppModule { }

```

We use **@NgModule** class decorator to define a Module and provide metadata about the Modules.

We add all the components, pipes and directives that are part of this module to the declarations array. We add all the other modules that are used by this module to imports array. We include the services in the providers array.

The Component that angular should load, when the app.module is loaded is assigned to bootstrap property.

The UserComponent imported

```
import {UserComponent} from './user/user.component'
```

And then added to the declarations array.

```

@NgModule({
  declarations: [
    AppComponent, UserComponent
  ]
.....

```

We want AppComponent (Root Component) to be loaded when Angular starts, thus we assign it to bootstrap property

```
bootstrap: [AppComponent]
})
```

That's it.

Finally, run the application from the command line using ng serve (or npm start).

```
>ng serve or ng s
>ng serve --open or ng s --o
>npm start
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

But when we run the project, we do not see anything related to the new component getting displayed in the browser.

The browser displays the following screen:



Here are some links to help you start:

- [Tour of Heroes](#)
 - [CLI Documentation](#)
 - [Angular blog](#)

We do not see anything related to the new component being displayed. The new component created has an .html file with following details:

```
<h1>{{title}}</h1>
<div align="center">
<input type="button" (click)="ButtonClick()" value="Click Me" />
</div>
```

But we are not getting the same in the browser. Let us now see the changes required to get the new components contents to get displayed in the browser.

The selector '**user-section**' is created for new component from **user.component.ts** as shown below:

```
import { Component } from '@angular/core';
@Component({
  selector: 'user-section',
  templateUrl: './user.component.html',
  styleUrls: ['./user.component.css']
})
export class UserComponent{
  title: string = "Welcome to User Section";
  public ButtonClick(){
    alert("You have clicked Button !!!");
  }
}
```

The selector, i.e., **user-section** needs to be added in the **app.component.html** with modified content, i.e., the main parent created by default as follows:

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

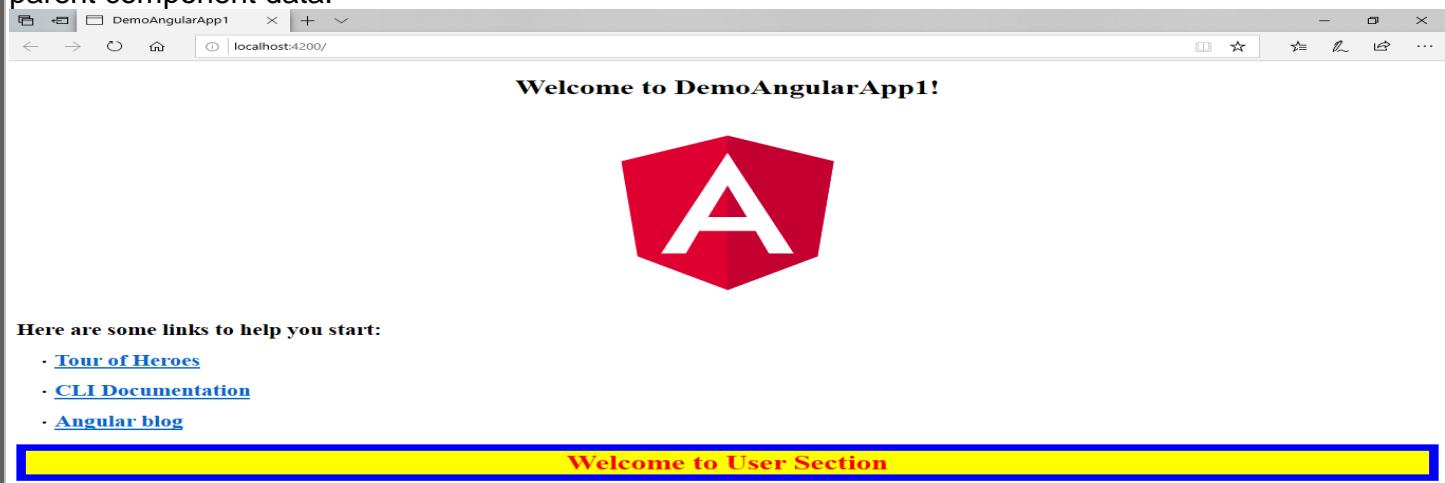
<!--The content below is only a placeholder and can be replaced.-->

```

<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  
</div>
<h2>Here are some links to help you start: </h2>
<ul>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/tutorial">Tour of
Heroes</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/cli">CLI
Documentation</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h2>
  </li>
</ul>
<router-outlet></router-outlet>
<user-section></user-section>

```

When the **<user-section></user-section>** tag is added, all that is present in the .html file, i.e., **user.component.html** of the new component created will get displayed on the browser along with the parent component data.



Welcome to DemoAngularApp1!



Welcome to User Section

Here are some links to help you start:

- [Tour of Heroes](https://angular.io/tutorial)
- [CLI Documentation](https://angular.io/cli)
- [Angular blog](https://blog.angular.io/)

Similarly, we can create components and link the same using selector in the **app.component.html** file as per our requirements.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Creating component using Angular CLI:

Now, angular-cli has a command to create your own component. However, the app component which is created by default will always remain the parent and the next components created will form the child components.

Create a component using the following command in the integrated terminal of Visual Studio Code:

Syntax

ng generate component component-name

Or

ng g c component-name

- **ng** calls the angular CLI
- **g** is the abbreviation of generate (note, you can use the whole word generate if you wish)
- **c** is the abbreviation of component (note, you can use the whole word component if you wish). component is the type of element that is going to be generated.
- **component-name** is the **name of the component**

Let's see how to create a new component by using command line.

Open Command prompt or terminal and stop **ng serve** command if it is running on the browser.

Type **ng generate component student** to create a new component named student as following:

PS D:\AngularApps\DemoAngularApp1> ng generate component student

You can also use a shortcut **ng g c student** to do the same task as following:

PS D:\AngularApps\DemoAngularApp1> ng g c student

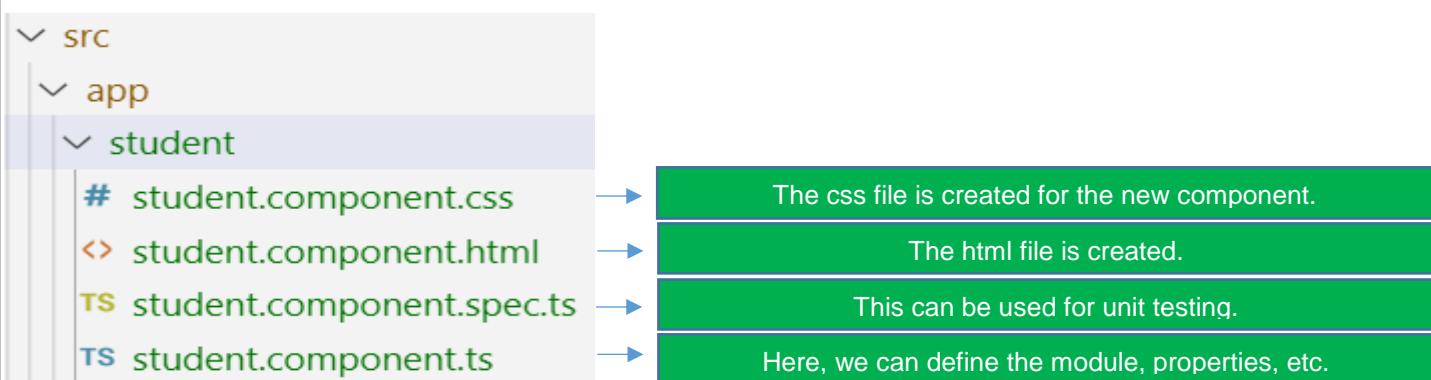
When you run the above command on the command line or terminal, you will receive the following output:

PROBLEMS TERMINAL ... 1: powershell + └ ┌ └ └ └ └ └ └ └

```
PS D:\AngularApps> cd DemoAngularApp1
PS D:\AngularApps\DemoAngularApp1> ng generate component student
CREATE src/app/student/student.component.html (22 bytes)
CREATE src/app/student/student.component.spec.ts (635 bytes)
CREATE src/app/student/student.component.ts (273 bytes)
CREATE src/app/student/student.component.css (0 bytes)
UPDATE src/app/app.module.ts (542 bytes)
PS D:\AngularApps\DemoAngularApp1>
```

Now, if we check the file structure, we will get the new student folder created under the **src/app** folder.

The following files are created in the student folder:



The changes are added to the **app.module.ts** file as follows:

app.module.ts:

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { UserComponent } from './user/user.component';
import { StudentComponent } from './student/student.component' //includes the student component we created

@NgModule({
  declarations: [
    AppComponent,
    UserComponent,
    StudentComponent // here it is added in declarations and will behave as a child component
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent] //for bootstrap, the AppComponent is given as the main component.
})
export class AppModule { }
```

The **student.component.ts** file is generated as follows –

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

@Component({
  selector: 'app-student',
  templateUrl: './student.component.html',
  styleUrls: ['./student.component.css']
})
export class StudentComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }
}
```

If you see the above **student.component.ts** file, it creates a new class called **StudentComponent**, which implements **OnInit**. In, which has a constructor and a method called **ngOnInit ()**. **ngOnInit()** is called by default whenever the class is run.

Let's see how the flow works. As you already know, the app component, which is created by default, becomes the main component. Any component added later becomes the child component.

When we click on the URL on <http://localhost:4200/> browser, first run the index.html file shown below:

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Index.html:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>DemoAngularApp1</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

The above is the normal html file and we do not see anything printed in the browser. Let's see at the tag in the body section.

```
<app-root></app-root>
```

This is the root tag created by Angular by default. This tag has the reference in the main.ts file in your application.

main.ts:

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';
if (environment.production) {
  enableProdMode();
}
platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

AppModule is imported from the app of the main parent module and the same is given to the bootstrap module, which loads the application module.

Now let's see the file app.module.ts -

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { UserComponent } from './user/user.component';
import { StudentComponent } from './student/student.component'
@NgModule({
  declarations: [
    AppComponent,
    UserComponent,
    StudentComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Here, **AppComponent** is the specified name, which is the variable to store the reference of the **app.component.ts** and the same is specified to the bootstrap.

Now let's see the file **app.component.ts**.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  title = 'DemoAngularApp1';
}
```

The angular core is imported and is known as Component and is used in the decorator as following:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
```

In the decorator reference to the **selector**, **templateUrl** and **styleUrls** are supplied. The selector here is no more than the tag that was inserted in the index.html file we saw above.

The **AppComponent** class has a variable called **title**, which is displayed in the browser.

@Component uses the **templateUrl** called **app.component.html**, which is the following:

```
<!--The content below is only a placeholder and can be replaced.-->
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  Tour of Heroes</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/cli">CLI Documentation</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h2>
  </li>
</ul>
```

It has only the html code and the variable title in brackets. It is replaced with the value, which is present in the app.component.ts file. This is called data binding.

Now that we have created a new component that is **student**. The same is included in the **app.module.ts** file, when the command is executed to create a new component.

app.module.ts has a reference to the every new component created.

Now let's see the new files created in the student.

student.component.ts:

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-student',
  templateUrl: './student.component.html',
  styleUrls: ['./student.component.css']
})
export class StudentComponent implements OnInit {
  constructor() { }
  ngOnInit() {
  }
}
```

Here, we must also import the core. The component reference is used in the decorator.

The decorator has the **selector** called **app-student** and **templateUrl** and **styleUrls**.

The .html called **student.component.html** is as follows:

student.component.html:

```
<p>student works!</p>
```

As we saw above, we have the html code, i.e. the `<p>` tag. The style file is empty because we do not need a style at the moment. But when we run the project, we do not see anything related to the new component that is displayed in the browser. Now add something or change something and the same can be seen in the browser later. The selector, which is the **app-student** must be added in the **app.component.html** file as follows:

app.component.html:

```
<!--The content below is only a placeholder and can be replaced.-->
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  
</div>
<h2>Here are some links to help you start: </h2>
<ul>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/tutorial">Tour of Heroes</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/cli">CLI Documentation</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h2>
  </li>
</ul>
<app-student></app-student>
```

When the `<app-student> </app-student>` tag is added, everything in the .html file of the newly created component will be displayed in the browser along with the parent component data.

Let's see the new component html file and the student.component.ts file.

student.component.ts:

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-student',
  templateUrl: './student.component.html',
  styleUrls: ['./student.component.css']
})
export class StudentComponent implements OnInit {
  message = "Hello, Welcome to the Student Section";
  constructor() { }
  ngOnInit() {
  }
}
```

In the class, we have added a variable called a message and the value is "Hello, Welcome to the Student Section".

The above variable is bind in the student.component.html file as follows:

```
<h1>Student Component!</h1>
<h2>{{ message }}</h2>
```

Now, since we have included the `<app-student> </app-student>` selector in the app.component.html which is the html of the parent component, the content in the new component .html file (student.component.html) is displayed in the browser as follows:



Here are some links to help you start:

- [Tour of Heroes](#)
- [CLI Documentation](#)
- [Angular blog](#)

Student Component!

Hello, Welcome to the Student Section

Similarly, we can create components and link the same using selector in the `app.component.html` file as per our requirements.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

The component selector:

The Angular renders the components view in the DOM inside the CSS selector that we defined in the Component decorator.

```
@Component({
  selector: 'app-root',
  ....
})
```

The selector `<app-root></app-root>` is in the `index.html` file (under the application root folder).

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>DemoAngularApp1</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

When we build Angular Components, we are actually building new HTML elements. We specify the name of the HTML element in the selector property of the component metadata. And then we use it in our HTML. The Angular, when instantiating the component, searches for the selector in the HTML file and renders the Template associated with the component.

That gives several options to use our component selector.

Using CSS class name:

```
@Component({
  selector: '.app-root',
  ....
})
```

And in the HTML markup use the CSS Class name as following:

```
<div class="app-root"></div>
```

Using attribute name:

```
@Component({
  selector: '[app-root]',
  ....
})
```

And you can now use the attribute as follows:

```
<div app-root></div>
```

Using attribute name and value:

```
@Component({
  selector: 'div[app=components]',
  ....
})
```

Now you can use it in HTML markup as follows:

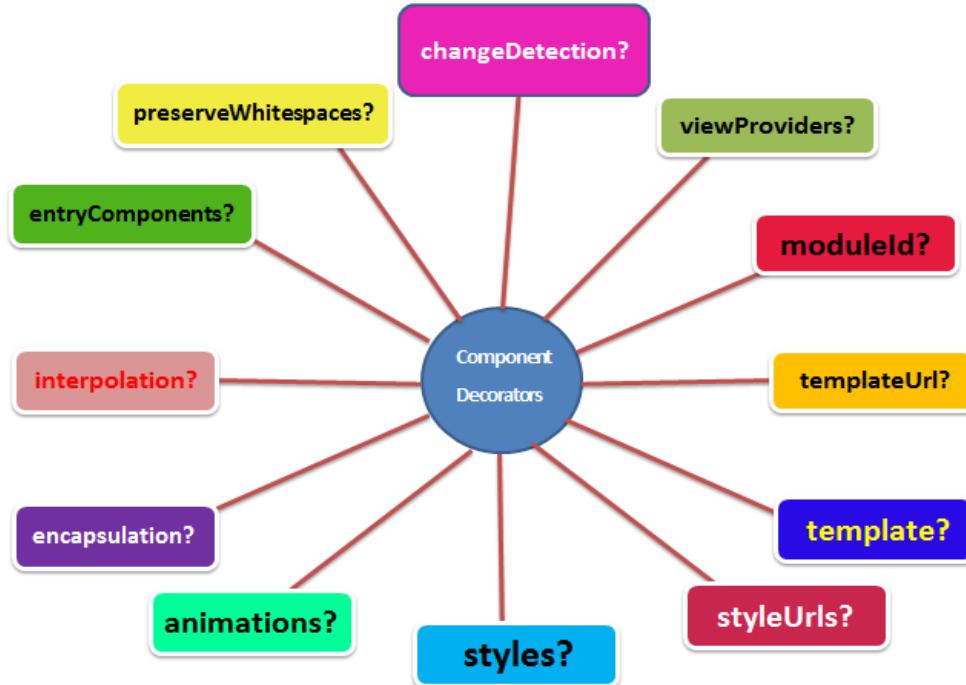
```
<div app="components"></div>
```

Component Decorators:

@Component decorator provides additional metadata that determines how to process, instantiate and use the component at runtime (the decorators in Typescript are like annotations in Java or attributes in C#)

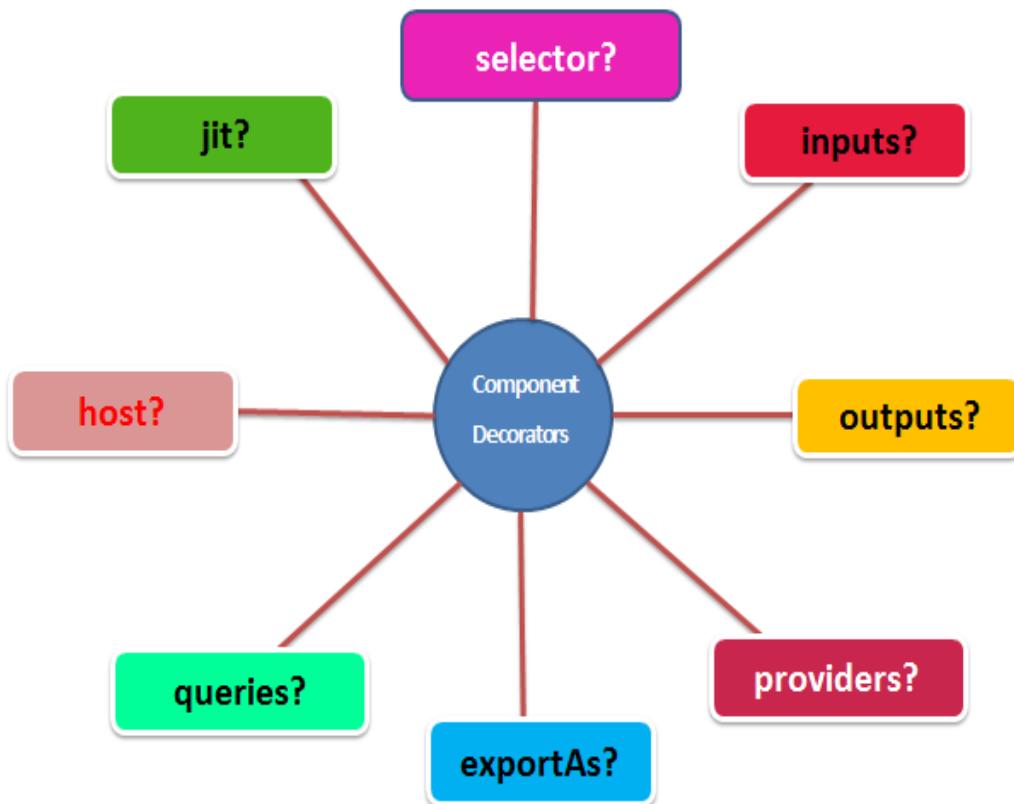
@Component Decorator accepts the required configuration object that requires information to create and display the component in real time.

```
@Component({
  changeDetection?: ChangeDetectionStrategy
  viewProviders?: Provider[]
  moduleId?: string
  templateUrl?: string
  template?: string
  styleUrls?: string[]
  styles?: string[]
  animations?: any[]
  encapsulation?: ViewEncapsulation
  interpolation?: [string, string]
  entryComponents?: Array<Type<any> | any[]>
  preserveWhitespaces?: boolean
  // inherited from core: Directive
  selector?: string
  inputs?: string[]
  outputs?: string[]
  providers?: Provider[]
  exportAs?: string
  queries?: {...}
  host?: {...}
  jit?: boolean
})
```



- **changeDetection** - change the detection strategy used by this component.
- **viewProviders** - list of providers available for this component and the view of their children.
- **moduleId** - Module ID ES/CommonJS of the file in which this component is defined.
- **templateUrl** - url in an external file that contains a template for the view.
- **template** - template defined inline template for the view.
- **styleUrls** - url list for style sheets that will be applied to the view of this component.
- **styles** - styles defined online that will be applied to the view of this component.
- **animations** - animation's list of this component.
- **encapsulation** - strategy of style encapsulation used by this component.
- **interpolation** - custom interpolation markers used in the template of this component.
- **entryComponents** - entryComponents is the list of components that are dynamically inserted into the view of this component.
- **preserveWhitespaces** - Using this property, we can remove all whitespaces from the template.

// inherited from core: Directive



- **selector** - css selector which identifies this component in a template.
- **inputs** - it is property within one component (child component) to receive a value from another component (parent component).
- **outputs** - it is property of a component to send data from one component (child component) to calling component (parent component).
- **providers** - Providers are usually singleton objects (an instance), to which other objects have access through dependency injection (DI).
- **exportAs** - name under which the component instance is exported to a template.
- **queries** - allows you to configure queries that can be inserted into the component.
- **host** - Map of class properties to host element links for events, properties, and attributes.
- **jit** - if true, the AOT compiler will ignore this directive/component and will therefore always be compiled using JIT.

template & templateUrl:

We know that the @Component decorator functions take an object and this object contains many properties. So now we will learn about the template & templateUrl properties.

We can render our HTML code within the @Component decorator in two ways.

Inline Templates:

The Inline templates are specified directly in the component decorator. In this, we will find the HTML inside the TypeScript file. This can be implemented using the "template" property.

The literal values of the template with back-ticks marks allow strings on multiple lines. It means that if you have HTML on more than one line, you should use backticks instead of single or double quotes, as shown below.

Use the following code in **app.component.ts**.

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

@Component({
  selector: 'app-root',
  template: `
    <h1>Inline Template</h1>
    <hr>
    <h3>My First Angular Application</h3>`,
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title:string = 'DemoAngularApp1';
}
```

This will be the output of the above code.



Inline Template

My First Angular Application

External Template:

The External templates define HTML in a separate file and refer to this file in **templateUrl** means in this we will find a separate HTML file instead of finding the HTML inside the TypeScript file. Here, the TypeScript file contains the path to that HTML file with the help of the "**templateUrl**" property.

Use the following code in “**app.component.ts**” file.

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title:string = 'DemoAngularApp1';
}
```

Use the following code in “**app.component.html**” file.

```
<h1>External Template</h1>
<hr>
<h3>
  Template outside of typescript file using an implementation of templateUrl property.
</h3>
```

This will be the output of the above code.



A screenshot of a Microsoft Edge browser window. The address bar shows "localhost:4200/". The page content is a single line of text: "External Template".

External Template

Template outside of typescript file using an implementation of templateUrl property.

Points to Remember:

- Each of these two must be presented in the component to link it to an external template (i.e. templateUrl) or an inline template (i.e. a template). Therefore, this is one of the properties you must have at all times.
- If we use the "template" and "templateUrl" both properties, it will generate errors.

The official [Angular Style Guide](#) recommends extracting the templates into a separate file if the view template has more than 3 lines.

Let's try to understand why it is better to extract a view template in a separate file if it is longer than 3 lines.

With an inline template:

1. We lose the features of the Visual Studio editor, like Intellisense, code-completion and formatting.
2. TypeScript code is not easy to read and understand if combined with the HTML inline template.
3. The view template is inline in a pair of backticks marks. The question that comes to mind is why we cannot include HTML in a pair of single or double quotes. The answer is "YES", we can as long as the HTML is on a single line. This means that the above code can be rewritten using a pair of single quotes as shown below.

```
template: '<h1>Inline Template</h1><hr><h3>My First Angular Application</h3>'
```

or

```
template: "<h1>Inline Template</h1><hr><h3>My First Angular Application</h3>"
```

4. If the HTML is present in more than one line, backticks should be used instead of single or double quotation marks, as shown below. If you use single or double quotes instead of backticks, an error will occur.

```
template: `
    <h1>Inline Template</h1>
    <hr>
    <h3>My First Angular Application</h3>`
```

With an external view template:

1. We have the features of Visual Studio editor, Intellisense, code -completion and formatting.
2. Not only is the code in "app.component.ts" is clean, it is also easier to read and understand.

styles and styleUrls:

We know that the decorator functions of @Component take object and this object contains many properties. So now we will learn about the styles and styleUrls properties.

We can represent our styles i.e. the CSS code within the @Component decorator in two ways.

Inline Styles:

The Inline Style are specified directly in the component decorator. In this, you will find the CSS within the TypeScript file. This can be implemented using the "styles" property.

Use the following code in "app.component.ts".

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: ['h1{color:green}', 'div{font-family: Arial; color: blue}']
})
export class AppComponent {
  title:string = 'DemoAngularApp1';
}
```

Use the following code in “app.component.html” file.

```
<h1>Inline Styles</h1>
<hr>
<div>
  Styles inside TypeScript file using an implementation of
  <b><i>styles</i></b>
  property, this property takes an array of strings that contain CSS code.
</div>
```

This will be the output of the above code.



Inline Styles

Styles inside TypeScript file using an implementation of **styles** property, this property takes an array of strings that contain CSS code.

External Styles:

The External styles define CSS in a separate file and refer to this file in styleUrls means in this we will find a separate CSS file instead of finding a CSS within the TypeScript file. Here, the TypeScript file contains the path to that style sheet file with the help of the "**styleUrls**" property.

Use the following code in “app.component.ts” file.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title:string = 'DemoAngularApp1';
}
```

Use the following code in “app.component.html” file.

```
<h1>External Styles</h1>
<hr>
<div>
  Styles outside TypeScript file called an external styles using an implementation of
  <b><i>styleUrls</i></b>
  property, this property takes path of style sheet file where the css code is present.
</div>
```

Use the following code in “app.component.css” file.

```
h1{color:green}
div{font-family: Arial; color: blue}
```

This will be the output of the above code.



Styles outside TypeScript file called an external styles using an implementation of **styleUrls** property, this property takes path of style sheet file where the css code is present.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Points to Remember:

- It is not mandatory to use any of the properties. If we do not want to implement any style on a web page, we can leave both properties.
- If we use both properties, "styles" and "styleUrls", then we always represent the style from an external file, i.e. use the "styleUrls" property.

preserveWhitespaces:

We know that the decorator functions of @Component take object and this object contains many properties. So now we will learn about the **preserveWhitespaces** property.

Using this property, we can remove all whitespaces from the template. It takes a Boolean value, that is:

If it is **false**, it will remove all whitespace from the compiled template.

If it is **true**, it will not remove whitespace from the compiled template.

preserveWhitespaces With true

Use the following code in app.component.ts file.

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

@Component({
  selector: 'app-root',
  template: `<h1>preserveWhitespaces With true</h1>
<hr />
<div> Using this property, we can remove all whitespaces from the template.<br />
it takes a Boolean value, that is:<br />
If it is false, it will remove all whitespace from the compiled template.<br />
If true, it will not remove whitespace from the compiled template.
</div>
<button>Angular2</button> <button>Angular4</button> <button>Angular5</button>
<button>Angular6</button> <button>Angular7</button>`,
  styles: ['h1{color:red }', 'div{font-family: Arial; color: blue}'],
  preserveWhitespaces:true
})
export class AppComponent {
  title = 'DemoAngularApp1';
}
```

This will be the output of the above code.



preserveWhitespaces With true

Using this property, we can remove all whitespaces from the template.

It takes a Boolean value, that is:

If it is false, it will remove all whitespace from the compiled template.

If true, it will not remove whitespace from the compiled template.

[Angular2](#) [Angular4](#) [Angular5](#) [Angular6](#) [Angular7](#)

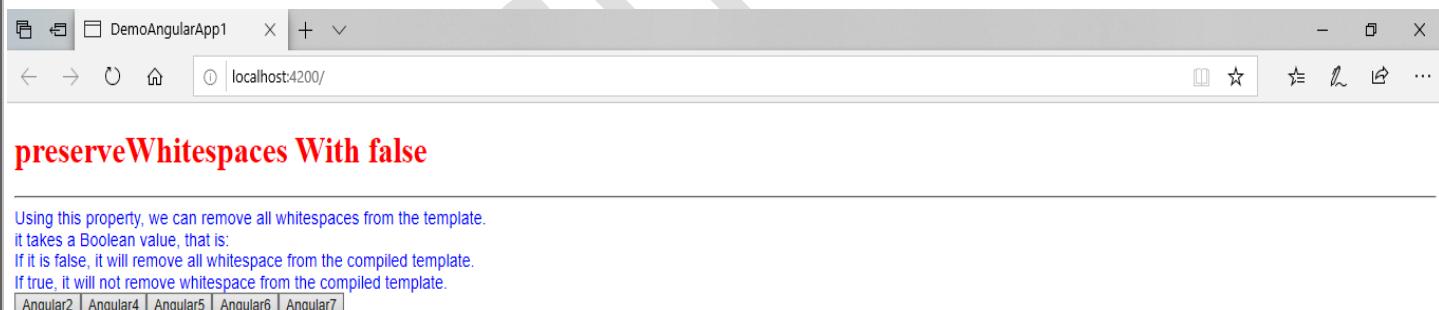
preserveWhitespaces With false

Use the following code in app.component.ts file.

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

@Component({
  selector: 'app-root',
  template: `<h1>preserveWhitespaces With false</h1>
<hr />
<div> Using this property, we can remove all whitespaces from the template.<br />
it takes a Boolean value, that is:<br />
If it is false, it will remove all whitespace from the compiled template.<br />
If true, it will not remove whitespace from the compiled template.
</div>
<button>Angular2</button> <button>Angular4</button> <button>Angular5</button>
<button>Angular6</button> <button>Angular7</button>`,
  styles: ['h1{color:red}', 'div{font-family: Arial; color: blue}'],
  preserveWhitespaces:false
})
export class AppComponent {
  title = 'DemoAngularApp1';
}
```

This will be the output of the above code.



Using this property, we can remove all whitespaces from the template.
 It takes a Boolean value, that is:
 If it is false, it will remove all whitespace from the compiled template.
 If true, it will not remove whitespace from the compiled template.

Angular2 Angular4 Angular5 Angular6 Angular7 Angular2 Angular7

viewProviders:

We know that the decorator functions of @Component take object and this object contains many properties. So now we will learn about the **viewProviders** property.

The viewProviders property allows us to make providers available only for the component's view.

When we want to use a class in our component that is defined outside the @Component () decorator function, then, first of all, we need to inject this class into our component, and we can achieve this with the help of the "viewProviders" property of a component.

Use the following code in app.component.ts file.

```

import { Component } from '@angular/core';
class DemoProvider {
  constructor() {
    console.log("Demo Provider Property");
  }
  VarDemoProvider = "VarDemoProvider";
}
class TestProvider {
  VarTestProvider = "VarTestProvider";
  constructor() {
  }
  getString(name) {
    console.log("Test Provider Property: " + name);
  }
}
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  viewProviders:[DemoProvider, TestProvider]
})
export class AppComponent {
  constructor(public demo: DemoProvider, public test: TestProvider) {
    test.getString("RakeshSoftNet Technologies");
    console.log(demo.VarDemoProvider);
    console.log(test.VarTestProvider);
  }
  title = 'DemoAngularApp1';
}

```

In this, we have created two classes, like: "DemoProvider" and "TestProvider". If we want to use these classes in our component, first inject the name of your class in the **viewProviders**.

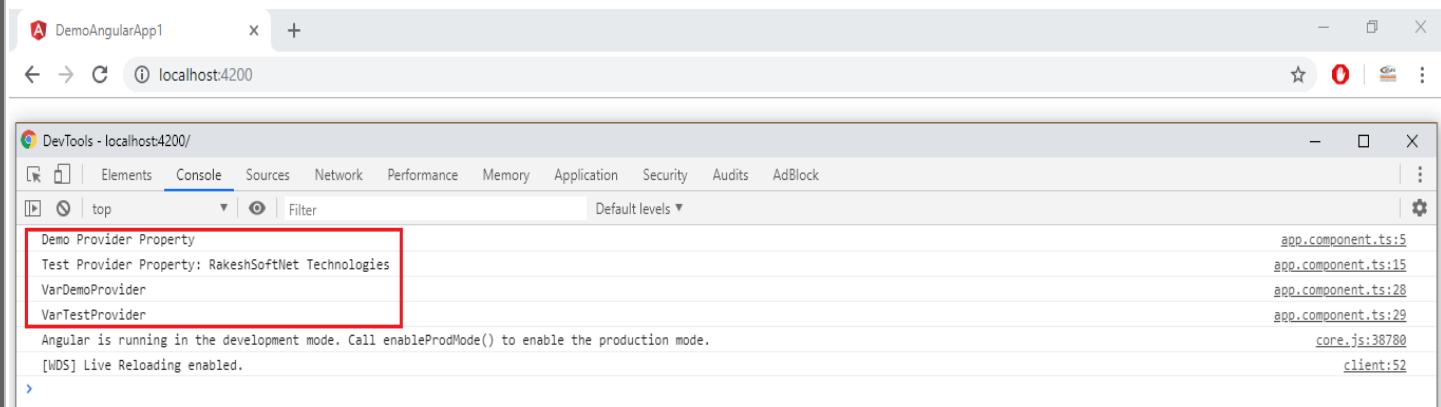
viewProviders:[DemoProvider, TestProvider]

Then, create an object into the constructor of AppComponent Class.

constructor(public demo: DemoProvider, public test: TestProvider)

So with the help of these objects (demo, test), we can use the methods of the DemoProvider class and TestProvider class in our component.

This will be the output of the above code.



The screenshot shows the browser's DevTools Console tab for localhost:4200. The output window displays the following text:

```

Demo Provider Property
Test Provider Property: RakeshSoftNet Technologies
VarDemoProvider
VarTestProvider

```

The first two lines are highlighted with a red box. The right side of the screen shows the file paths for each line of code: app.component.ts:5, app.component.ts:15, app.component.ts:28, app.component.ts:29, core.js:38780, and client.js:52.



changeDetection:

We know that the decorator functions of @Component take object and this object contains many properties. So, now we will learn about the **changeDetection** property.

Change Detection means updating the DOM every time the data is changed.

When modifying any of the models, Angular detects the changes and updates the views immediately. This is the detection of changes in Angular. The purpose of this mechanism is to ensure that the underlying views are always synchronized with their corresponding models.

A model in Angular can change as a result of one of the following scenarios:

- DOM events (click, submit, hover over, etc.)
- AJAX requests (XHR Request)
- Timers (setTimeout(), setInterval())

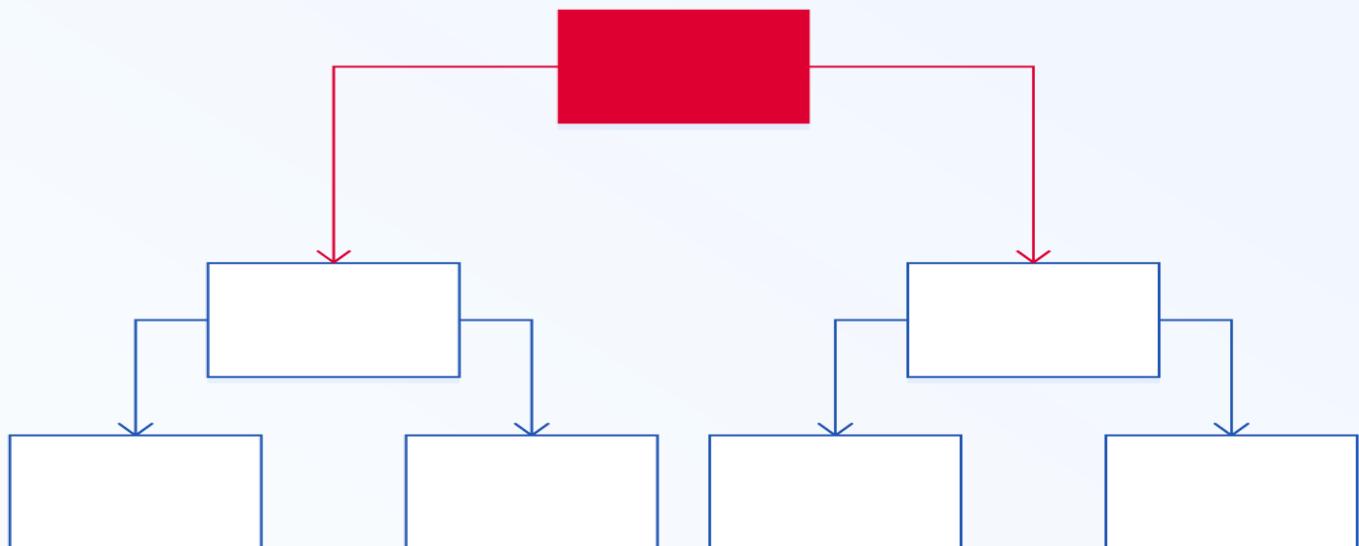
Change Detectors:

All Angular apps are made up of a hierarchical tree of components. At runtime, Angular creates a separate change detector class for every component in the tree, which then eventually forms a hierarchy of change detectors similar to the hierarchy tree of components.

Whenever change detection is triggered, Angular walks down this tree of change detectors to determine if any of them have reported changes.

The change detection cycle is always performed once for every detected change and starts from the root change detector and goes all the way down in a sequential fashion. This sequential design choice is nice because it updates the model in a predictable way since we know component data can only come from its parent.

Change Detector Hierarchy



The change detectors provide a way to keep track of the component's previous and current states as well as its structure in order to report changes to Angular.

If Angular gets the report from a change detector, it instructs the corresponding component to re-render and update the DOM accordingly.

Change Detection Strategies:

Change Detection means updating the DOM whenever data is changed. Angular provides two strategies for Change Detection such as **Default Strategy** and **OnPush Strategy**.

In its **Default** strategy, whenever any data is mutated or changed, Angular will run the change detector to update the DOM. In the **OnPush** strategy, Angular will only run the change detector when a new reference is passed to @Input() data.

To update the DOM with updated data, Angular provides its own change detector to each component, which is responsible for detecting change and updating the DOM.

Let's say we have a **MessageComponent**, as listed below:

message.component.ts:

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

@Component({
  selector: 'app-message',
  template: `
    <h2>
      Hey {{person.firstname}} {{person.lastname}} !
    </h2>
  `
})
export class MessageComponent {
  @Input() person;
}
```

In addition, we are using MessageComponent inside AppComponent as shown below:

app.component.ts

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

@Component({
  selector: 'app-root',
  template: `
    <app-message [person]='p'></app-message>
    <button (click)='changeName()'>Change Name</button>
  `
})
export class AppComponent implements OnInit{
  title = 'DemoAngularApp1';
  p: any;
  ngOnInit(): void {
    this.p = {
      firstname: 'Rakesh',
      lastname: 'Singh'
    };
  }
}
```

Let's examine the code: all we are doing is using `MessageComponent` as a child inside `AppComponent` and setting the value of `person` using the property binding.

At this point in running the application, you will get the name printed as output.

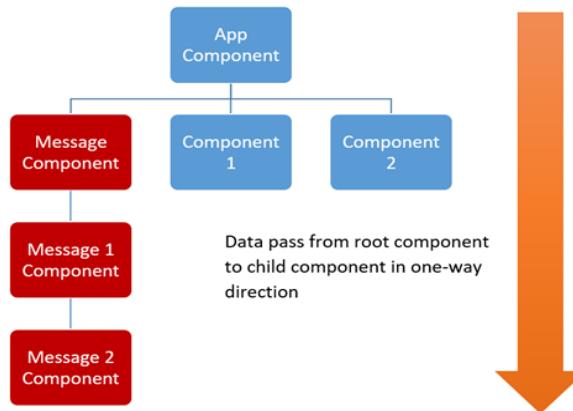


Next, let's go ahead and update the `firstname` property on the button click in the `AppComponent` class below:

```
changeName() {
  this.p.firstname = 'Rohan';
}
```

As soon as we changed the property of mutable object 'p', Angular fires the change detector to make sure that the DOM (or view) is in sync with the model (in this case, object p). For each property changes, Angular change detector will traverse the component tree and update the DOM.

Let's start with understanding the component tree. An Angular application can be seen as a component tree. It starts with a root component and then goes through to the child components. In Angular, data flows from top to bottom in the component tree.



Whenever the `@Input` type property will be changed, the Angular change detector will start from the root component and traverse all child components to update the DOM. Any changes in the primitive type's property will cause Angular change detection to detect the change and update the DOM.

In the above code snippet, you will find that on click of the button, the first name in the model will be changed. Then, change detection will be fired to traverse from root to bottom to update the view in `MessageComponent`.

There could be various reasons for Angular change detector to come into action and start traversing the component tree. They are:

1. Events fired such as button click, etc.
2. AJAX call or XHR requests.
3. Use of JavaScript timer functions such as `setTimeout`, `setInterval`.

Now, as you see, a single property change can cause change detector to traverse through the whole component tree. Traversing and change detection is a heavy process, which may cause performance degradation of application.

Imagine that there are thousands of components in the tree and mutation of any data property can cause change detector to traverse all thousand components to update the DOM. To avoid this, there could be a scenario when you may want to instruct Angular that when change detector should run for a component and its subtree, you can instruct a component's change detector to run only when object references changes instead of mutation of any property by choosing the **OnPushChangeDetection** strategy.

You may wish to instruct Angular to run change detection on components and their sub-tree only when new references are passed to them versus when data is simply mutated by setting change detection strategy to **OnPush**.

Let us go back to our example where we are passing an object to **MessageComponent**. In the last example, we just changed the **firstname** property and that causes change detector to run and to update the view of **MessageComponent**. However, now we want change detector to only run when the reference of the passed object is changed instead of just a property value. To do that, let us modify **MessageComponent** to use the **OnPush** ChangeDetection strategy. To do this set the **changeDetection** property of the **@Component** decorator to **ChangeDetectionStrategy.OnPush** as shown in listing below:

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

@Component({
  selector: 'app-message',
  template: `
    <h2>
      Hey {{person.firstname}} {{person.lastname}} !
    </h2>
    `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class MessageComponent {
  @Input() person;
}
```

At this point when you run the application, on the click event of the button in the **AppComponent** change detector will not run for **MessageComponent**, as only a property is being changed and reference is not changing. Since the change detection strategy is set to **OnPush**, now the change detector will only run when the reference of the **@Input** property is changed.

```
changeName() {
  this.p = {
    firstname: 'Rohan',
    lastname: 'Singh'
  };
  //OR
  this.p = {...this.p, firstName: "Rohan"}
}
```

In the above code snippet, we are changing the reference of the object instead of just mutating one property. Now when you run the application, you will find on the click of the button that the DOM is being updated with the new value.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

ChangeDetectorRef:

When using a change detection strategy of OnPush, other than making sure to pass new references every time something should change, we can also make use of the ChangeDetectorRef for complete control.

ChangeDetectorRef is responsible for performing change detection manually.

The ChangeDetectorRef is a base class for Angular views, providing change detection functionality. As described before, Angular has a change detection tree that has views to check for changes. Using the provided methods for this class, you can add or remove views from the tree, trigger change detection, and tell Angular what views need to be re-rendered.

```
abstract class ChangeDetectorRef {  
    abstract markForCheck(): void  
    abstract detach(): void  
    abstract detectChanges(): void  
    abstract checkNoChanges(): void  
    abstract reattach(): void  
}
```

Methods:

- **markForCheck()**: When a view uses the OnPush (checkOnce) change detection strategy, explicitly marks the view as changed so that it can be checked again.

Components are normally marked as dirty (in need of re-rendering) when inputs have changed or events have fired in the view. Call this method to ensure that a component is checked even if these triggers have not occurred.

- **detach()**: Detaches this view from the change-detection tree. A detached view is not checked until it is reattached. Use in combination with detectChanges() to implement local change detection checks.

Detached views are not checked during change detection runs until they are re-attached, even if they are marked as dirty.

- **detectChanges()**: Checks this view and its children. Use in combination with detach to implement local change detection checks.
- **checkNoChanges()**: Checks the change detector and its children, and throws if any changes are detected.
- **reattach()**: Re-attaches the previously detached view to the change detection tree. Views are attached to the tree by default.

Examples: The following examples demonstrate how to modify default change-detection behaviour to perform explicit detection when needed.



app.component.ts:

```

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

@Component({
  selector: 'app-root',
  template: `
    <p>{{ counter }}</p>
    <button (click)='Detach()'>Detach</button>&nbsp;
    <button (click)='Reattach()'>Reattach</button>&nbsp;
    <button (click)='Detect()'>Detect</button>
  `,
  changeDetection:ChangeDetectionStrategy.OnPush
})
export class AppComponent {
  counter = 0;

  constructor(private cdr: ChangeDetectorRef) {
    setInterval(() => {
      this.counter++;
      // require view to be updated
      this.cdr.markForCheck();
    }, 1000);
  }

  Detach(){
    this.cdr.detach();
  }

  Reattach(){
    this.cdr.reattach();
  }

  Detect(){
    this.cdr.detectChanges();
  }
}

```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

moduleId:

We know that the decorator functions of **@Component** take object and this object contains many properties. So, now we will learn about the **moduleId** property.

It is used to resolve relative paths for your stylesheets and templates. The module ID of the module that contains the component. The component must be able to resolve relative URLs for templates and styles.

SystemJS exposes the **__moduleName** variable within each module. In **CommonJS**, this can be set to **module.id**.

Definition:

moduleId?: string

moduleId parameter inside the **@Component** decorator takes a **string** value which is the module id of the module that contains the component.

CommonJS usage: `module.id`,

SystemJS usage: `__moduleName`

interpolation:

We know that the decorator functions of **@Component** take object and this object contains many properties. So, now we will learn about the **interpolation** property.

It Overrides the default encapsulation start and end delimiters (`{{` and `}}`)

interpolation?: [string, string]

Interpolation `{{...}}` refers to embedding expressions into marked up text. By default, interpolation uses as its delimiter the double curly braces, `{{` and `}}` and it can be override using **interpolation** metadata property of the **@Component** decorator with any other.

In the following snippet, `{{ title }}` is an example of interpolation.

src/app/app.component.html:

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

The text between the braces is often the name of a component property. Angular replaces that name with the string value of the corresponding component property.

src/app/app.component.html:

```
<p>{{message}}</p>
<div></div>
```

In the example above, Angular evaluates the **message** and **itemImageUrl** properties and fills in the blanks, first displaying some message text and then an image.

So, if you want to override the default interpolation expression, use **interpolation** metadata property of the **@Component** decorator in component file and then bind any property of component class in html template (view) with the new expression as following:

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

src/app/app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls:['./app.component.css'],
  interpolation:["[[", "]]"] -> It overrides with new expression i.e. [[ and ]]
})
export class AppComponent {
  title = "Angular Application"
}
```

src/app/app.component.html:

```
<h1>
  Welcome to [[ title ]]!
</h1>
```

encapsulation:

We know that the decorator functions of `@Component` take object and this object contains many properties. So, now we will learn about the **encapsulation** property.

Angular is inspired from Web Components, a core feature of which is the **Shadow DOM**. The Shadow DOM lets us include styles into Web Components without letting them leak outside the component's scope. The Shadow DOM defines how to use encapsulated style and markup in web components.

Angular also provides this feature for Components and we can control it with the **encapsulation** property.

“One of the fundamental concept in object oriented programming (OOP) is *Encapsulation*. It defines the idea that all the data and methods that operate on that data are kept private in a single unit (or class). It is like hiding the implementation detail from the outside world. The consumer of encapsulated object know that it works, but do not know how it works.”

Syntax of encapsulation property:

encapsulation: **ViewEncapsulation**

ViewEncapsulation:

ViewEncapsulation in Angular defines how the styles defined in the template affects the other parts of the application. While rendering the view, angular uses following mainly three strategies such as **ViewEncapsulation.Emulated**, **ViewEncapsulation.ShadowDOM** and **ViewEncapsulation.None**.

Angular allows us to specify the component specific styles. This is done by specifying either inline style **styles**: or using the external style sheet **styleUrls**: in the `@Component` decorator.

Example of External Style sheet:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls:['./app.component.css']
})
```

Example of Inline style:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: ['p { color:blue}']
})
```

The CSS Styles has global scope. CSS rules applies to the entire document. You cannot apply rules to the part of the document. Hence, we use **class**, **id** & **CSS specificity** rules to ensure that the styles do not interfere with each other

In case of Angular apps the components co exists with the other components. Hence it become very important to ensure that the CSS Styles specified in one component does not override the rules in another component.

Angular does this by using the **View Encapsulation strategies**

The View Encapsulation in Angular is a strategy which determines how angular hides (encapsulates) the styles defined in the component from bleeding over to the other parts of the application.

The following three strategies provided by the Angular to determine how styles are applied.

- ViewEncapsulation.None
- ViewEncapsulation.Emulated
- ViewEncapsulation.ShadowDOM

The **ViewEncapsulation.Native** is deprecated since Angular version 6.0.8, and is replaced by **ViewEncapsulation.ShadowDom**

Adding View Encapsulation to components:

The Encapsulation methods are added using the **encapsulation** metadata of the **@Component** decorator as shown below:

```
@Component({
  template: `<p>Using Emulator</p>`,
  styles: ['p { color:red }'],
  encapsulation: ViewEncapsulation.Emulated //This is default
  //encapsulation: ViewEncapsulation.None
  //encapsulation: ViewEncapsulation.ShadowDOM
})
```

ViewEncapsulation.Emulated is the default encapsulation method.

Let's try to understand it using an example. We have created a component, as shown below:

app.component.ts:

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



app.component.html:

```
<h1>{{ title }}</h1>
<p>I am a paragraph in green</p>
```

Open the src/styles.css and add the following CSS:

```
p {color: green;}
```

Run the app and you should able to see the paragraph in green.

Open the chrome developer tools and check the elements section. The CSS rules are inserted in the head section of the page.

View Encapsulation in Angular

I am a paragraph in green

```

<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>ViewEncapsulation</title>
    <base href="/">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
    <style type="text/css">
      p {color: green;}
    <!--
      sourceMappingURL=data:application/json;base64,eyJ2ZXJzaW9uIjoxLCJzb3VyY2VzIjpBInNyYy9zdHlsZXMuYSxQUFDIiwIZmlsZSI6InNyYy9zdHlsZXMuY3NzIiwic291cmNlc0NvbRbnQiOlsicCAge2NvbG9yOiBncmVlbjt9Il
    -->
  </head>
  ... <body> == $0
    <app-root _ngcontent-c0 ng-version="7.1.4">
      <h1 _ngcontent-c0>View Encapsulation in Angular</h1>
      <p _ngcontent-c0>I am a paragraph in green</p>
    </app-root>
    <script type="text/javascript" src="runtime.js"></script>
  
```

ViewEncapsulation.None

The ViewEncapsulation.None is used, when we do not want any encapsulation. When you use this, the styles defined in one component affects the elements of the other components.

Now, let us look at ViewEncapsulation.None does.

Create a new component ViewNoneComponent and add the following code.

```
import { Component, ViewEncapsulation } from '@angular/core';
@Component({
  selector: 'app-none',
  template: `<p>I am not encapsulated and in blue
            (ViewEncapsulation.None) </p>`,
  styles: ['p { color:blue }'],
  encapsulation: ViewEncapsulation.None
})
export class ViewNoneComponent {
```

We have added encapsulation: **ViewEncapsulation.None**.

We have also defined the inline style **p { color:blue }**

Do not forget to import & declare the component in **AppModule**.

You also need to add the **<app-none></app-none>** selector in **app.component.html**

```
<h1>{{title}}</h1>
<p>I am a paragraph in green</p>
```

```
<app-none></app-none>
```

Run the code and as expected both the paragraphs *turn blue*.

That is because, the global scope of CSS styles. The style defined in the **ViewNoneComponent** is injected to the global style and overrides any previously defined style. The style **p {color: blue;}** overrides the style **p {color: green;}** defined in the **styles.css**.

In the browser, when you examine source code, you will find the p style has been declared in the head section of the DOM.

Therefore, in **ViewEncapsulation.None**, the style gets moved to the DOM's head section and is not scoped to the component. There is no Shadow DOM for the component and the component style can affect all nodes in the DOM.

View Encapsulation in Angular

I am a paragraph in green

I am not encapsulated and in blue (ViewEncapsulation.None)

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>View Encapsulation</title>
    <base href="/">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
    <style type="text/css">
      p {color: green;}
      /*# sourceMappingURL=data:application/json;base64,eyJ2ZXJzaW9uIjozLCJzb3VyY2VzIjpBInNyYy9zdHlsZXMuY3NzIl0sImShbHVY5SxDQUFDIiwImlzZSI6InNyYy9zdHlsZXMuY3NzIiwic291cmNlc0NvbnRlbnQiOlsicCAge2NvbG9yOiBncmVlbjt9I119 */
    </style>
    <style>...</style>
    <style>p { color:blue}</style>
  </head>
  <body>
    <app-root _ngcontent-c0 ng-version="7.1.4">
      <h1 _ngcontent-c0>View Encapsulation in Angular</h1>
      <p _ngcontent-c0>I am a paragraph in green</p>
      <app-none _ngcontent-c0>
        <p>I am not encapsulated and in blue (ViewEncapsulation.None) </p>
      </app-none>
    </app-root>
  </body>
</html>
```

The important points are:

The styles defined in the component affect the other components (Style is not scoped to the component.)

The global styles affect the element styles in the component



ViewEncapsulation.Emulated:

In an HTML page, we can easily add an id or a class to the element to increase the specificity of the CSS rules so that the CSS rules do not interfere with each other.

The ViewEncapsulation.Emulated strategy in angular add the unique HTML attributes to the component CSS styles and to the markup to achieve the encapsulation. This is not true encapsulation. The Angular emulates the encapsulation, Hence the name Emulated.

If you do not specify encapsulations in components, the angular uses the ViewEncapsulation.Emulated strategy.

Create a new component in Angular app and name it as ViewEmulatedComponent as shown below:

```
import { Component, ViewEncapsulation } from '@angular/core';
@Component({
  selector: 'app-emulated',
  template: `<p>I am now encapsulated using ViewEncapsulated.Emulated</p>`,
  styles: ['p { color:red }'],
  encapsulation: ViewEncapsulation.Emulated
})
export class ViewEmulatedComponent { }
```

Update app.component.html

```
<h1>{{title}}</h1>
<p>I am a paragraph in green</p>
<app-none></app-none>
<app-emulated></app-emulated>
```

Now, you can see that the style does not spill out to other components, when you use emulated mode i.e. because angular add `_ngcontent-dkg-c2` attributes to the emulated components and makes necessary changes in the generated styles.

You can see this by opening the chrome developer console

`_ngcontent-dkg-c2` attribute is added in the style and to the p element, making the style local to the particular component only.

```
<style>p[_ngcontent-dkg-c2] { color:red}</style>
```

```
<app-emulated _ngcontent-dkg-c0="" _nghost-dkg-c2="">
<p _ngcontent-dkg-c2="">I am now encapsulated using ViewEncapsulated.Emulated</p>
</app-emulated>
```

View Encapsulation in Angular

I am a paragraph in green

I am not encapsulated and in blue (ViewEncapsulation.None)

I am now encapsulated using ViewEncapsulation.Emulated

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>ViewEncapsulation</title>
    <base href="/">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
  </head>
  <body>
    <app-root _ngcontent-c0 ng-version="7.1.4">
      <h1 _ngcontent-c0>View Encapsulation in Angular</h1>
      <p _ngcontent-c0>I am a paragraph in green</p>
      <app-none _ngcontent-c0></app-none>
      <app-emulated _ngcontent-c0 _nghost-c2>
        <p _ngcontent-c2>I am now encapsulated using ViewEncapsulation.Emulated</p>
      </app-emulated>
    </app-root>
  </body>
</html>
```



The important points are:

- Style will be scoped to the component.
- This is the default value for encapsulation.
- The Angular adds the attributes to the styles and mark up
- Angular will not create a Shadow DOM for the component.

ViewEncapsulation.ShadowDOM:

The Shadow DOM is a scoped sub-tree of the DOM. It is attached to an element (called shadow host) of the DOM tree. The shadow DOM do not appear as child node of the shadow host, when you traverse the main DOM.

The browser keeps the shadow DOM separate from the main DOM. The rendering of the Shadow DOM and the main DOM happens separately. The browser flattens them together before displaying it to the user. The feature, state & style of the Shadow DOM stays private and not affected by the main DOM. Hence it achieves the true encapsulation.

"The Shadow DOM is part of the Web Components standard. Not all browsers support shadow DOM. Use Google Chrome for the following examples"

Introduction to Shadow DOM:

We will discuss what shadow DOM is and how to use it with a simple example.

The CSS Styles are global in scope. The styles affect the entire web site, irrespective of where they are placed in the page. The global scoped CSS rules has few advantages. For Example you can set the font of the entire web site at one place. But it also makes it easier to break the site. The CSS rules might target unwanted elements or clash with other CSS selectors. Similarly the JavaScript may also accidentally modify unintended parts of the app.

The Shadow DOM solves the above problem, by creating an encapsulated DOM tree within the parent DOM tree.

What is Shadow DOM?

The Shadow DOM is a scoped sub-tree of DOM. It is attached to an element (shadow host) of the DOM tree, but do not appear as child nodes of that element.

How Shadow DOM Work?

To understand how Shadow DOM Work, let us build a simple example

Consider the following HTML. We have an h1 and p elements. The p element style is defined under the head section.

```
<html>
  <head>
    <style>
      p {color: orange;}
    </style>
    <title>Hello Shadow DOM </title>;
  </head>
  <body>
    <h1>What is Shadow DOM</h1>
    <p>The Shadow DOM is a DOM inside a DOM</p>
  </body>
</html>
```

The Browser creates a Document Object Model or DOM of the page. When it loads the web page as shown below. As expected the p element coloured orange.

What is Shadow DOM

The Shadow DOM is a DOM inside a DOM

```
<html>
... <head> == $0
  <style>
    p {color: orange;}
  </style>
  <title>Hello Shadow DOM </title>
</head>
<body>
  <h1>What is Shadow DOM</h1>
  <p>The Shadow DOM is a DOM inside a DOM</p>
</body>
</html>
```

Now, let us add the following HTML content.

```
<div>
  <style>p {color: blue;}</style>
  <h1>Shadow DOM Example</h1>
  <p>This is not Shadow DOM</p>
</div>
```

The style {color: blue;} is applied to the element p inside the div element, but it also affects the p element outside the div element i.e. because styles are global in scope. It affects the entire document irrespective of where it is appears in document.

What is Shadow DOM

The Shadow DOM is a DOM inside a DOM

Shadow DOM Example

This is not Shadow DOM

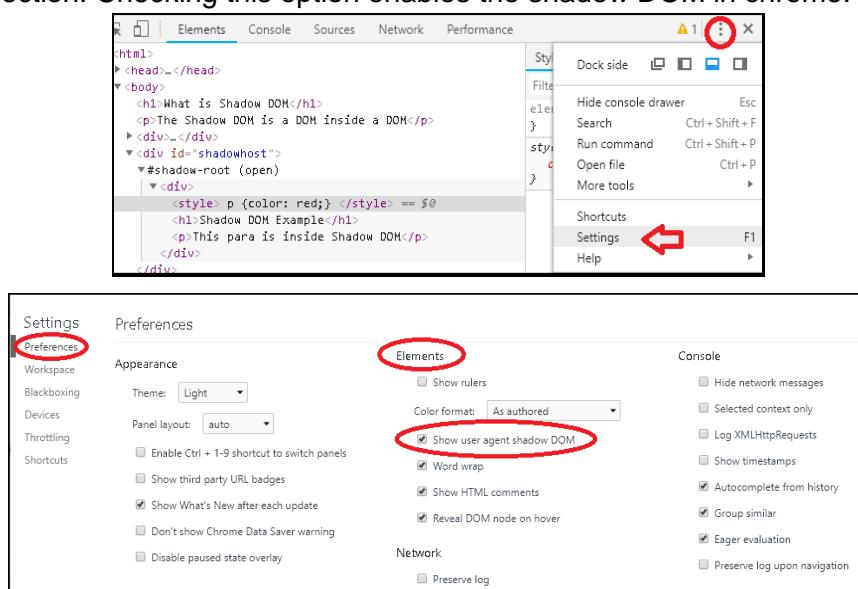
```
<html>
... <head>
  <style>
    p {color: orange;}
  </style>
  <title>Hello Shadow DOM </title>
</head>
<body>
  <h1>What is Shadow DOM</h1>
  <p>The Shadow DOM is a DOM inside a DOM</p>
</body>
<div>
  <style>p {color: blue;}</style>
  <h1>Shadow DOM Example</h1>
  <p>This is not Shadow DOM</p>
</div>
</html>
```

Shadow DOM Example:

Now, let us add shadow DOM to the above HTML. But before that let us enable shadow DOM in chrome. Internet explorer & edge not yet support shadow DOM.

Enabling Shadow DOM in Chrome:

To enable shadow DOM chrome browser, press Ctrl+Shift+I to open the chrome developer tools. On the top right corner, you will find 3 vertical dots as shown in the image below. Click on settings and select the preference tab as shown in the next image. Here you will find Show user agent shadow DOM checkbox under the Elements section. Checking this option enables the shadow DOM in chrome.



Now let us add the content using the Shadow DOM.

```
<html>
  <head>
    <style>
      p {color: orange;}
    </style>
    <title>Hello Shadow DOM</title>
  </head>
  <body>
    <h1>What is Shadow DOM?</h1>
    <p>The Shadow DOM is a DOM inside a DOM</p>
    <div>
      <style>p {color: blue;}</style>
      <h1>Shadow DOM Example</h1>
      <p>This is not Shadow DOM</p>
    </div>
    <div id="shadowhost"></div>
  </body>
</html>
<script>
  var host = document.getElementById('shadowhost');
  var shadowRoot = host.attachShadow({mode: 'open'});
  var div = document.createElement('div');
  div.innerHTML='<style> p {color: red;} </style> <h1>Shadow DOM Example</h1> <p>This para is inside Shadow DOM</p>';
  shadowRoot.appendChild(div);
</script>
```

First, we added a **div** element with the id **shadowhost**. This is where our shadow DOM will be attached. This element is called **Shadow host**.

```
<div id="shadowhost"></div>
```

Next, in the JavaScript code, get the reference to the **shadowhost** element

```
var host = document.getElementById('shadowhost');
```

Then, we attach the Shadow DOM to the **shadowhost** element by invoking the **attachShadow** method. The **attachShadow** method returns the **Shadow Root**

```
var shadowRoot=host.attachShadow({mode: 'open'});
```

Next, create a **div** element and add the content

```
var div = document.createElement('div');
div.innerHTML='<style>p{color:red;}</style><h1>Shadow DOM Example</h1><p>This para is inside Shadow DOM</p>'
```

Finally, attach the **div** element to the **shadowRoot**

```
shadowRoot.appendChild(div);
```

What is Shadow DOM

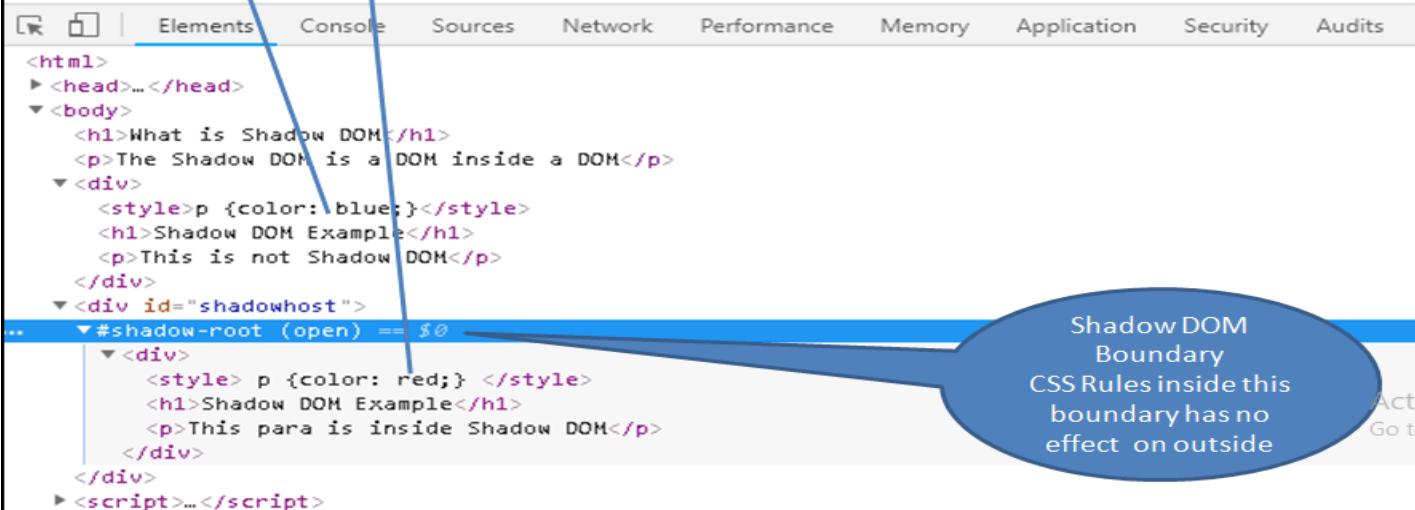
The Shadow DOM is a DOM inside a DOM

Shadow DOM Example

This is not Shadow DOM

Shadow DOM Example

This para is inside Shadow DOM



```
<html>
  <head>...</head>
  <body>
    <h1>What is Shadow DOM</h1>
    <p>The Shadow DOM is a DOM inside a DOM</p>
    <div>
      <style>p {color: blue;}</style>
      <h1>Shadow DOM Example</h1>
      <p>This is not Shadow DOM</p>
    </div>
    <div id="shadowhost">
      <#shadow-root (open) == $0>
        <div>
          <style> p {color: red;} </style>
          <h1>Shadow DOM Example</h1>
          <p>This para is inside Shadow DOM</p>
        </div>
    </div>
    <script>...</script>
  </body>
</html>
```

As we have seen from the above example, any styles applied inside the shadow DOM is scoped to the shadow root and does not spill out to the other areas of the DOM. Similarly, styles applied on the parent also does not affect the elements in the Shadow DOM.

Terminology

Shadow Host

The host or shadow host is an element to which the shadow DOM gets attached. It is part of the DOM tree and not part of the shadow DOM. The host can access its shadow property using the **shadowRoot** method. The host can have shadow DOM attached to it and still have child nodes in the DOM tree.

Shadow Root

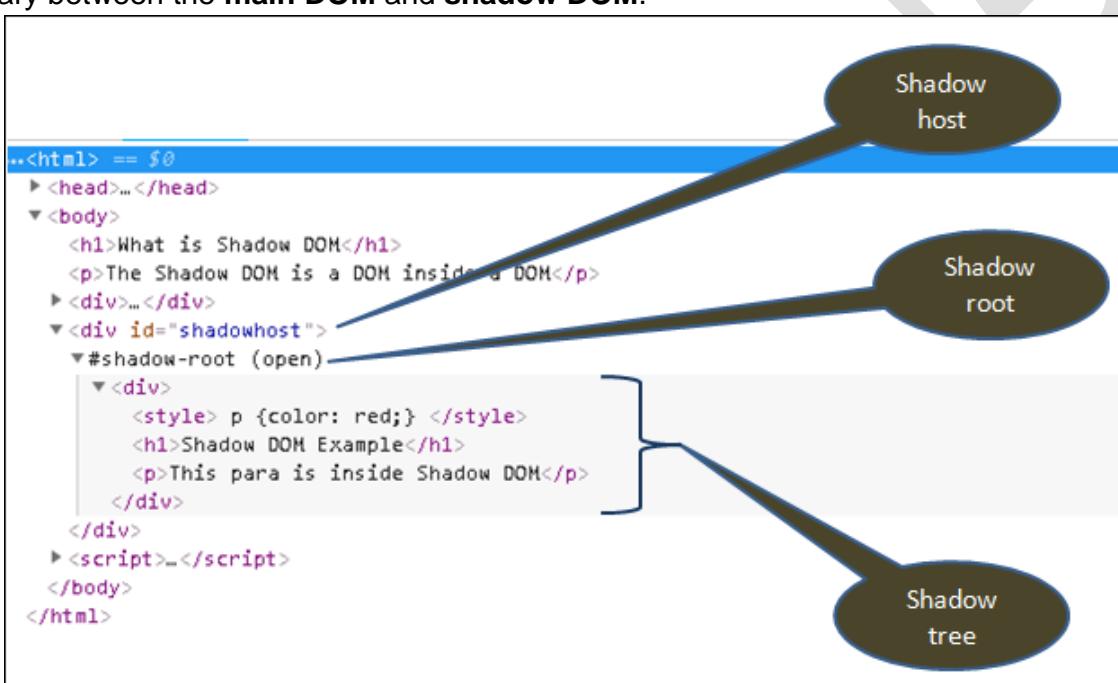
The root node of the shadow DOM. The shadow root gets attached to the shadow host.

Shadow Tree

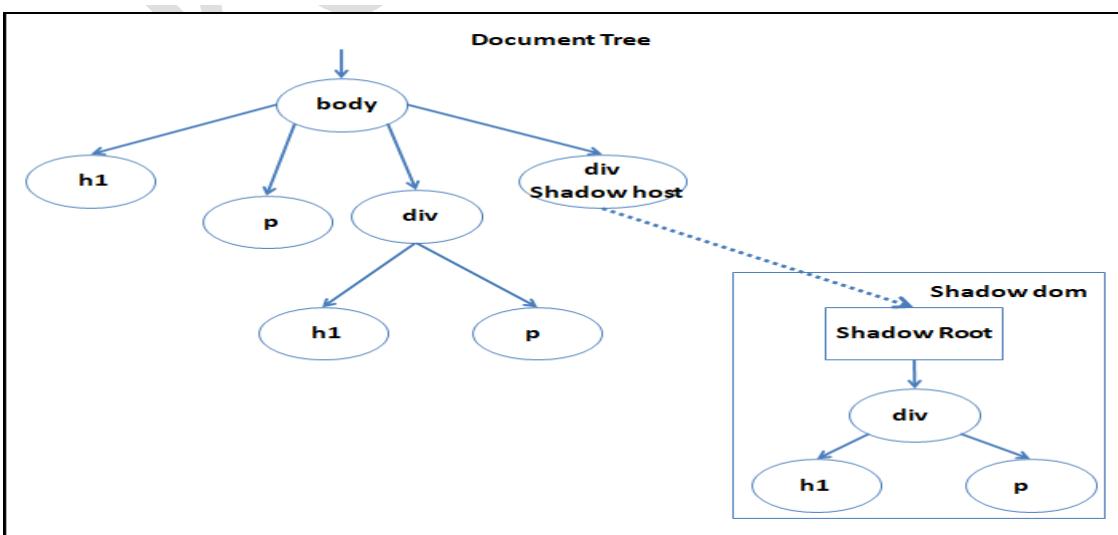
All the elements that go into the **Shadow Root**, which is scoped from outside world, is called **Shadow Tree**

Shadow Boundary

The boundary between the **main DOM** and **shadow DOM**.



The following image shows how the DOM tree of the above example app looks like. The **shadow host** is part of **main DOM** tree. The **shadow DOM** is attached to the **main DOM** at **Shadow host** element.



To create shadow DOM in angular, all we need to do is to add the `ViewEncapsulation.ShadowDom` as the encapsulation strategy.

Create a new component `ViewShadowdomComponent` and add the following code

shadowdom.component.ts:

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

@Component({
  selector: 'app-shadowdom',
  template: `<p>I am encapsulated inside a Shadow DOM using ViewEncapsulation.ShadowDom</p>`,
  styles: ['p { color:brown}'],
  encapsulation: ViewEncapsulation.ShadowDom
})
export class ViewShadowdomComponent {
```

The component renders as follows

View Encapsulation in Angular

I am a paragraph in green
 I am not encapsulated and in blue (ViewEncapsulation.None)
 I am now encapsulated using ViewEncapsulation.Emulated
 I am encapsulated inside a Shadow DOM ViewEncapsulation.ShadowDom

Elements Console Sources Network Performance Memory Application

```
<!doctype html>
<html lang="en">
  <head>...
  <body>
    <app-root _ngcontent-c0 ng-version="7.1.4">
      <h1 _ngcontent-c0>View Encapsulation in Angular</h1>
      <p _ngcontent-c0>I am a paragraph in green</p>
      <app-none _ngcontent-c0>I am not encapsulated and in blue (ViewEncapsulation.None)</app-none>
      <app-emulated _ngcontent-c0 _ngcontent-c2></app-emulated>
      <app-shadowdom _ngcontent-c0>
        <#shadow-root (open)>
          <style>...</style>
          <style>p { color:blue}</style>
          <style>p[_ngcontent-c2] { color:red}</style>
          <style>p { color:brown}</style>
          <p>I am encapsulated inside a Shadow DOM ViewEncapsulation.ShadowDom</p>
        </app-shadowdom>
      </app-root>
    </body>
  </html>
```

app-shadowdom becomes Shadow host, to which Shadow DOM is attached

Style copied from the ShadowDomComponent

Styles are also copied from parent and sibling component

The Shadow DOM starts at #shadow-root

It is rendered independently from the rest of the document

The angular renders the component inside the `#shadow root` element. The styles from the component along with the styles from the parent and other components are also injected inside the `shadow root`.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



Shadow DOM terminology

The **app-shadowdom** is the CSS selector in the **ViewShadowdomComponent**. We used it in our **app-component.html**. The Angular renders component as **shadow DOM** and attaches it to the **app-shadowdom** selector. Hence, we call the element as **Shadow host**

The **Shadow DOM** starts from **#shadow-root** element. Hence, we call this element as **shadow root**. The Angular injects the component into the **shadow root**.

The **Shadow boundary** starts from the **#shadow-root**. The browser encapsulates everything inside this element including the node **#shadow-root**.

The **shadow DOM** achieves the true encapsulation. It truly isolates the component from the styles from the other parts of the app.

The styles from the parent component & sibling components are still injected into the shadow DOM but that is an angular feature. The angular wants the component to share the parent & sibling styles. Without this the component may look out of place with the other component.

The important points are:

The shadow DOM achieves the true encapsulation.

The parent and sibling styles still affect the component but that is Angular's implementation of shadow DOM

The shadow DOM not yet supported in all the browsers.

entryComponents:

A set of components that should be compiled along with this component. For each component listed here, Angular creates a **ComponentFactory** and stores it in the **ComponentFactoryResolver**.

entryComponents: `Array<Type<any> | any[]>`

animations:

Defining animations and attaching them to the HTML template. Animations are defined in the metadata of the component that controls the HTML element to be animated. Put the code that defines your animations under the **animations:** property within the `@Component()` decorator.

One or more animation **trigger()** calls, containing **state()** and **transition()** definitions.

animations: `any[]`

selector:

The CSS selector that identifies this directive in a template and triggers instantiation of the directive.

selector: `string`

Declare as one of the following:

- **element-name:** Select by element name.
- **.class:** Select by class name.
- **[attribute]:** Select by attribute name.
- **[attribute=value]:** Select by attribute name and value.
- **:not(sub_selector):** Select only if the element does not match the sub_selector.
- **selector1, selector2:** Select if either selector1 or selector2 matches.

1. Element Name Selector:

Using the element name selector you can target specific HTML element tags, this can be either a standard Html element or an Angular component. The example below is targeting all element tags with the name **app-root**.

```
@Component({
  selector: 'app-root'
})
```

2. Attribute Selector:

Using the attribute selector you can target specific HTML elements that contain the specific attribute(s). Attributes selectors are contained within square brackets [].

The example below is targeting all elements that have an attribute with the name **RSN**.

```
@Component({
  selector: '[RSN]'
})
```

Attribute selectors can also be further restricted to the value the attribute is being set to. The example below is targeting all attributes that have an attribute with the name **RSN** with a value of **Angular**.

```
@Component({
  selector: '[RSN=Angular]'
})
```

3. Class Selector:

Using the class selector you can target specific HTML elements with the specified CSS class names. The CSS selector is prefixed with a dot, which is the same as CSS syntax.

The example below is targeting all elements that have a CSS class names containing **rsn**.

```
@Component({
  selector: '.rsn'
})
```

One warning with the class selector is that you cannot target class names that are dynamically inserted into the DOM using an expression binding either via a **class binding** or **NgClass attribute directive**.

Chaining Selectors (and)

Each of the selectors (element, class and attribute) can be chained together to narrow the query for the relevant HTML elements like “selector and selector”. There are no tokens that separate each of the chained selectors, the selectors are separated using selector token itself either using square brackets for attribute selectors or a dot for class selectors.

```
@Component({
  selector: 'rsn[tech]'
})
```

The example above will select all HTML elements that have an element tag named **rsn** and an attribute named **tech**. Notice there is no space between the each of the individual selectors, only the token the square brackets is used to delineated between the selectors.

The only limitation with the order of selectors is a class selector cannot be succeeded by an element selector, as there is no symbol to separate the two selectors. Also an element name selector can only be specified once otherwise the directive will not find any elements.

```
@Component({
  selector: 'rsn.angular'
})
```

The above example will select all HTML elements with an element name of **rsn** and a CSS class name containing **angular**. The order cannot be reversed otherwise the selector would be querying for elements with a class name of **angularrsn**, which is not desired.

Chaining selectors can only include a single element name selector with zero or more of either class name or attribute selectors.

```
@Component({
  selector: 'rsn.angular[tech]'
})
```

The above example will select all HTML elements with an element name of **rsn**, a CSS class name containing **angular** and an attribute named **tech**.

Exclusion Selectors:

Querying for specific HTML elements sometimes there is a need to exclude specific elements that contain either a class name or attribute.

To exclude specific HTML elements use the keyword **:not(...)** with the specific selector inserted between the round brackets.

The example below will select all HTML elements with an element name of **rsn** but exclude those with an attribute named **tech**.

```
@Component({
  selector: 'rsn:not([tech])'
})
```

Exclusion selectors can be chained together to further refine the query.

```
@Component({
  selector: 'rsn:not([tech]):not(.angular)'
})
```

The example above will select all HTML elements with an element name of **rsn** but exclude those with an attribute name **tech** and a CSS class containing **angular**.

Combining Selectors (or):

All the selectors' combinations above can be combined together in groups using a comma as the separating token to create an OR like query. There is no limit on the number of selectors that can be combined together for targeting of the directive. Where there are multiple possible matches found, only a single instance of the directive will be instantiated for the HTML DOM element.

```
@Component({
  selector: 'rsn,[tech]'
})
```

The example above will select all HTML elements with an element name of **rsn** or with an attribute named **tech**. If a HTML element contains both matching criteria the directive will only be attached once.



Inputs:

Enumerates the set of data-bound input properties for a directive

inputs: string[]

Inputs property is used within one component (child component) to receive a value from another component (parent component). This is a one-way communication from parent to child. A component can receive a value from another component through component inputs property. Now we will see how to use the inputs property.

It is the topic of the Component Interaction in angular. As we know, the angular application is based on small components, so passing data from the parent component to the child component is a bit complicated, in this scenario, the inputs property is useful. The Inputs specify which properties can be set in a component by a parent.

An alternative syntax is available to define inputs property in a component. Angular also allows you to use a @Input decorator to get the same output instead of using "inputs" property of the object passed to the @Component decorator.

- @Input is a decorator to mark an input property.
- @Input is used to define an input property to bind the component's input property.
- The input property of the component must be annotated with the input decorator to act as an input property.

Note that the style guide recommended by the angular team to use @Input:

Do Use the @Input() and @Output() class decorators instead of the @Directive and @Component metadata inputs and outputs properties.

Consider entering @Input() or @Output() on the same line as the decorated property.

It is true that @Input allows an easy definition of type, scope and default values.

An advantage of using inputs is that class users only need to look at the configuration object passed to the @Component decorator to find the input properties.

An advantages of using @Input is that we can define the type and whether it is private or public as shown below

```
@Input() public pData: string;
```

So in this, we will see the data of the parent component in a child component. For this, we need to create two components. Go to the terminal and create two components by writing the following command. Remember that we are using Angular CLI to generate the new component.

Step 1: Create parent and child components

Go to the terminal window and enter the following command.

```
ng g c parent
```

```
ng g c child
```

So, it will create an individual folder for both components. Type the following command to start angular development server.

```
ng serve --open
```

It will open the browser in the port: 4200.

At the moment, only the **app.component.ts** component is represented in the browser. If we want to render our parent component, we need to include it in an **app.component.html** file as HTML tags.

app.component.html:

```
<div style="text-align: center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  <app-parent></app-parent>
</div>
```

Now, if you see in the browser, you can see the representations of the parent component. "parent works!"

Step 2: Define HTML for parent component

Write the following code in the **parent.component.html** file

```
<h3>Parent Component</h3>

<label>Parent Component: </label>
<input type="text"/>
```

First, we pass data from the parent component to the child component. Here is the scenario, when the user enters something in the text box, we can see its value in the child component.

Step 3: Define HTML for child component

Write the following code in the **child.component.html** file.

```
<h3>Child Component</h3>
```

As we know, this is the child component, so you need to include the **<app-child>** tag in the parent component. So our main HTML component has this aspect.

parent.component.html:

```
<h3>Parent Component</h3>

<label>Parent Component</label>
<input type="text"/>
<app-child></app-child>
```

So, now our application looks like this.



Welcome to DemoAngularApp1!

Parent Component

Parent Component:

Child Component

Step 4: Use Input to display parent component value

Create a reference to input text of the parent component. So edit the following lines in the **parent.component.html** file.

```
<h3>Parent Component</h3>
<label>Parent Component: </label>
<input
  type="text"
  #pcomponent
  (keyup)="0"
/>
<app-child [PData]="pcomponent.value"></app-child>
```

Here, we first defined the reference for the input tag and then set the event listener. When a user types something in the text box, it will pass the value as a property to the child component.

The child component is ready to receive the property via an input property. So this is the first case of using inputs in angular.

The **child.component.ts** file looks like this.

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css'],
  inputs:[['PData']]
})
export class ChildComponent implements OnInit {
  PData: any;
  constructor() { }
  ngOnInit() {
  }
}
```

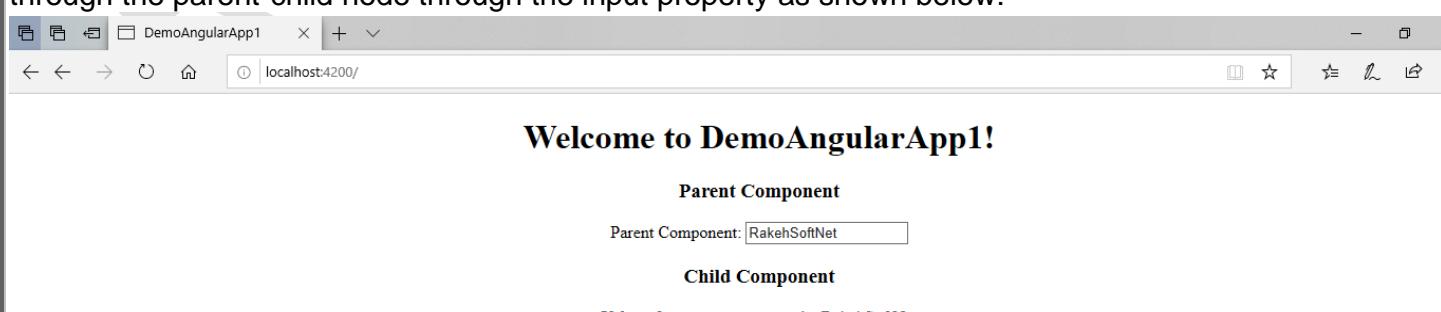
You can see that the property of this component is **PData**, which is the same property that we wrote in the **parent.component.html** file.

Finally, our **child.component.html** file looks like this. We just need to add interpolation to show the parent data in the child component.

child.component.html:

```
<h3>Child Component</h3>
<p>
  <b>Value of parent component is: </b>{{ PData }}
</p>
```

Now, if you type in the parent text box, its value is printed on the child component. Everything is done through the parent-child node through the input property as shown below:



Welcome to DemoAngularApp1!

Parent Component

Parent Component:

Child Component

Value of parent component is: RakehSoftNet



outputs:

Enumerates the set of event-bound output properties.

outputs: string[]

When an output property emits an event, an event handler attached to that event in the template is invoked.

The outputs property defines a set of **directiveProperty** to **bindingProperty** configuration:

- directiveProperty specifies the component property that emits events.
- bindingProperty specifies the DOM property the event handler is attached to.

Output property is used to send data from one component (child component) to calling component (parent component). This is a one-way communication from child to parent. This property name becomes a custom event name for the calling component. The output property can also create an alias with the property name as Output (alias) and now this alias name will be used in the custom event link in the call component. Now we will see how to use the outputs property.

It is the topic of the **Component Interaction** in angular. As we know, the angular application is based on small components, so it is very difficult to pass data from a child component to a parent component, in this scenario the outputs property is very useful. The angular components have a better way of notifying the parent components that something has changed through events. The "outputs" identify events that a component can fire to send information from the hierarchy to its parent from its child component.

An alternative syntax is available to define outputs properties in a component. Angular also allows you to use a @Output decorator to get the same result.

- @Output is a decorator to mark an output property.
- @Output is used to define an output property to bind the component's output property.
- The component output property must be annotated with the @Output decorator to act as an output property.

Note that the style guide recommended by the angular team to use @Output:

Do Use the @Input() and @Output() class decorators instead of the @Directive and @Component metadata inputs and outputs properties:

An advantage of using outputs is that class users only need to look at the configuration object passed to the @Component decorator to find the output properties.

An advantages of using @Output is that we can define the type and whether it is private or public as shown below

```
@Output() public pData: string;
```

So in this example, we will display child component's data into the parent component. For this, we need to create two components. Go to the terminal and create two components by writing the following command. Remember that we are using Angular CLI to generate the new component.

Step 1: Create parent and child components

Go to the terminal window and enter the following command.

```
ng g c parent
```

```
ng g c child
```

So, it will create an individual folder for both components. Type the following command to start angular development server.

```
ng serve --open
```

It will open the browser in the port: 4200.

At the moment, only the app.component.ts component is represented in the browser. If we want to render our parent component, we need to include it in an app.component.html file as HTML tags.

app.component.html:

```
<div style="text-align: center">
  <h1>Welcome to {{title}}</h1>
  <app-parent></app-parent>
</div>
```

Now, if you see in the browser, you can see the representations of the parent component. "Parent work!"

Step 2: Define HTML for parent component

Write the following code in the parent.component.html file.

parent.component.html:

```
<h3>Parent Component</h3>
```

First, we pass data from the child component to the parent component. Here is the scenario, when the user enters something in the text box, we can see its value in the parent component.

Step 3: Define HTML for child component

Write the following code in the child.component.html file.

child.component.html:

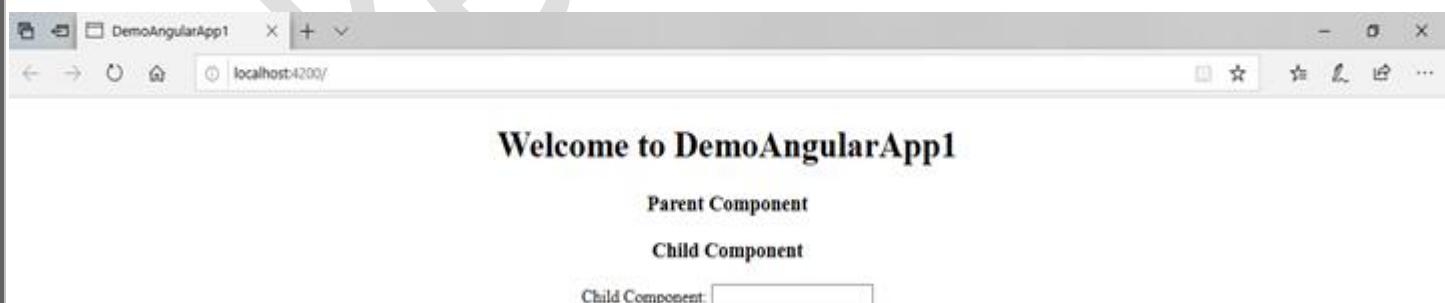
```
<h3>Child Component</h3>
<label>Child Component: </label>
<input type="text"/>
```

As we know, this is the child component, so you need to include the <app-child> tag in the parent component.

parent.component.html:

```
<h3>Parent Component</h3>
<app-child></app-child>
```

So, now our application looks like this.



Step 4: Pass value from child to parent component

Passing data from the child component to the parent component is a bit complicated. In this scenario, the child component has no reference to the parent component. So, in this case, we need to issue an event from the child component, and the parent component will listen to it and receive the data through the event and display it.

First, create a reference to the input in the child component and attach an event listener to it.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

child.component.html:

```
<h3>Child Component</h3>
<label>Child Component: </label>
<input type="text" #ccomponent (keyup)="onChange(ccomponent.value)"/>
```

Write the onChange function in the child.component.ts file.

child.component.ts:

```
import { Component, OnInit, EventEmitter } from '@angular/core';
@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css'],
  outputs:[ 'childEvent' ]
})
export class ChildComponent implements OnInit {
  childEvent = new EventEmitter();
  constructor() { }
  onChange(value)
  {
    this.childEvent.emit(value);
  }
  ngOnInit() {
  }
}
```

When the user types something in the text box of the child component, it begins to emitting the value of the child component. We just have to listen to that event emitter and show the value passed in the parent component.

Use an event binding in the **parent.component.html** file and listen to the event emitter.

```
<app-child (childEvent)="CData=$event"></app-child>
```

We need to define **CData** into the **parent.component.ts** file.

```
public cData:any;
```

Finally, **parent.component.ts** file look like this.

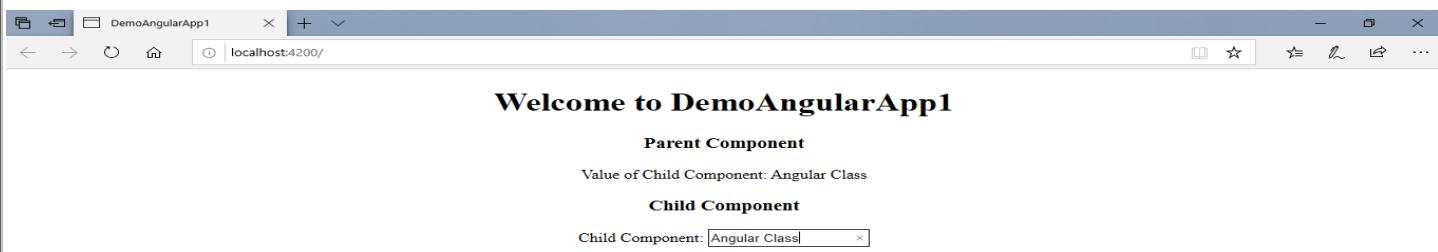
```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-parent',
  templateUrl: './parent.component.html',
  styleUrls: ['./parent.component.css']
})
export class ParentComponent implements OnInit {
  public cData:any;
  constructor() { }
  ngOnInit() {
  }
}
```

Finally, through interpolation, we can show its value in the `parent.component.html` file.

parent.component.html:

```
<h3>Parent Component</h3>
<p>Value of Child Component: {{ CData }}</p>
<app-child (childEvent)="CData=$event"></app-child>
```

Now, if you type in the child component text box, its value is printed on the parent component.



providers:

Configures the injector of this directive or component with a token that maps to a provider of a dependency.

providers: `Provider[]`

exportAs:

Defines the name that can be used in the template to assign this directive to a variable.

exportAs: `string`

queries:

Configures the queries that will be injected into the directive or component.

queries: {

[key: string]: any;

}

host:

Maps class properties to host element bindings for properties, attributes, and events, using a set of key-value pairs.

host: {

[key: string]: string;

}

Angular automatically checks host property bindings during change detection. If a binding changes, Angular updates the directive's host element.

When the key is a property of the host element, the property value is propagated to the specified DOM property.

When the key is a static attribute in the DOM, the attribute value is propagated to the specified property in the host element.

For event handling:

- The key is the DOM event that the directive listens to. To listen to global events, add the target to the event name. The target can be window, document or body.
- The value is the statement to execute when the event occurs. If the statement evaluates to false, then preventDefault is applied on the DOM event. A handler method can refer to the \$event local variable.

jit:

If true, this directive/component will be skipped by the AOT compiler and so will always be compiled using JIT.

jit: true

This exists to support Ivy's future work and currently has no effect.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Nested/Child Component in Angular:

A nested component is one component inside another component, or we can say it is a parent-child component.

A Component in Angular can have child components. Also, those child components can have their own further child components.

The first question that might be raised in our mind: "Does Angular framework support these types of components?" The answer is yes. We can put any number of components within another component. Also, Angular supports the nth level of nesting in general.

So, Angular seamlessly supports nested components.

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.

We will learn how to create a Child or Nested Components and host it in the AppComponent of our Angular Application. In our earlier example, we created an AppComponent, which is the root component of our application. The AppComponent is bootstrapped in the app.module.ts file and loaded in the index.html file using the directive.

How to add Child Component:

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

Adding a Child Component in Angular

Now, let us add a Child Component to our project. In our child component, let us add a component, which shows the list of customers.

1. Creating the Child Component

Creating the Child Component is no different to creating any other Parent Component.

First, we need a customer class

Creating the Customer Class

Go to the app folder and create a new folder named **customer** and then in this folder create a new file and name it as **customer.ts**.

customer/customer.ts:

```
export class Customer {  
    customerId: number;  
    name:string ;  
    address:string;  
    city:string;  
    state:string;  
    country:string;  
}
```

Note that we have used the **export keyword**. This enables us to use the above class in our components by importing it.



Creating the Child Component

In the **customer** folder of **app** folder create a new file name it as **customer-list.component.ts**.

customer/customer-list.component.ts:

```

import { Component } from '@angular/core';
import { Customer } from './customer';

@Component({
  selector: 'app-customerlist',
  templateUrl: './customer-list.component.html'
})
export class CustomerListComponent
{
  customers: Customer[] = [
    {customerId: 1, name: 'David', address: 'S.R. Nagar', city: 'Hyderabad', state: 'Telangana', country: 'India'},
    {customerId: 2, name: 'Smith', address: 'Andheri', city: 'Mumbai', state: 'Maharastra', country: 'India'},
    {customerId: 3, name: 'Fleming', address: 'B.R. Nagar', city: 'Kolkata', state: 'West Bengal', country: 'India'},
    {customerId: 4, name: 'Martin', address: 'Manas Nagar', city: 'Patna', state: 'Bihar', country: 'India'},
    {customerId: 5, name: 'Peter', address: 'Chandni Chowk', city: 'Delhi', state: 'Delhi', country: 'India'}
  ]
}

```

First, we import the required modules & classes. Our component requires Customer class, hence we import it along with the component module

```

import { Component } from '@angular/core';
import { Customer } from './customer';

```

The next step is to add the **@Component Directive**. The selector clause has the value “app-customerlist”. We need to use this in our parent view to display our view. The **templateURL** is “./customer-list.component.html”.

```

@Component({
  selector: 'app-customerlist',
  templateUrl: './customer-list.component.html'
})

```

The last step is to create the **Component class CustomerListComponent**. The class consists a single property **customer**, which is a collection of customers. We have initialized the collection with the few hard coded customers (static data).

```

export class CustomerListComponent
{
  customers: Customer[] = [
    {customerId: 1, name: 'David', address: 'S.R. Nagar', city: 'Hyderabad', state: 'Telangana', country: 'India'},
    {customerId: 2, name: 'Smith', address: 'Andheri', city: 'Mumbai', state: 'Maharastra', country: 'India'},
    {customerId: 3, name: 'Fleming', address: 'B.R. Nagar', city: 'Kolkata', state: 'West Bengal', country: 'India'},
    {customerId: 4, name: 'Martin', address: 'Manas Nagar', city: 'Patna', state: 'Bihar', country: 'India'},
    {customerId: 5, name: 'Peter', address: 'Chandni Chowk', city: 'Delhi', state: 'Delhi', country: 'India'}
  ]
}

```

Creating the View:

The next step is to create the view to display the list of customer. Go to the customer folder under the app folder and create the file with the name customer-list.component.html.

customer/customer-list.component.html:

```
<h3>List of Customers</h3>
<table border="1" style="border-collapse:collapse">
  <tr>
    <th>Customer Id</th>
    <th>Name</th>
    <th>Address</th>
    <th>City</th>
    <th>State</th>
  </tr>
  <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>
</table>
```

To iterate through the customers collection, we have used the **ngFor** Directive provided by the Angular. The Syntax for **ngFor** Directive starts with ***ngFor**. The expression “let customer of customers” is assigned to ***ngFor**. The let clause assigns the instance of customer object from the Customers collection to the local variable customer.

The local variable customer is used to build the template to display the details of the customer to the user. The **ngFor** directive is applied to the **tr** element of the table. The Angular repeats everything inside the **tr** element in the DOM tree.

2. Register the Child Component in the Module

Now, we have built the child component. We need to register it in our app module.

To add component to a module, you need to import it the app module file and declare it the declaration array. Every component/directive that we build must be declared in the NgModule. A Component cannot be part of more than one module.

The Module, which loads first is known as the root Module. Since our application has only one module (app.module), it will automatically become the root module

Open the app.module.ts under the app folder and update the code as shown below

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { CustomerListComponent } from './customer/customer-list.component';
@NgModule({
  declarations: [
    AppComponent, CustomerListComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Registering the component or directive in the module requires two steps:

First, import it

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

And, then declare it in declarations array

```
@NgModule({
  declarations: [
    AppComponent, CustomerListComponent
  ],
})
```

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 AppComponent. Open the app.component.html and add the following template:

app.component.html:

```
<h1>
  Welcome to {{ title }}!
</h1>
<app-customerlist></app-customerlist>
```

The @Component decorator of the customer-list.component, we provided the "app-customerlist" as selector in the metadata for the component. This CSS selector name must match the element tag that specified within the parent component's template.

```
<app-customerlist></app-customerlist>
```

Run the application from the command line/terminal using **ng serve --open**

It shows the output as following:

Welcome to DemoAngularApp1! Root/Parent Component (AppComponent)

List of Customers Child Component (CustomerListComponent)

Customer Id	Name	Address	City	State
1	David	S.R. Nagar	Hyderabad	Telangana
2	Smith	Andheri	Mumbai	Maharashtra
3	Fleming	B.R. Nagar	Kolkata	West Bengal
4	Martin	Manas Nagar	Patna	Bihar
5	Peter	Chandni Chowk	Delhi	Delhi

Welcome to DemoAngularApp1! Nested Component Parent - Child Component

List of Customers Child Component

Customer Id	Name	Address	City	State
1	David	S.R. Nagar	Hyderabad	Telangana
2	Smith	Andheri	Mumbai	Maharashtra
3	Fleming	B.R. Nagar	Kolkata	West Bengal
4	Martin	Manas Nagar	Patna	Bihar
5	Peter	Chandni Chowk	Delhi	Delhi

DevTools - localhost:4200

Elements Console Sources Network Performance Memory Application Security Audits

Styles Computed Event Listeners

Filter :hov .cls +

element.style { }

html[Attributes Style] { -webkit-Locale: "en"; }

html { user agent stylesheet }

display: block; color: -internal-root-color;

margin = border = padding = 1360 x 255.594 =

Component Communication in Angular:

The Components are useless if they do not share data between them. The Parent Component communicates with the child component using the **@Input Decorator**. The child components detect changes to these Input properties using OnChanges life Cycle hook or with a Property Setter. The child component can communicate with the parent by raising an event, which the parent can listen.

Angular Pass data to child component

We will learn how Angular passes the data to the child component. The Angular Components communicate with each other using **@Input Decorator**. We also look at how child components detect changes to these Input properties using OnChanges life Cycle hook or with a Property Setter.

Passing data to a child/nested component

In the previous section, we **built an Angular Application** and then **added a child component** to it. We also looked at how Angular Component communicates with its View (templates) using the **data binding**.

These Components are useless if they are not able to communicate with each other. They need to communicate with each other if they want to serve any useful purpose.

How to pass data to a 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

1. Import the **Input** module from **@angular/core** Library
2. Mark those property, which you need data from parent as input property using **@Input Decorator**

In the Parent Component

1. Bind the Child Component property in the Parent Component when instantiating the Child

@Input Decorator

The **@Input Decorator** is used to configure the input properties of the component. This decorator also supports change tracking.

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.

@Input example

Now let us build a simple component to demonstrate the use of **@Input**.

Our application will have a counter which is incremented/decremented by the Parent Component. The Parent Component then passes this counter to the child component for display in its template.

The Child Component with @Input Decorator

Create the child.component.ts under the child folder of app folder with the following code:

child -> child.component.ts:

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

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

@Component({
  selector: 'app-child',
  template: `<h2>Child Component</h2>
              <b>Current Count Is: </b> {{count}}
`)

export class ChildComponent
{
  @Input() count: number;
}
```

Now, let us look at the code in detail

First, we import the **Input** module from **@angular/core**

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

We have defined the inline template for the child component, where it displays the current count.

```
@Component({
  selector: 'app-child',
  template: `<h2>Child Component</h2>
              <b>Current Count Is: </b> {{count}}
`)
```

The Child Component expects the count to come from the Parent Component. Hence in ChildComponent decorate the count property with **@Input** decorator

```
export class ChildComponent
{
  @Input() count: number;
}
```

Bind to Child Property in Parent Component

Now, time to pass the Count values to the Child Component from the Parent

Open the **app.component.ts** and write the following code:

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

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  preserveWhitespaces: true
})
```

```
export class AppComponent {
  title: string = 'TestAngularApp1';
  counter: number = 1
```

```
Increment(){
  this.counter++;
}
```

```
Decrement(){
  this.counter--;
}
```

app.component.html:

```
<h1>Welcome to {{title}}</h1>
<button (click)="Increment()">Increment</button>
<button (click)="Decrement()">Decrement</button>
<app-child [count]="counter"></app-child>
```

The external template (**app.component.html**) in the Parent Component has two buttons. The Buttons Increments/Decrements the counter.

```
<button (click)="Increment()">Increment</button>
<button (click)="Decrement()">Decrement</button>
```

In the next line, we are invoking the child component inside

```
<app-child [count]="counter"></app-child>
```

Here, we are using count property, which is a property of the Child Component inside the square bracket. We bind it to counter property of the Parent Component.

Remember square bracket represents the **Property Binding in Angular**.

Finally, we will add counter in Parent component and set it to 1 as shown below.

```
export class AppComponent {
  title:string = 'TestAngularApp1';
  counter: number = 1;

  Increment(){
    this.counter++;
  }

  Decrement(){
    this.counter--;
  }
}
```

That's it.

Now run the Code and you should see the following displayed in the browser



Welcome to DemoAngularApp1

Child Component

Current Count Is: 1

Click on Increment & Decrement buttons to see that the changes are propagated to the child component.

Various ways to use @Input Decorator

We used input **@Input** decorator to mark the property in child component as input property. There are two ways you can do in Angular.

1. Using the **@Input** decorator to decorate the class property
2. Using the inputs array metadata of the component decorator

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Using the @Input decorator to decorate the class property

We saw this in our above example.

child.component.ts

```
export class ChildComponent {
  @Input() count: number;
}
```

Using the input array metadata of the component decorator

The same result can be achieved by using Input array of the @Component decorator as shown below:

child.component.ts:

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

@Component({
  selector: 'app-child',
  template: `<h2>Child Component</h2>
    <b>Current Count Is: </b> {{count}}
  `,
  inputs:['count']
})

export class ChildComponent { count: number; }
```

We have moved the count property to inputs array of the component metadata.

Aliasing input Property

You can Alias the input property and use the aliased name the parent component as shown below:

child.component.ts:

```
export class ChildComponent {
  @Input('MyCount') count: number;
}
```

Here, we are aliasing count property with **MyCount** alias

In the parent component, we can use the **MyCount** as shown below:

app.component.html:

```
<h1>Welcome to {{title}}</h1>
<button (click)="Increment()">Increment</button>
<button (click)="Decrement()">Decrement</button>
<app-child [MyCount]="counter"></app-child>
```

Detecting the Input changes

We looked at how to pass the data from parent to the child using **@Input decorator** and **property binding**.

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

1. Using OnChanges LifeCycle Hook

2. Using Input Setter

Let us look at both the methods in detail

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 **SimpleChanges** 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 **@angular/core** library.
2. Implement the **ngOnChanges()** method. The method receives the **SimpleChanges** object containing the changes each input property.

Let us update our Child Component to use the **OnChanges** hook

Open the **child.component.ts** and make the following changes:

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

@Component({
  selector: 'app-child',
  template: `<h2>Child Component</h2>
    <b>Current Count Is: </b> {{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);
      }
    }
  }
}
```

First, we are importing the required libraries

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

Next, Modify the class to implement the OnChanges interface

```
export class ChildComponent implements OnChanges {
```

ngOnChanges method:

```
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);
    }
  }
}
```

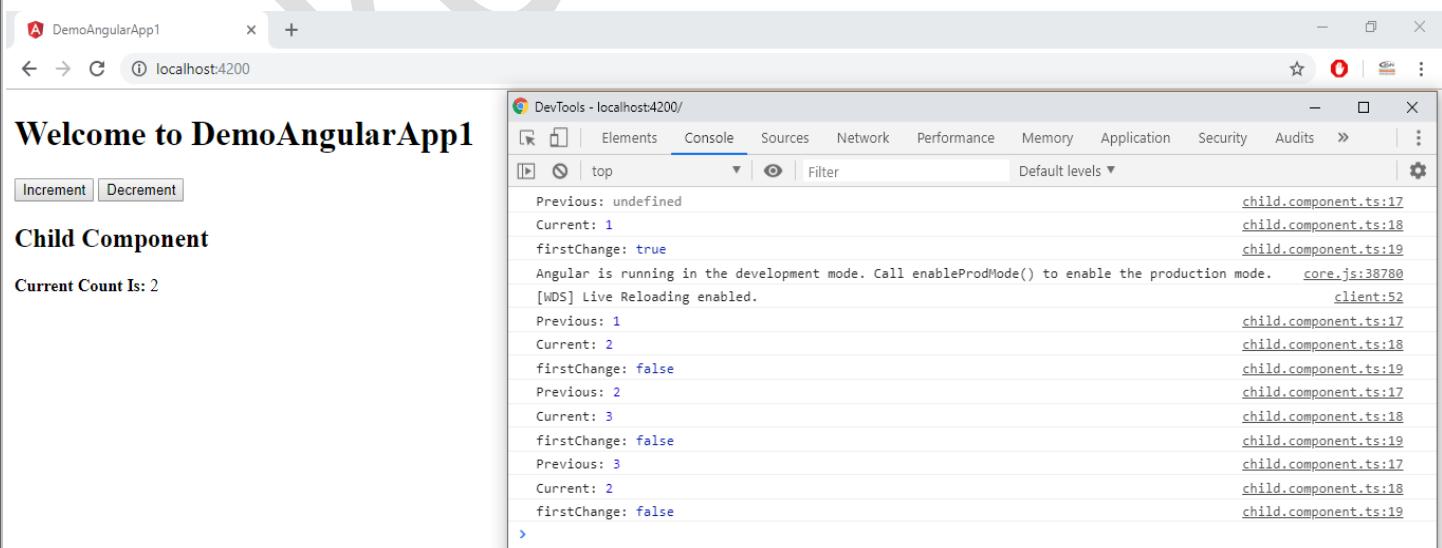
This method receives all the changes made to the input properties as **SimpleChanges** object. The **SimpleChanges** object whose keys are property names and values are instances of **SimpleChange**.

SimpleChange class represents a basic change from a previous to a new value. It has three class members.

Property Name	Description
previousValue:any	Previous value of the input property.
currentValue:any	New or current value of the input property.
FirstChange:boolean	Boolean value, which tells us whether it was the first time the change has taken place.

And we loop through the **SimpleChanges** to get our property count

Run the code and open the console log to watch the logs as you click on Increment and Decrement buttons in parent component.



The screenshot shows a browser window for 'DemoAngularApp1' at 'localhost:4200'. The page content includes a title 'Welcome to DemoAngularApp1', a 'Child Component' section, and a 'Current Count Is: 2' message. Below the browser is the 'DevTools - localhost:4200/' panel, specifically the 'Console' tab. The console output shows the following log entries:

```

Welcome to DemoAngularApp1
Child Component
Current Count Is: 2
[...]
DevTools - localhost:4200/
[...]
Previous: undefined
Current: 1
firstChange: true
Angular is running in the development mode. Call enableProdMode() to enable the production mode. core.js:38780
[WDS] Live Reloading enabled.
[...]
Previous: 1
Current: 2
firstChange: false
Previous: 2
Current: 3
firstChange: false
Previous: 3
Current: 2
firstChange: false
[...]

```

Using Input Setter

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

In the Child Component create a private property called _count

```
private _count = 0;
```

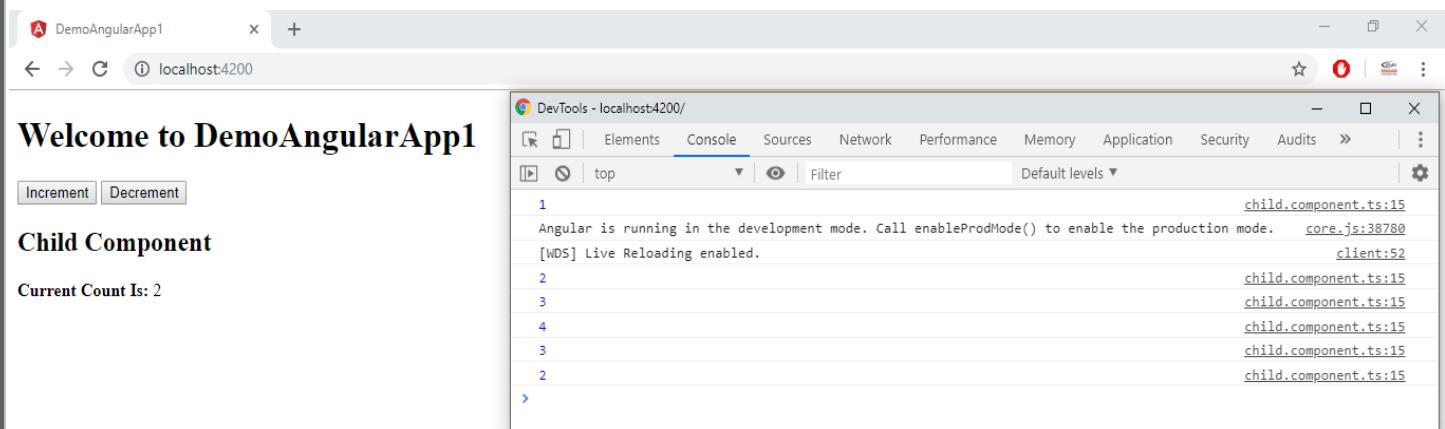
Create getter & setter on property count and attach @Input decorator. We intercept the input changes from setter function and log it to console.

child.component.ts:

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

@Component({
  selector: 'app-child',
  template: `<h2>Child Component</h2>
    <b>Current Count Is: </b> {{count}}
  `
})

export class ChildComponent {
  private _count = 0;
  @Input()
  set count(count: number) {
    this._count = count;
    console.log(count);
  }
  get count(): number { return this._count; }
}
```



The screenshot shows a browser window titled "DemoAngularApp1" at "localhost:4200". The page content includes a title "Welcome to DemoAngularApp1", a button group with "Increment" and "Decrement" buttons, and a section titled "Child Component" with the text "Current Count Is: 2". To the right, the DevTools Console tab is open, displaying the following logs:

```
1
Angular is running in the development mode. Call enableProdMode() to enable the production mode. core.js:38780
[WDS] Live Reloading enabled.
2
3
4
3
2
>
```

Conclusion

We looked at how to pass data from parent to child Component. The Child Component decorates the property using the @Input decorator. In the Parent Component, we use property binding to bind it to the Property or method of Parent Component.

We can also track changes made to the Input Property either by Using hooking to ngOnChanges life cycle hook. Or using the Property setter.



Angular Pass data to parent component

We will learn how to pass data to Parent Component from Child Component in Angular. In the above section, we looked at how the **Parent component communicates with its child** by setting its input property. The Child can send data to Parent by raising an event, Parent can interact with the child via local variable or Parent can call @ViewChild on the child. We will look at all these options.

Pass data to parent component

There are three ways in which parent component can interact with the child component

1. Parent Listens to Child Event
2. Parent uses Local Variable to access the child
3. Parent uses a @ViewChild to get reference to the child component

Let us look at each of those scenarios in detail

Parent listens for child event

The Child Component exposes an **EventEmitter** Property. This Property is decorated with the **@Output** decorator. When Child Component needs to communicate with the parent it raises the event. The Parent Component listens to that event and reacts to it.

EventEmitter Property

To raise an event, the component must declare an **EventEmitter** Property. The Event can be emitted by calling the **.emit()** method

For Example:

```
countChanged: EventEmitter<number> = new EventEmitter();
```

And then call **emit()** method passing the whatever the data you want to send as shown below:

```
this.countChanged.emit(this.count);
```

@Output Decorator

Using the **EventEmitter** Property gives the components ability to raise an event. But to make that event accessible from parent component, you must decorate the property with **@Output** decorator.

How to pass data to parent component using @Output

In the child component

1. Declare a property of type **EventEmitter** and instantiate it
2. Mark it with a **@Output** decorator
3. Raise the event passing it with the desired data

In the Parent Component

1. Bind to the Child Component using **Event Binding** and listen to the child events
2. Define the event handler function

Passing data from child to parent component via Events (Example)

Now let us build an application to demonstrate this

In the last **passing data from parent to child component** example, we built a counter in the parent component. We assigned the initial value to the counter and added increment/decrement methods. In the child Component, we used the @Input decorator to bind count property to the parent component. Whenever parent count is changed in the parent the child component is updated and displayed the count.

In this example, we will move the counter to the child component. We will raise an event in the child component whenever the count is increased or decreased. Then we bind to that event in the parent component and display the count in the parent component.

Child Component

Open the child.component.ts and write the following code:

child.component.ts:

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

@Component({
  selector: 'app-chlid',
  template: `<h2>Child Component</h2>
    <button (click)="Increment()">Increment</button>
    <button (click)="Decrement()">Decrement</button>
  `,
  preserveWhitespaces:true
})
export class ChildComponent {
  count: number=1;

  @Output() countChanged: EventEmitter<number> = new EventEmitter();

  Increment() {
    this.count++;
    this.countChanged.emit(this.count);
  }

  Decrement() {
    this.count--;
    this.countChanged.emit(this.count);
  }
}
```

Now, let us look at the code in detail

First, as usual, we need to import **Output & EventEmitter** from **@angular/core library**

child.component.ts:

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

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

In the inline template, we have two buttons Increment and Decrement.

child.component.ts:

```
@Component({
  selector: 'app-chlid',
  template: `<h2>Child Component</h2>
    <button (click)="Increment()">Increment</button>
    <button (click)="Decrement()">Decrement</button>
  `,
  preserveWhitespaces:true
})
```

In the child component, define the **countChanged** event of type **EventEmitter**. Decorate the property with **@Output** decorator to make it accessible from the parent component

child.component.ts:

```
@Output() countChanged: EventEmitter<number> = new EventEmitter();
```

Finally, we raise the event in **Increment()** & **Decrement()** methods using **emit**.

child.component.ts:

```
Increment() {
  this.count++;
  this.countChanged.emit(this.count);
}
```

```
Decrement() {
  this.count--;
  this.countChanged.emit(this.count);
}
```

Parent Component:

In the parent component, we need to listen to the “countChanged” event

The “countChanged” event is enclosed in parentheses. It is then assigned to the method “**countChangedHandler**” in the component class. The syntax is similar to **Event Binding**

app.component.html:

```
<h1>Welcome to {{title}}</h1>
<h2>Parent Component</h2>
<p><b>Current Count is: </b> {{counter}} </p>
<app-chlid (countChanged)="countChangedHandler($event)"></app-chlid>
```

The **countChangedHandler(\$event)** method accepts the **\$event** argument. The data associated with event is now available to in the **\$event** property

Our **CountChangedHandler** is as follows. It just updates the **counter** and also logs the count to console.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.component.ts:

```
countChangedHandler(count: number) {
    this.counter = count;
    console.log(count);
}
```

The complete code is as follows:

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

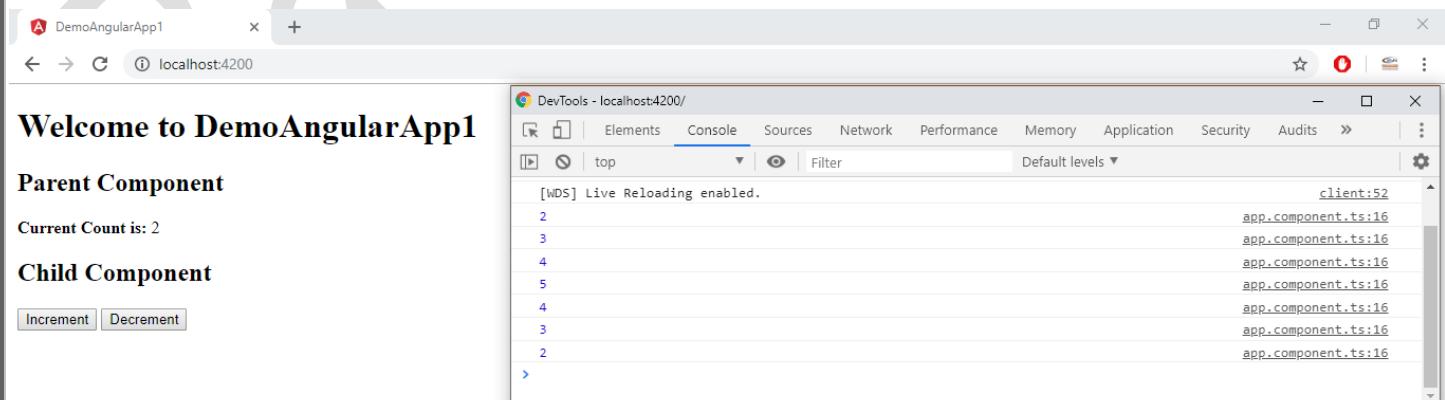
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  preserveWhitespaces: true
})

export class AppComponent implements OnInit {
  title: string = 'DemoAngularApp1';
  counter: number = 1;

  ngOnInit() {
    console.log(this.counter);
  }

  countChangedHandler(count: number) {
    this.counter = count;
    console.log(count);
  }
}
```

Run the code. Whenever the Increment/Decrement buttons clicked from child component, it raises the event. The Parent component gets notified of this and updates the counter with the latest value.



A screenshot of a browser window showing the Angular application "Welcome to DemoAngularApp1". The page has two sections: "Parent Component" and "Child Component". The "Parent Component" shows the message "Current Count is: 2". The "Child Component" has two buttons: "Increment" and "Decrement". Below the buttons, the "Console" tab of the DevTools is open, showing the following log entries:

```
[WDS] Live Reloading enabled.
2
3
4
5
4
3
2
>
```

Parent uses local variable to access the Child in Template

Parent Template can access the child component properties and methods by creating the **template reference variable**.

Child Component

Let us update the child component

child.component.ts:

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

@Component({
  selector: 'app-chlid',
  template: `<h2>Child Component</h2>
    <p><b>Current Count Value in Child is: </b>{{count}}</p>
  `
})
export class ChildComponent {
  count: number=1;

  Increment() {
    this.count++;
    console.log(`Count Value in Child Is ${this.count}`);
  }

  Decrement() {
    this.count--;
    console.log(`Count Value in Child Is ${this.count}`);
  }
}
```

We have removed the **Input, Output & EventEmitter**.

Our component is now have property **count** and two methods to **Increment()** and **Decrement()**

Parent Component:

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl:'./app.component.html',
  styleUrls:[ './app.component.css'],
  preserveWhitespaces:true
})

export class AppComponent {
  title:string = 'DemoAngularApp1';
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.component.html:

```
<h1>Welcome to {{title}}</h1>
<h2>Parent Component</h2>
<p><b>Current Count Value in Parent is: </b> {{child.count}} </p>
<button (click)="child.Increment()">Increment</button>
<button (click)="child.Decrement()">Decrement</button>
<app-chlid #child></app-chlid>
```

We have created a local variable, **#child**, on the tag **<app-child>**. The “**child**” is called **template reference variable**, which is now represents the child component.

The Template Reference variable is created, when you use **#<variableName>** and attach it to a DOM element. You can then, use the variable to reference the DOM element in your Template.

app.component.html:

```
<app-chlid #child></app-chlid>
```

Now you can use the local variable elsewhere in the template to refer to the child component methods and properties as shown below:

app.component.html:

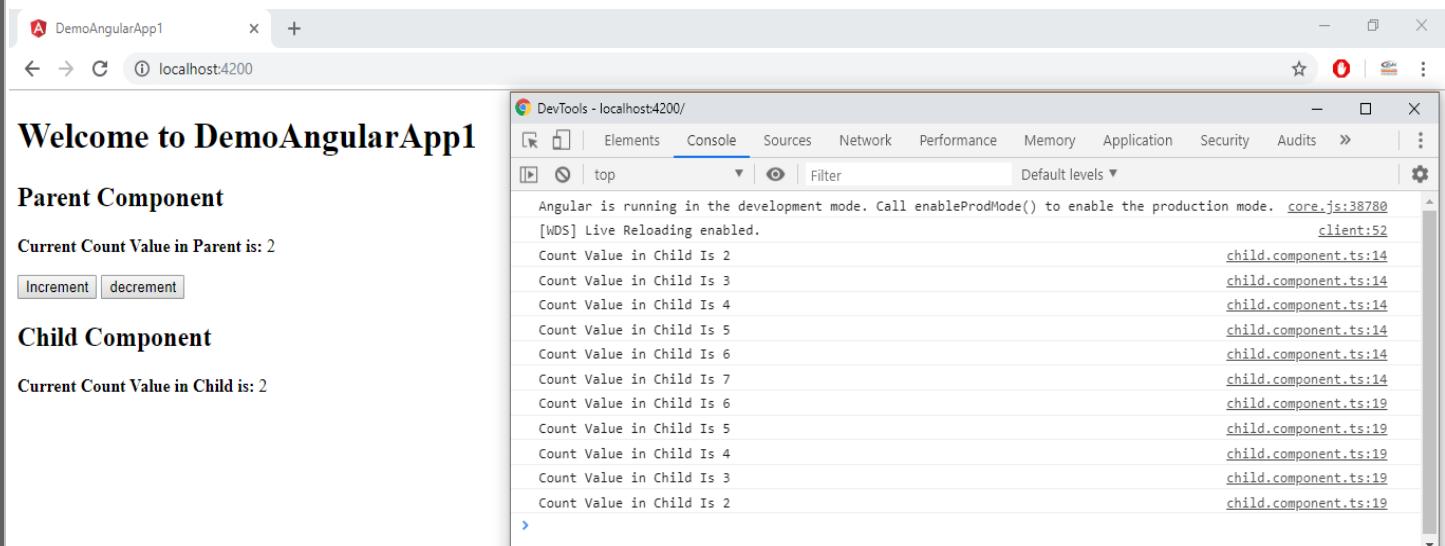
```
<p><b>Current Count Value in Parent is: </b> {{child.count}} </p>
<button (click)="child.Increment()">Increment</button>
<button (click)="child.Decrement()">Decrement</button>
```

The code above wires child components Increment & Decrement methods from the parent component.

The local variable approach is simple and easy. But it is limited because the parent-child wiring must be done entirely within the parent template. The parent component itself has no access to the child.

You can't use the local variable technique if an instance of the parent component class must read or write child component values or must call child component methods.

Run the code. Whenever the Increment/Decrement buttons clicked from the parent component, it calls the child component methods and count value gets changed there. Then Parent component DOM gets notified of this and updates the count with the latest value.



```
Angular is running in the development mode. Call enableProdMode() to enable the production mode. core.js:38780
[WDS] Live Reloading enabled.
Count Value in Child Is 2
Count Value in Child Is 3
Count Value in Child Is 4
Count Value in Child Is 5
Count Value in Child Is 6
Count Value in Child Is 7
Count Value in Child Is 8
Count Value in Child Is 9
Count Value in Child Is 10
Count Value in Child Is 11
Count Value in Child Is 12
Count Value in Child Is 13
Count Value in Child Is 14
Count Value in Child Is 15
Count Value in Child Is 16
Count Value in Child Is 17
Count Value in Child Is 18
Count Value in Child Is 19
```

Parent uses a **@ViewChild()** to get reference to the Child Component:

Injecting an instance of the child component into the parent as a **@ViewChild** is the another technique used by the parent to access the property and method of the child component

The **@ViewChild** decorator takes the name of the **component/directive** as its input. It is then used to decorate a property. The Angular then injects the reference of the component to the Property.

For Example

In the Parent component, declare a property **child** which is of type **ChildComponent**

```
child: ChildComponent;
```

Next, decorate it with **@ViewChild** decorator passing it the name of the component to inject.

```
@ViewChild(ChildComponent, {static: true}) child: ChildComponent;
```

Now, when angular creates the child component, the reference to the child component is assigned to the **child** property.

We now update the code from previous example

Child Component

There is no change in the child component.

Parent Component

In parent component, we need to import the **ViewChild** annotation. We also need to import the **ChildComponent**

```
import { Component, ViewChild} from '@angular/core';
import { ChildComponent } from './child/child.component';
```

Next, create a property **child** which is an instance of type **ChildComponent**. Apply the **ViewChild** annotation on the child component as shown below:

```
@ViewChild(ChildComponent, {static: true}) child: ChildComponent;
```

Finally, add Increment and Decrement method, which invokes the methods in the child component

```
Increment() {
  this.child.Increment();
}

Decrement() {
  this.child.Decrement();
}
```

Now, the parent can access the properties and methods of child component.

And in the template make necessary changes in **app.component.html**:

```
<h1>Welcome to {{title}}</h1>
<h2>Parent Component</h2>
<p><b>Current Count Value in Parent is: </b> {{child.count}} </p>
<button (click)="Increment()">Increment</button>
<button (click)="Decrement()">Decrement</button>
<app-chlid></app-chlid>
```

The complete `app.component.ts` is as follows:

```
import { Component, ViewChild } from '@angular/core';
import { ChildComponent } from './child/child.component';
import { from } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  preserveWhitespaces: true
})

export class AppComponent {
  title: string = 'DemoAngularApp1';

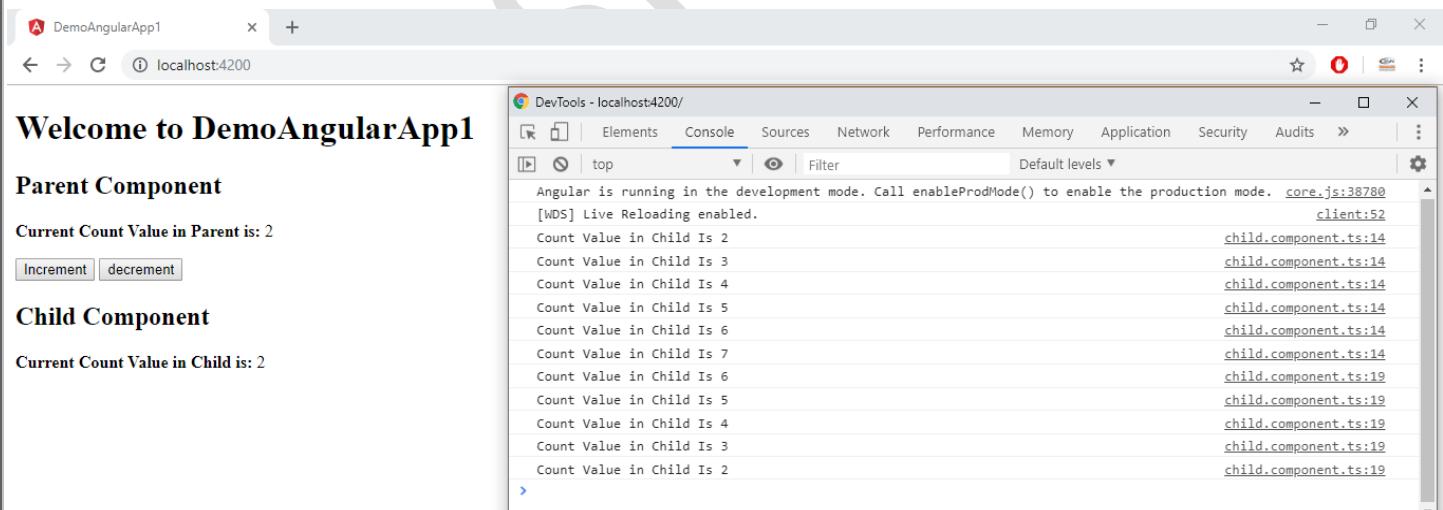
  @ViewChild(ChildComponent, { static: true }) child: ChildComponent;

  Increment() {
    this.child.Increment();
  }

  Decrement() {
    this.child.Decrement();
  }
}
```

That's all.

Run the application.



```
Angular is running in the development mode. Call enableProdMode() to enable the production mode. core.js:38780
[WDS] Live Reloading enabled.
Count Value in Child Is 2
Count Value in Child Is 3
Count Value in Child Is 4
Count Value in Child Is 5
Count Value in Child Is 6
Count Value in Child Is 7
Count Value in Child Is 6
Count Value in Child Is 5
Count Value in Child Is 4
Count Value in Child Is 3
Count Value in Child Is 2
```

Conclusion

In this, we looked at how the parent can communicate with the child component. The Parent can subscribe to the **events of the child component**. It can use the **template reference variable** to access the properties and methods. We can also use **@ViewChild decorator** to inject the child component instance to the parent.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



Component Life-Cycle Hooks in Angular:

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 behaviour 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 its 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.

Every component in Angular has its own lifecycle events that occurs as the component gets created, renders, changes its property values or gets destroyed. Angular invokes certain set of methods or we call them hooks that gets executed as soon as those lifecycle events gets fired.

The number of methods called in a sequence to execute a component at specific moments is known as lifecycle hook.

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()`

So, in Angular, every component has a life-cycle, a number of different stages it goes through. There are 8 different stages in the component lifecycle. Every stage is called as lifecycle hook event. So, we can use these hook events in different phases of our application to obtain control of the components. Since a component is a TypeScript class, every component must have a constructor method. The constructor of the component class executes, first, before the execution of any other lifecycle hook events. If we need to inject any dependencies into the component, then the constructor is the best place to inject those dependencies. After executing the constructor, Angular executes its lifecycle hook methods in a specific order.

Lifecycle hooks are wrapped in certain interfaces which are included in the angular core '@angular/core' library.

One thing to note that each of these interfaces includes one method whose name is same as of the interface name but it is just prefixed by "ng". For example `OnInit` interface has one method `ngOnInit()`.

The following lifecycle hooks are exposed by Angular corresponding to different lifecycle events.

The first step of the lifecycle is to create a component by calling its **constructor**.

Once the component is created, the lifecycle hook methods are called by Angular in the following sequence:

- `ngOnChanges()`
- `ngOnInit()`
- `ngDoCheck()`
- `ngAfterContentInit()`
- `ngAfterContentChecked()`
- `ngAfterViewInit()`
- `ngAfterViewChecked()`
- `ngOnDestroy()`

Each hook defined above is invoked by Angular when certain events occurs.

ngOnChanges:

The angular invokes this life cycle hook whenever any data-bound property of 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. This method is called **once** on component's creation and then **every time** changes are detected in one of the component's *input* properties. It receives a **SimpleChanges** object as a parameter, which contains information regarding which of the *input* properties has changed - in case we have more than one - and its current and previous values.

Child Component:

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

@Component({
  selector: 'app-child',
  template: `
    <h3>Child Component</h3>
    <p>TICKS: {{ lifecycleTicks }}</p>
    <p>DATA: {{ data }}</p>
  `
})

export class ChildComponent implements OnChanges {
  @Input() data: string;
  lifecycleTicks: number = 0;

  ngOnChanges() {
    this.lifecycleTicks++;
  }
}
```

Parent Component:

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

@Component({
  selector: 'app-parent',
  template: `
    <h1>ngOnChanges Example</h1>
    <app-child [data]="arbitraryData"></app-child>
  `
})

export class ParentComponent {
  arbitraryData: string = 'initial';

  constructor() {
    setTimeout(() => {
      this.arbitraryData = 'final';
    }, 5000);
  }
}
```

Summary: ParentComponent binds input data to the ChildComponent. The component receives this data through its @Input property. ngOnChanges fires. After five seconds, the setTimeout callback triggers. ParentComponent mutates the data source of ChildComponent's input-bound property. The new data flows through the input property. ngOnChanges fires yet again.

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 initialization logic for your component. Note that ngOnChanges hook is fired before ngOnInit. Which means all the input properties are available to use when ngOnInit hook is called. This hook is fired only once. The hook does not fire as ChildComponent receives the input data. Rather, it fires right after the data renders to the ChildComponent template.

Child Component:

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

@Component({
  selector: 'app-child',
  template: `
    <h3>Child Component</h3>
    <p>TICKS: {{ lifecycleTicks }}</p>
    <p>DATA: {{ data }}</p>
  `
})

export class ChildComponent implements OnInit {

  @Input() data: string;

  lifecycleTicks: number = 0;

  ngOnInit() {
    this.lifecycleTicks++;
  }
}
```

Parent Component:

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

@Component({
  selector: 'app-parent',
  template: `
    <h1>ngOnInit Example</h1>
    <app-child [data]="arbitraryData"></app-child>
  `
})


```

```
export class ParentComponent {
    arbitraryData: string = 'initial';

    constructor() {
        setTimeout(() => {
            this.arbitraryData = 'final';
        }, 5000);
    }
}
```

Summary: ParentComponent binds input data to the ChildComponent. ChildComponent receives this data through its @Input property. The data renders to the template. ngOnInit fires. After five seconds, the setTimeout callback triggers. ParentComponent mutates the data source of ChildComponent's input-bound property. ngOnInit **DOES NOT FIRE**. ngOnInit is a one-and-done hook. Initialization is its only concern.

Constructor vs ngOnInit?

Probably you have wondered why place your initialization logic inside ngOnInit when you can do it in the class constructor. Well, basically the constructor is best left to be used for dependency injection and our initialization logic should be put on ngOnInit.

That is because the JavaScript engine is what handles the constructor, not Angular. And this is one of the reasons why the ngOnInit hook was created, which is called by Angular and becomes part of the component's lifecycle that is managed by it. Also, of course, due to the fact that you **can't yet access component's Input properties on the constructor**.

ngDoCheck

This hook can be interpreted as an “extension” of ngOnChanges. You can use this method to **detect changes that Angular can't or won't detect**. It is called in **every** change detection, immediately after the ngOnChanges and ngOnInit hooks.

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

Normally Angular automatically does the change detection of properties and events and updates the view or invokes any other event accordingly. But sometimes, we need to execute some functionality of keep checking certain thing on every change detection. This is where ngDoCheck does the job for us.

Please note that implementing this method is very costly as this gets executed on every change cycle. So do try to avoid it and use if it is really required.

```
import { Component, DoCheck, ChangeDetectorRef } from '@angular/core';
@Component({
    selector: 'app-example',
    template: `
        <h1>ngDoCheck Example</h1>
        <p>DATA: {{ data[data.length - 1] }}</p>
    `
})

```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

```

export class ExampleComponent implements DoCheck {
  lifecycleTicks: number = 0;
  oldTheData: string;
  data: string[] = ['initial'];

  constructor(private changeDetector: ChangeDetectorRef) {
    this.changeDetector.detach(); // lets the class perform its own change detection

    setTimeout(() => {
      this.oldTheData = 'final';
      this.data.push('intermediate');
    }, 3000);

    setTimeout(() => {
      this.data.push('final');
    }, 6000);
  }

  ngDoCheck() {
    console.log(this.lifecycleTicks++);

    if (this.data[this.data.length - 1] !== this.oldTheData) {
      this.changeDetector.detectChanges();
    }
  }
}

```

Pay attention to the console versus the display. The data progress up to 'intermediate' before freezing. Three rounds of change detection occur over this period as indicated in the console. One more round of change detection occurs as 'final' gets pushed to the end of this.data. One last round of change detection then occurs. The evaluation of the 'if statement' determines no updates to the view are necessary.

Summary: Class constructor initiates setTimeout twice. After three seconds, the first setTimeout triggers change detection. ngDoCheck marks the display for an update. Three seconds later, the second setTimeout triggers change detection. No view updates needed according to the evaluation of ngDoCheck.

ngAfterContentInit

This hook method is called only **once** during the component's lifecycle, after the first *ngDoCheck*. Within this hook, we have access for the first time to the *ElementRef* of the *ContentChild* after the component's creation; after Angular has already projected the external content into the component's view.

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.

Content projection is a way to import HTML content from outside the component and insert that content into the component's template in a designated spot.

CComponent:

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

@Component({
  selector: 'app-c',
  template: `
    <p>I am C.</p>
    <p>Hello World!</p>
  `
})
export class CComponent { }
```

BComponent:

```
import { Component, ContentChild, AfterContentInit, ElementRef, Renderer2 } from '@angular/core';
import { CComponent } from './c.component';
@Component({
  selector: 'app-b',
  template: `
    <p>I am B</p>
    <ng-content></ng-content>
  `
})
```

```
export class BComponent implements AfterContentInit {
  @ContentChild("BHeader", { static:true, read: ElementRef }) hRef: ElementRef;
  @ContentChild(CComponent, {static:true, read: ElementRef }) cRef: ElementRef;

  constructor(private renderer: Renderer2) { }

  ngAfterContentInit() {
    this.renderer.setStyle(this.hRef.nativeElement, 'background-color', 'yellow')

    this.renderer.setStyle(this.cRef.nativeElement.children.item(0), 'background-color', 'pink');
    this.renderer.setStyle(this.cRef.nativeElement.children.item(1), 'background-color', 'red');
  }
}
```

AComponent:

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

@Component({
  selector: 'app-a',
  template: `
```

```

<h1>ngAfterContentInit Example</h1>
<p>I am A.</p>
<app-b>
<h3 #BHeader>BComponent Content DOM</h3>
<app-c></app-c>
</app-b>
`
```

)

```
export class AComponent { }
```

The **@ContentChild** query results are available from **ngAfterContentInit**. Renderer2 updates the content DOM of **BComponent** containing an **h3** tag and **CComponent**.

Summary: Rendering starts with **AComponent**. For it to finish, **AComponent** must render **BComponent**. **BComponent** projects content nested in its element through the **<ng-content></ng-content>** element. **CComponent** is part of the projected content. The projected content finishes rendering. **ngAfterContentInit** fires. **BComponent** finishes rendering. **AComponent** finishes rendering. **ngAfterContentInit** will not fire again.

ngAfterContentChecked:

This lifecycle hook method executes every time the content of the component has been checked by the change detection mechanism of Angular. This method is called after the **ngAfterContentInit()** method. This method is also called on every subsequent execution of **ngDoCheck()**.

CComponent:

```

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

@Component({
  selector: 'app-c',
  template: `
    <p>I am C.</p>
    <p>Hello World!</p>
`
```

)

```
export class CComponent { }
```

BComponent:

```

import { Component, ContentChild, AfterContentChecked, ElementRef, Renderer2 } from '@angular/core';
import {CComponent} from './c.component';
@Component({
  selector: 'app-b',
  template: `
    <p>I am B</p>
    <button (click)="$event">CLICK</button>
    <ng-content></ng-content>
`
```

)

```

export class BComponent implements AfterContentChecked {
  @ContentChild("BHeader", { static:false, read: ElementRef }) hRef: ElementRef;
  @ContentChild(CComponent, {static:false, read: ElementRef }) cRef: ElementRef;

  constructor(private renderer: Renderer2) { }

  randomRGB(): string {
    return `rgb(${Math.floor(Math.random() * 256)},
    ${Math.floor(Math.random() * 256)},
    ${Math.floor(Math.random() * 256)})`;
  }

  ngAfterContentChecked() {
    this.renderer.setStyle(this.hRef.nativeElement, 'background-color', this.randomRGB());
    this.renderer.setStyle(this.cRef.nativeElement.children.item(0), 'background-
color', this.randomRGB());
    this.renderer.setStyle(this.cRef.nativeElement.children.item(1), 'background-
color', this.randomRGB());
  }
}

```

AComponent:

```

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

@Component({
  selector: 'app-a',
  template: `
    <h1>ngAfterContentChecked Example</h1>
    <p>I am A.</p>
    <app-b>
      <h3 #BHeader>BComponent Content DOM</h3>
      <app-c></app-c>
    </app-b>
  `
})

export class AComponent { }

```

This hardly differs from **ngAfterContentInit**. A mere **<button></button>** was added to **BComponent**. Clicking it causes a change detection loop. This activates the hook as indicated by the randomization of background-color.

Summary: Rendering starts with **AComponent**. For it to finish, **AComponent** must render **BComponent**. **BComponent** projects content nested in its element through the **<ng-content></ng-content>** element. **CComponent** is part of the projected content. The projected content finishes rendering. **ngAfterContentChecked** fires. **BComponent** finishes rendering. **AComponent** finishes rendering. **ngAfterContentChecked** may fire again through change detection.



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**.

ngAfterViewInit waits on **@ViewChild()** queries to resolve.

CComponent:

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

@Component({
  selector: 'app-c',
  template: `
    <p>I am C.</p>
    <p>Hello World!</p>
  `
})
export class CComponent { }
```

BComponent:

```
import { Component, ViewChild, AfterViewInit, ElementRef, Renderer2 } from '@angular/core';

@Component({
  selector: 'app-b',
  template: `
    <p #BStatement>I am B.</p>
    <ng-content></ng-content>
  `
})

export class BComponent implements AfterViewInit {
  @ViewChild("BStatement", {static: true, read: ElementRef}) pStmt: ElementRef;
  constructor(private renderer: Renderer2) { }

  ngAfterViewInit() {
    this.renderer.setStyle(this.pStmt.nativeElement, 'background-color', 'yellow');
  }
}
```

AComponent:

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

@Component({
  selector: 'app-a',
  template: `
    <h1>ngAfterViewInit Example</h1>
    <p>I am A.</p>
    <app-b>
      <h3>BComponent Content DOM</h3>
      <app-c></app-c>
    </app-b>
  `
})
export class AComponent { }
```



It changes the background color of BComponent's paragraph with #BStatement. This indicates the view element was successfully queried.

Summary: Rendering starts with **AComponent**. For it to finish, **AComponent** must render **BComponent**. **BComponent** projects content nested in its element through the `<ng-content></ng-content>` element. **CComponent** is part of the projected content. The projected content finishes rendering. **BComponent** finishes rendering. `ngAfterViewInit` fires. **AComponent** finishes rendering. `ngAfterViewInit` will not fire again.

ngAfterViewChecked

This method is called after the `ngAfterViewInit()` method. It is executed every time the view of the given component has been checked by the change detection algorithm of Angular. This method executes after every subsequent execution of the `ngAfterContentChecked()`. This method also executes when any binding of the children directives has been changed. So this method is very useful when the component waits for some value which is coming from its child components.

CComponent:

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

@Component({
  selector: 'app-c',
  template: `
    <p>I am C.</p>
    <p>Hello World!</p>
  `
})
export class CComponent { }
```

BComponent:

```
import { Component, ViewChild, AfterViewChecked, ElementRef, Renderer2 } from '@angular/core';
@Component({
  selector: 'app-b',
  template: `
    <p #BStatement>I am B.</p>
    <button (click)="event">CLICK</button>
    <ng-content></ng-content>
  `
})
export class BComponent implements AfterViewChecked {
  @ViewChild("BStatement", {static: true, read: ElementRef}) pStmt: ElementRef;

  constructor(private renderer: Renderer2) { }

  randomRGB(): string {
    return `rgb(${Math.floor(Math.random() * 256)}, ${Math.floor(Math.random() * 256)}, ${Math.floor(Math.random() * 256)})`;
  }

  ngAfterViewChecked() {
    this.renderer.setStyle(this.pStmt.nativeElement, 'background-color', this.randomRGB());
  }
}
```

AComponent:

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

@Component({
  selector: 'app-a',
  template: `
    <h1>ngAfterContentInit Example</h1>
    <p>I am A.</p>
    <app-b>
      <h3>BComponent Content DOM</h3>
      <app-c></app-c>
    </app-b>
  `
})

export class AComponent { }
```

Summary: Rendering starts with **AComponent**. For it to finish, **AComponent** must render **BComponent**. **BComponent** projects content nested in its element through the **<ng-content></ng-content>** element. **CComponent** is part of the projected content. The projected content finishes rendering. **BComponent** finishes rendering. **ngAfterViewChecked** fires. **AComponent** finishes rendering. **ngAfterViewChecked** may fire again through change detection.

Clicking the **<button></button>** element initiates a round of change detection. **ngAfterContentChecked** fires and randomizes the background-color of the queried elements each button click.

ngOnDestroy

This method will be executed just before Angular destroys the components. This method is very useful for unsubscribing from the observables and detaching the event handlers to avoid memory leaks. Actually, it is called just before the instance of the component is finally destroyed. This method is called just before the component is removed from the DOM.

ChildComponent:

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

@Component({
  selector: 'app-child',
  template: `
    <h2>Child Component</h2>
  `
})

export class ChildComponent implements OnDestroy {

  constructor() {
    console.log("ChildComponent:Constructor");
  }

  ngOnDestroy(){
    console.log("Child Component: Destroy");
  }
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

AppComponent:

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

@Component({
  selector: 'app-root',
  template: `
    <h1>{{title}}</h1>
    <button (click)="toggle()">Hide/Show Child</button>
    <app-child *ngIf="displayChild"></app-child>
  `
})

export class AppComponent {
  title:string = 'DemoAngularApp1';

  displayChild: boolean=false;

  constructor() {
    console.log("AppComponent:Constructor");
  }

  toggle() {
    this.displayChild=!this.displayChild;
  }
}
```

Summary: When the button is clicked Child Component can be added/removed using the `*ngIf` directive. The structural directive `*ngIf` evaluates to true/false. `*ngIf` removes its child then `ngOnDestroy` of child component fires.

Microsoft .Net Solution

Develop Your **Microsoft** Skills and Knowledge through great Training



Hyderabad's Best Institute for .NET

Angular



Rakesh Singh

RAKESHSOFTNET TECHNOLOGIES Hyderabad

🌐 <http://www.rakeshsoftnet.com>

FACEBOOK <https://www.facebook.com/RakeshSoftNet>

FACEBOOK <https://www.facebook.com/RakeshSoftNetTech>

TWITTER <https://twitter.com/RakeshSoftNet>

CALL +91 40 4200 8807 MOBILE +91 89191 36822



Modules in Angular:

The building blocks of Angular Applications **consists of Components, Templates, Directives, Pipes, and Services**. We build a lot of such blocks to create the complete application. As the application grows bigger in size managing these blocks become difficult. The Angular Provides a nice way organize these blocks in a simple and effectively using Angular modules (Also known as **NgModules**).

The Module is an overloaded term. It means different things depending on the context. There are two kind of modules exists in Angular itself. One is the **JavaScript Module** and the other one is **Angular Module**.

What is Angular Module?

The Angular module (also known as **NgModule**) help us to organize the application parts into cohesive blocks of functionality. Each block is focused on providing a specific functionality or a feature.

The Angular module must implement a specific feature. The **Components, Directives, Pipes, and Services** which implement such a feature, will become part of that Module.

The Modular design helps to keep the Separation of concerns. Keep the features together. Makes it easy to maintain the code. Makes it easier to reuse the code.

The Angular itself is built on the concept of Modules. The '**@angular/core**' is the main Angular module which implements the Angular's core functionality, low-level services, and utilities.

The Features like **Forms**, **HTTP** and **Routing** are implemented as separate Feature modules such as **FormsModule** for working with forms, **HttpClientModule** for sending HTTP requests, and **RouterModule** for providing routing mechanism to Angular application. There are many third-party libraries available as NgModules such as such as **Material Design**, **Ionic**, and **AngularFire2** etc...

Angular Applications are assembled from the modules. The module exports Component, directive, service, pipe etc..., which can be then imported in another module.

JavaScript Modules vs. NgModules

JavaScript Modules

The JavaScript Modules, which also goes by the name JS modules or ES modules or ECMAScript modules are part of the JavaScript Language. The JS Modules are stored in a file. There is exactly one module per file and one file per module. These modules contains a small units of independent, reusable code. They export a value, which can be imported and used in some other module.

The following is a JavaScript Module which exports the **SomeComponent**.

```
export class SomeComponent {  
  //do something  
}
```

The **SomeOtherComponent** is another JavaScript Module, which imports and uses the **SomeComponent**.

```
import { SomeComponent } from './some.component';  
  
export class SomeOtherComponent {  
  //do some other thing  
}
```

NgModules:

Angular applications begin from the root NgModule. Angular manages an application's dependencies through its module system comprised of NgModules. Alongside plain JavaScript modules, NgModules ensure code modularity and encapsulation.

Modules also provide a top-most level of organizing code. Each NgModule sections off its own chunk of code as the root. This module provides top-to-bottom encapsulation for its code. The entire block of code can then export to any other module.

The NgModule is a TypeScript class decorated with **@NgModule** Decorator which is a fundamental feature of Angular.

The NgModule is a class and work with the **@NgModule** decorator function and also takes a metadata object that tells Angular how to compile and run module code.

The Angular module helps you to organize an application into associative blocks of functionality.

An angular module represents a core concept and plays a fundamental role in structuring Angular applications.

The NgModule is used to simplify the ways you define and manage the dependencies in your applications and also you can consolidate different components and services into associative blocks of functionality.

Every Angular application should have at least one module and it contains the components, service providers, pipes and other code files whose scope is defined by the containing NgModule.

The purpose of the module is to declare everything you create in Angular and group them together.

Every application has at least one Angular module, the root module that you bootstrap to launch the application. The Angular root module is called **AppModule**.

A module can import other modules and can expose its functionality to other modules.

The modules can be loaded eagerly when the application starts or lazy loaded asynchronously by the router.

Every Angular Application has at least one module. It is generally referred as "**root module**". This root module is used to bootstrap the Application. Normally, root module is required in the simple Application. As the Application grows, we need to refactor the root module into the different feature modules, which have related functionality. These feature modules are imported into the root module.

The angular loads a root dynamically because it is bootstrapped in the Angular Module.

Root Module - AppModule

Every Angular Application has a root module called "**AppModule**" (**app.module.ts**) by convention.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
```

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

AppModule is the Root module which is added by default by the Angular CLI when we set up the new project which bootstraps **AppComponent** (Root Component) that has the selector and template with it.

At the top are the import statements. The next section is where you configure the **@NgModule** decorator by stating what components and directives belong to it (**declarations**) as well as which other modules it uses (**imports**). The **providers** is an array that may contain the list of any providers (**injectable services**). The **bootstrap** property of the module identifies the Application component as a bootstrap component and it renders HTML inside the component in DOM, when Angular launches an Application.

@NgModule Decorator:

The **@NgModule** decorator identifies AppModule as an NgModule class.

The **@NgModule** takes a metadata object that tells Angular how to compile and launch the application.

The NgModule's important metadata properties are as follows:

- declarations
- imports
- exports
- providers
- entryComponents
- bootstrap
- schemas
- id
- jit

```
@NgModule({
  declarations?: (any[] | Type<any>)[]
  imports?: (any[] | Type<any> | ModuleWithProviders<{}>)[]
  exports?: (any[] | Type<any>)[]
  providers?: Provider[]
  entryComponents?: (any[] | Type<any>)[]
  bootstrap?: (any[] | Type<any>)[]
  schemas?: (any[] | SchemaMetadata)[]
  id?: string
  jit?: true
})
```

Let understand in detail about NgModule metadata properties are as follows-

Declarations - This is where components, directives, and pipes that belong to this NgModule are declared. You should add only those, which belong to this module. The services must not be declared here. They are defined in the providers array. The Component cannot belong to more than one module. If you want to use your component in multiple modules, you need to bundle that component into a separate module and import that in the module.

Imports - A list of modules and it used to import the supporting modules like **BrowserModule**, **FormsModule**, **RouterModule**, **CommonModule**, or any other custom made feature module. Look at the code snippet.

```
imports: [
  BrowserModule,
]
```

This snippet imports the **BrowserModule** as our application is executing in the browser. Every application that needs the browser, needs the BrowserModule to be imported in the application.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Exports - Using the exports array, you can define which components, directives, pipes and modules are available for any NgModule that imports this NgModule. Only those components, directives, pipes and modules declared here are visible to the other NgModules, when they import this module.

Providers - A list of dependency injection (DI) providers and it defines the set of injectable objects (services) that are available in the injector of this module.

EntryComponents - A list of components that should be compiled when this module is defined. By default, an Angular app always has at least one entry component, the root component, AppComponent.

A bootstrapped component is an entry component that Angular loads into DOM during the application launch and other components loaded dynamically into entry components.

Bootstrap - A list of components that are automatically bootstrapped when this module is bootstrapped and the listed components will be added automatically to entryComponents. The main component of this module, which needs to be loaded when the module is loaded is specified here i.e. root component (AppComponent). This is a must if you are the first module (called the root module) that is loaded when the Angular App starts. It is the responsibility of the root module to load the first view and it is done by specifying the component here. If the module is not root module, then you should keep this blank. Only use this in the root module (AppModule).

Schemas - Defines a schema that will allow any non-Angular elements and properties. Schemas are a way to define how Angular compiles templates, and if it will throw an error when it finds elements that aren't standard HTML or known components. By default, Angular throws an error when it finds an element in a template that it doesn't know, but you can change this behaviour by setting the schema to either **NO_ERRORS_SCHEMA** (to allow all elements and properties) or **CUSTOM_ELEMENTS_SCHEMA** (to allow any elements or properties with a dash (-) in their name. Dash case is the naming convention for custom elements.).

Id - This property allows you to give an NgModule a unique ID, which you can use to retrieve a module factory reference. This is a rare use case currently.

Jit - If true, this module will be skipped by the AOT compiler and so will always be compiled using JIT.

Angular offers two ways to compile your application:

- Just-in-Time (JIT), which compiles your app in the browser at runtime.
- Ahead-of-Time (AOT), which compiles your app at build time.

JIT compilation is the default when you run the ng build (build only) or ng serve (build and serve locally) CLI commands:

ng build

ng serve

For AOT compilation, include the --aot option with the ng build or ng serve command:

ng build --aot

ng serve --aot

The ng build command with the --prod meta-flag (ng build --prod) compiles with AOT by default.

If you want to turn off AOT for the production build, you can do so by setting --aot option to false as shown below.

ng build --prod --aot false

What are the advantages of splitting an angular application into multiple Angular Modules?

Well, there are several benefits of Angular Modules.

Organizing Angular Application: First of all, Modules are a great way to organise an angular application. Every feature area is present in its own feature module. All Shared pieces (like components, directives & pipes) are present in a Shared module. All Singleton services are present in a core module. As we clearly know what is present in each module, it's easier to understand, find and change code if required

Code Reuse: Modules are great way to reuse code. For example, if you have components, directives or pipes that you want to reuse, you include them in a Shared module and import it into the module where you need them rather than duplicating code. Code duplication is just plain wrong, and results in unmaintainable and error prone code.

Code Maintenance: Since Angular Modules promote code reuse and separation of concerns, they are essential for writing maintainable code in angular projects.

Performance: Another great reason to refactor your application into modules is performance. Angular modules can be loaded either eagerly when the application starts or lazily on demand when they are actually needed or in the background. Lazy loading angular modules can significantly boost the application start up time.

Types of Angular Modules (NgModules):

There are following types of Angular Modules (NgModules):

- Root Module
- Feature Module
- Routing Module
- Shared Module
- Service Module
- Widget Module

Root Module: Every Angular application has at least one module, the root module. By default, this root module is called AppModule. We bootstrap this root module to launch the application. If the application that you are building is a simple application with a few components, then all you need is the root module. As the application starts to grow and become complex, in addition to the root module, we may add several feature modules. We then import these feature modules into the root module. Root module (AppModule) as follows:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Feature Module:

Feature modules are NgModules for the purpose of organizing code. The feature modules are modules that goal of organizing an application code. Applications should be made up of multiple feature modules.

As your app grows, you can organize code relevant for a specific feature. This helps apply clear boundaries for features. With feature modules, you can keep code related to a specific functionality or feature separate from other code.

It is a class, which decorates with @NgModule decorator. Its metadata is very similar to the root module and it has the same properties as the metadata for a root module.

Both the modules share the same execution context and dependency injector. The following two are major differences between root modules and feature modules.

Root Modules	Feature Modules
Required to boot the root module to run the Application.	Required to import the feature module to extend the Application.
It is very straightforward and visible.	It can hide or expose the module implementation from the other modules.

A feature module delivers a cohesive set of functionality focused on a specific application need such as a user workflow, routing, or forms. While you can do everything within the root module, feature modules help you partition the app into focused areas.

How to make a feature module:

Assuming you already have an app that you created with the Angular CLI, create a feature module using the CLI by entering the following command in the root project directory.

ng generate module <modulename>

or

ng g m <modulename>

Example:

name.component.ts

```
import { Component } from '@angular/core';
@Component({
selector: 'app-name',
template: '<input type="text" #txtName (keyup)="0" />' +
'<br />' +
'<p><b>You Have Entered: </b>{{txtName.value}}</p>' +
})
export class NameComponent { }
```

name.module.ts

```
import { NgModule } from '@angular/core';
import { NameComponent } from './name.component';
@NgModule({
imports: [],
declarations: [ NameComponent ],
exports: [ NameComponent ],
})
export class NameModule { }
```

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title: string = 'DemoAngularApp1';
}
```

app.component.html

```
<h1>Welcome to {{title}}</h1>
<app-name></app-name>
```

app.module.ts

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { NameModule } from './name.module';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    NameModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Output:



Welcome to DemoAngularApp1

Enter Name:

You Have Entered: Angular Application

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Shared Module: The shared module allows you to organize your application code. You can put your commonly used components, directives, and pipes into the one module and use whenever required to this module.

As the name implies, the **shared module** contains all the commonly used directives, pipes, and components that we want to share with other modules that import **this shared module**.

Things to consider when creating a shared module:

- The Shared Module may re-export other common angular modules, such as CommonModule, FormsModule, and ReactiveFormsModule etc. Instead of writing the same code in every feature module to import these commonly used Angular modules we can re-export them from a Shared Module, so these commonly used Angular Modules are available to all the feature modules that import this Shared Module.
- The SharedModule should not have providers. This is because, lazy loaded modules create their own branch on the Dependency Injection tree. As a result of this, if a lazy loaded module imports the shared module, we end up with more than one instance of the service provided by the shared module. If this does not make sense at the moment. For this same reason, the Shared Module should not import or re-export modules that have providers.
- The Shared Module is then imported by all the Feature Modules where we need the shared functionality. The Shared Module can be imported by both - eager loaded Feature Modules as well as lazy loaded Feature Modules.

Example: Consider the following module:

```
import { CommonModule } from '@angular/common';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { CustomerComponent } from './customer.component';
import { NewItemDirective } from './new-item.directive';
import { OrdersPipe } from './orders.pipe';

@NgModule({
  imports:      [ CommonModule ],
  declarations: [ CustomerComponent, NewItemDirective, OrdersPipe ],
  exports:      [ CustomerComponent, NewItemDirective, OrdersPipe,
    CommonModule, FormsModule ]
})
export class SharedModule { }
```

Note the following:

- It imports the CommonModule because the module's component needs common directives.
- It declares and exports the utility pipe, directive, and component classes.
- It re-exports the CommonModule and FormsModule.

By re-exporting CommonModule and FormsModule, any other module that imports this SharedModule, gets access to directives like NgIf and NgFor from CommonModule and can bind to component properties with [(ngModel)], a directive in the FormsModule.

Even though the components declared by SharedModule might not bind with [(ngModel)] and there may be no need for SharedModule to import FormsModule, SharedModule can still export FormsModule without listing it among its imports. This way, you can give other modules access to FormsModule without having to import it directly into the @NgModule decorator.

Routing Module: The Routing is used to manage routes and also enables navigation from one view (page) to another view (page) as users perform application tasks. Here the pages that we are referring to will be in the form of components. We have already seen how to create a component. Let us now create a component and see how to use routing with it.

During the project setup, we have already included the routing module and the same is available in app.module.ts as shown below:

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

AppRoutingModule is added as shown above and included in the imports array.

File details of **app-routing.module** are given below:

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

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Here, we have to note that this file is generated by default when the routing is added during project setup. If not added, the above files have to be added manually.

So in the above file, we have imported Routes and RouterModule from @angular/router.

There is a const **routes** defined which is of type Routes. It is an array which holds all the routes we need in our project.

The const routes is given to the RouterModule as shown in @NgModule. To display the routing details to the user, we need to add <router-outlet> directive where we want the view to be displayed.

The same is added in app.component.html as shown below:

```
<h1>Welcome to {{title}}</h1>
<h1>Angular 8 Routing Demo Example</h1>
<router-outlet></router-outlet>
```

Now let us create 2 components called as **Home** and **Contact Us** and navigate between them using routing.

Component Home

First, we shall discuss about Home. Following is the syntax for Component Home:

ng g component home

```
PS D:\AngularApps\DemoAngularApp1> ng g component home
CREATE src/app/home/home.component.html (19 bytes)
CREATE src/app/home/home.component.spec.ts (614 bytes)
CREATE src/app/home/home.component.ts (261 bytes)
CREATE src/app/home/home.component.css (0 bytes)
UPDATE src/app/app.module.ts (467 bytes)
```

Component Contact Us

Following is the syntax for Component Contact Us:

ng g component contactus

```
PS D:\AngularApps\DemoAngularApp1> ng g component contactus
CREATE src/app/contactus/contactus.component.html (24 bytes)
CREATE src/app/contactus/contactus.component.spec.ts (649 bytes)
CREATE src/app/contactus/contactus.component.ts (281 bytes)
CREATE src/app/contactus/contactus.component.css (0 bytes)
UPDATE src/app/app.module.ts (561 bytes)
```

We are done with creating components home and contact us. Below are the details of the components in app.module.ts –

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { ContactusComponent } from './contactus/contactus.component';

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ContactusComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Now let us add the routes details in **app-routing.module.ts** as shown below –

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { ContactusComponent } from './contactus/contactus.component';

const routes: Routes = [
  {path:"home", component:HomeComponent},
  {path:"contactus", component:ContactusComponent}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

The routes array has the component details with path and component. The required component is imported as shown above.

Here, we need to notice that the components we need for routing are imported in **app.module.ts** and also in **app-routing.module.ts**. Let us import them in one place, i.e., in **app-routing.module.ts**.

So we will create an array of component to be used for routing and will export the array in **app-routing.module.ts** and again import it in **app.module.ts**. So we have all the components to be used for routing in **app-routing.module.ts**.

This is how we have done it **app-routing.module.ts** –

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { ContactusComponent } from './contactus/contactus.component';

const routes: Routes = [
  {path:"home", component:HomeComponent},
  {path:"contactus", component:ContactusComponent}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
export const RoutingComponent = [HomeComponent,ContactusComponent];
```

The array of components i.e., **RoutingComponent** is imported in **app.module.ts** as follows –

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

```
import { AppRoutingModule, RoutingComponent } from './app-routing.module';
import { AppComponent } from './app.component';
```

```
@NgModule({
  declarations: [
    AppComponent,
    RoutingComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

So now we are done with defining the routes. We need to display the same to the user, so let us add two buttons, Home and Contact Us in app.component.html and on click of the respective buttons, it will display the component view inside <router-outlet> directive which we have added in add.component.html.

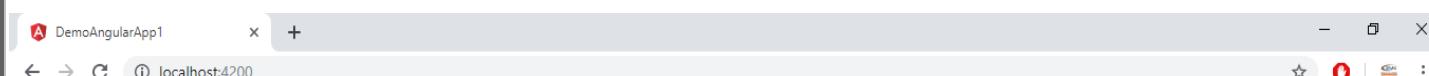
Create button inside app.component.html and give the path to the routes created.

app.component.html

```
<h1>Welcome to {{title}}</h1>
<h1>Angular 8 Routing Demo Example</h1>
<nav>
  <a routerLink="/home">Home</a> &ampnbsp
  <a routerLink="/contactus">Contact Us</a>
</nav>
<router-outlet></router-outlet>
```

In .html, we have added anchor links, Home and Contact us and used routerLink to give the path to the routes we have created in app-routing.module.ts.

Let us now test the same in the browser –



Welcome to DemoAngularApp1

Angular 8 Routing Demo Example

[Home](#)[Contact Us](#)

This is how we get it in browser. Let us add some styling to make the links look good.

We have added following css in app.component.css –

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

```
a:link, a:visited {
    background-color: #848686;
    color: white;
    padding: 10px 25px;
    text-align: center;
    text-decoration: none;
    display: inline-block;
}
a:hover, a:active {
    background-color: #BD9696;
}
```

This is the display of the links in the browser –



Welcome to DemoAngularApp1

Angular 8 Routing Demo Example



Click on Home link, to see the component details of home as shown below –



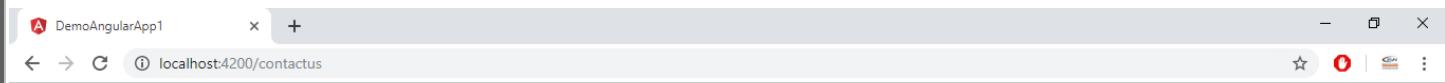
Welcome to DemoAngularApp1

Angular 8 Routing Demo Example



Welcome to Home Page!

Click on Contact Us, to see its component details as given below –



Welcome to DemoAngularApp1

Angular 8 Routing Demo Example



Welcome to Contact Us Page!

As you click on the link, you will also see the page URL in the address bar changing. It appends the path details at the end of the page as seen in the screenshot shown above.

Service Module: The modules that only contain services and providers. It provides utility services such as data access and messaging. The root AppModule is the only module that should import service modules. The HttpClientModule is a good example of a service.

Widget Module: The third party UI component libraries are widget modules.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



Data Binding in Angular:

In this we are going to look at the How Data Binding works in Angular with examples. We will take a look at various ways to bind data in Angular like **Interpolation**, **Property Bindings**, **Event Bindings** & **Two Way Bindings**. We will look at how to use the **ngModel directive** to achieve the **Two-Way Binding**. We also take a look at the **Template Expressions** and **Template Statements**, which are used in data binding.

The data binding is a powerful feature of Angular that allows us to communicate between the component and its view. Data binding is one of the finest and useful features of the Angular framework. Due to this feature, the developer needs to write less code compared to any other client-side library or framework. Data binding in an Angular application is the automatic synchronization of data between the model and view components. In Angular, we can always treat the model as a single-source-of-truth of data in our web application with the data binding. In this way, the UI or the view always represents the data model at all times. With the help of data binding, developers can establish a relation between the application UI and business logic. If we establish the data binding in a correct way, and the data provides the proper notifications to the framework, then, when the user makes any data changes in the view, the elements that are bound to the data reflect changes automatically.

Data-binding means communication between the TypeScript code of your component and your template which the user sees. Suppose, you have some business logic in your component TypeScript code to fetch some dynamic data from the server and want to display this dynamic data to the user via template because the user sees only the template. Here, we need some kind of binding between your TypeScript code and template (View). This is where data-binding comes into the picture in Angular because it is responsible for this communication.

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 to display component class property values to the user, change element styles, respond to a user event etc.

Template Expression:

Before jumping into the Data Binding in Angular, let us look at the Template Expression. In our earlier examples, we used `{{title}}` expression to display the value of the title property from the component class.

Let us look at our component class:

app.component.ts:

```
export class AppComponent {  
  title:string = 'DemoAngularApp1';  
}
```

And in our template, we refer to the title property using `{{title}}`

app.component.html:

```
<h1>Welcome to {{title}}</h1>
```

The content inside the double braces is called **Template Expression** in Angular.

The Angular first evaluates the Template Expression and converts it into a string. Then it replaces Template expression with this result in the HTML Markup.

In the example above, title is the name of the property in our component class. The Angular evaluates it and replaces it with the value of the property from the component class.

The Template Expression is much more powerful than just getting the property of the component class. You can use it to invoke any method on the component class or to do some mathematical operations etc.

The Template expression should not change the state of the application. The Angular uses it to read the values from the component and populate the view. If the Template expression changes the component values, then the rendered view would be inconsistent with the model.

Template Statement:

Template statement is similar to Template expression but can change the state of the application.

Template statements are used in case of Event Bindings. It responds to the event raised by the user like clicking on a save button (Click event) or modifying the value of textbox (Change event) etc... and invokes the method in the component class. Template statement is often a method in the Component class.

Angular evaluates the Template statement just like the Template expression. Angular updates the view after the Template statement is invoked

Difference between Template Expression and Template Statement:

Template statement can change the Application state from the user input. The Template expression should not change the application state.

Both use different Parsers.

Template expression does not support assignment (= only) operator. Template statement does.

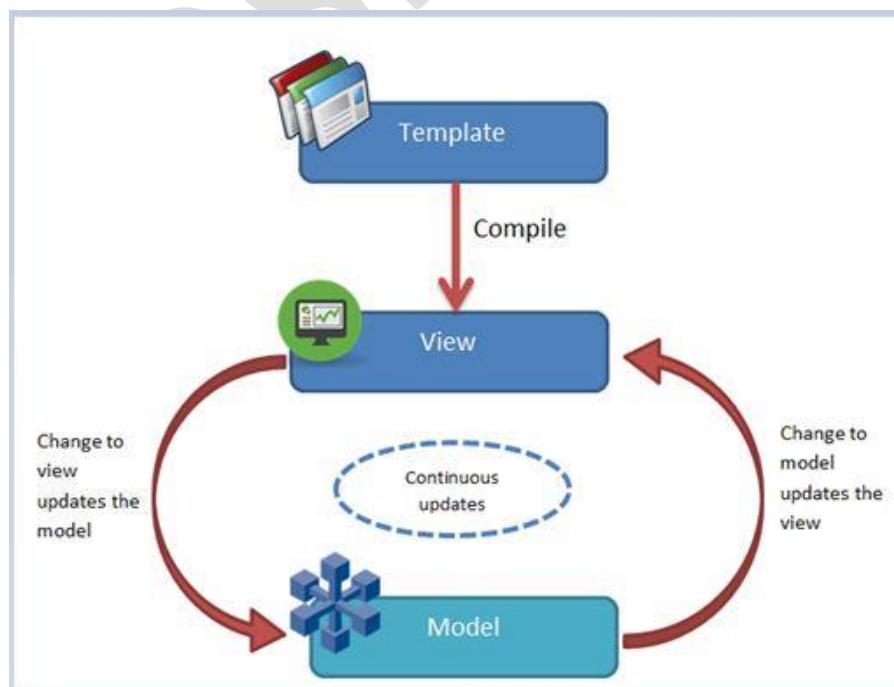
Template Expression does not support chaining of expressions. Template statement does.

Data Binding:

Basic Concept of Data Binding:

In case of any web-based application development, we always need to develop a connection bridge between the back end, where data is stored and the front end or user interface through which user performs their application data manipulation. Now, this entire process always depends on consecutive networking interactions, repetitive communication between the server (i.e. back end) and the client (i.e. front end).

Due to the consecutive and repetitive communication between client and server, most web framework platforms focus on one-way data binding. Basically, this involves reading the input from DOM, serializing the data, sending it to the back end or server, and waiting for the process to finish. After that, the process modifies the DOM to indicate if any errors occurred or reload the DOM element if the call is successful. While this process provides a traditional web application all the time it needs to perform data processing, this benefit is only really applicable to web apps with highly complex data structures. If your application has a simpler data structure format, with relatively flat models, then the extra work can needlessly complicate the process and decrease the performance of your application.



The angular framework addresses this with the data binding concept. Data binding provides a continuous data upgradation process so that when the user makes any changes in the interface that will automatically update the data and vice-versa. In this way, the data model of the application always acts as an atomic unit so that the view of the application can be always updated in respect of the data. This process can be done with the help of a complex series of event handlers and event listeners in many frameworks. But that approach can be fragile very quickly. In Angular Framework, this process related to the data becomes one of the primary parts of its architecture. Instead of creating a series of call-backs to handle the changing data, Angular does this automatically without any needed intervention by the programmer. Basically, this feature is a great relief for the programmer and reduces much more development time.

So, the first and foremost advantage of data binding is that it updates the data models automatically in respect of the view. So, when the data model updates, that will automatically update the related view element in the application. In this way, angular provides us a correct data encapsulation on the front end and it also reduced the requirement to perform complex and destructive manipulation of the DOM elements.

Why Data Binding Required?

From day one, the Angular framework provides this special and powerful feature called Data Binding which always brings smoothness and flexibility in any web application. With the help of data binding, developers can gain better control over the process and steps which are related to the data binding process. This process makes the developer's life easier and reduces development time with respect to other frameworks. Some of the reasons related to why data binding is required for any web-based application are listed below –

1. With the help of data binding, data-based web-pages can be developed quickly and efficiently.
2. We always get the desired result with a small volume of coding size.
3. Due to this process, it improves the quality of the application.
4. With the help of event emitter, we can achieve better control over the data binding process.

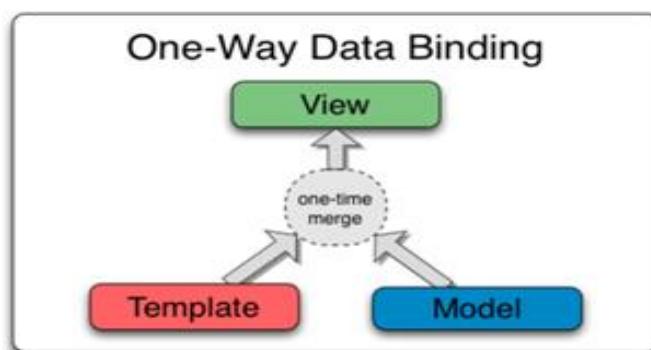
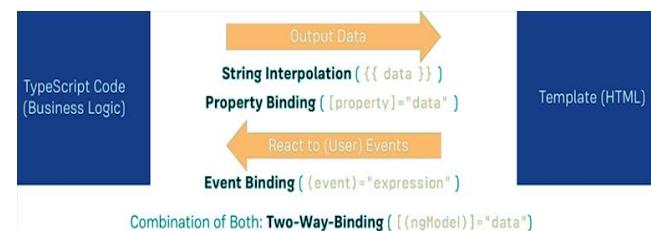
The data binding can be a **one-way data binding** [Angular Interpolation / String Interpolation, Property Binding, Event Binding] or a **two-way data binding**.

In the one-way data binding (unidirectional binding), the model value is inserted into an HTML element (DOM) and the model cannot be updated from the view. In the two-way binding (bi-directional binding), the automatic synchronization of data occurs between the Model and the View (each time the Model changes, it will be reflected in the View and vice versa)

Angular has four types of Data binding or there are four ways you can bind data in Angular

- **Interpolation/String Interpolation** (one-way data binding)
- **Property Binding** (one-way data binding)
- **Event Binding** (one-way data binding)
- **Two-Way Binding**

One – Way Data Binding:





One-way data binding will bind the data from the component to the view (DOM) or from view to the component. One-way data binding is unidirectional. You can only bind the data from component to the view or from view to the component. However it is one time merging and readonly, any modification in view will not affect the model.

One-way Data Binding: [Component to View]

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

Different types of one-way data binding

- Interpolation Binding
- Property Binding
- Attribute Binding
- Class Binding
- Style Binding

One way Data-Binding [View to Component]

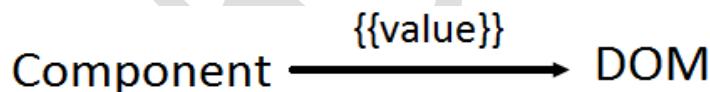
It will bind the data from View to Component using the following way.

- Event Binding

Now we will learn the interpolation (one-way data binding)

Interpolation/String Interpolation:

String Interpolation is a one-way data-binding which is used to output the data from a TypeScript code to HTML template (view). We can display all kind of properties data into view e.g. string, number, date, arrays, list or map. It uses the template expression in double curly braces to display the data from the component to the view. Interpolation moves data in one direction from our components to HTML elements.



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

app.component.html:

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

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

app.component.ts:

```
export class AppComponent {
  title:string = 'Data Binding in Angular';
}
```

Run the application, you should see only "Data Binding in Angular" in the browser

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 color etc...

Invoke a method in the component:

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

```
getTitle(): string {
    return this.title;
}
```

Now open the app.component.html and copy the following

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

Concatenate two string:

```
<p>{{ 'Hello & Welcome to' + ' Angular Data binding' }}</p>
```

```
<p>{{ firstName + ' ' + lastName }}</p>
```

Perform some mathematical operations:

```
<p>{{100 * 80}}</p>
```

```
<p>{{ num1 + num2 }}</p>
```

Bind to the Element Property:

In the following code, we bind to the color property of the paragraph element. Using interpolation we can bind to any property of the DOM element.

```
<p style.color="{{color}}>This is Red</p>
```

```

```

```
<h3>
```

```
    {{name.value}}
```

```
</h3>
```

Display array items:

We can use interpolation along with ***ngFor** directive to display an array of items.

```
export class AppComponent {
  // declared array of week days.
  days:string[] = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"];
}

<p>Week Days:</p>
<ul>
  <li *ngFor="let day of days">
    {{ day }}
  </li>
</ul>

<p>Week Days:</p>
<select>
  <option *ngFor="let day of days">{{ day }}
```

```

interface IDays{
  Day:string,
  Value:number
}
export class AppComponent {
  Days:IDays[] =
  [
    {Day:"Monday",Value:1},
    {Day:"Tuesday",Value:2},
    {Day:"Wednesday",Value:3},
    {Day:"Thursday",Value:4},
    {Day:"Friday",Value:5},
    {Day:"Saturday",Value:6},
    {Day:"Sunday",Value:7},
  ];
}
<p>Week Days:</p>
<select>
  <option *ngFor = "let d of Days" value="{{d.Value}}">{{ d.Day }}</option>
</select>

```

The pipe operator () in Interpolation:

The result of an expression might require some transformation before you're ready to use it in a binding. For example, you might display a number as a currency, change text to uppercase, or filter a list and sort it.

Pipes are simple functions that accept an input value and return a transformed value. They're easy to apply within template expressions, using the pipe operator ():

```
<p> {{title | uppercase}} </p>
```

Safe Navigation Operator (?) in Interpolation:

Safe Navigation Operator (?.) is an angular template expression operator, safe navigation operator prevents angular from throwing an exception when there is a **null** or **undefined** property.

Safe Navigation Operation “?” can also be called as “**Elvis Operator**”. Elvis Operator prevents the angular engine from complaining if you try to access values on objects that are **null** or **undefined**. The statement will simply be ignored instead of causing an error when the property you're trying to access is on a null or undefined.

Syntax

Object?.path

Example:

```

class User {
  public name: string;
  public city: string;

  constructor(name: string, city: string) {
    this.name = name;
    this.city = city;
  }
}

```

```
export class AppComponent {
  user1: User = new User('Smith', 'Hyderabad');
  user2: User;
}
```

We have created a new object for **user1** and have passed the value to its constructor and for **user2** we have just declared it as a **User** type.

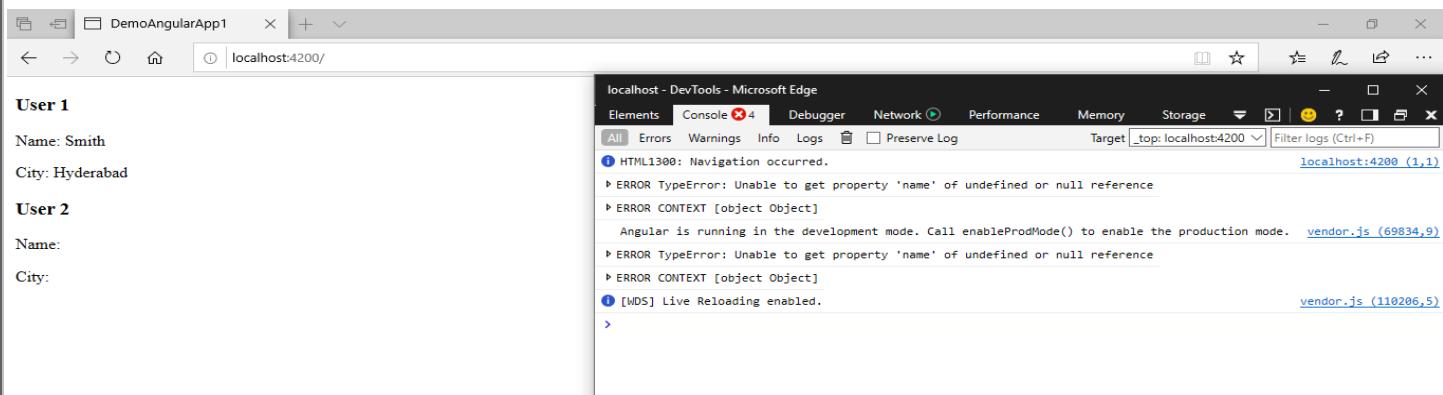
app.component.html

```
<h3>User 1</h3>
<p>Name: {{user1.name}}</p>
<p>City: {{user1.city}}</p>

<h3>User 2</h3>
<p>Name: {{user2.name}}</p>
<p>City: {{user2.city}}</p>
```

Output:

Upon running the above code we will be getting the error i.e. “**Unable to get property 'name' of undefined or null reference**”, since the **user2** is not defined.



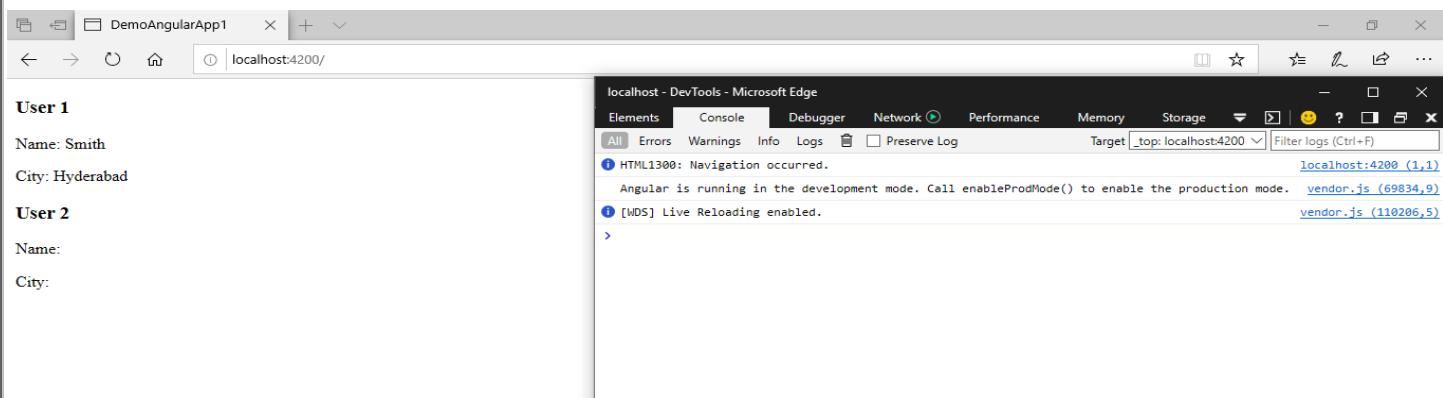
The screenshot shows a browser window with the URL `localhost:4200`. The page displays two sections: "User 1" and "User 2". Under "User 1", there are fields for "Name: Smith" and "City: Hyderabad". Under "User 2", there are empty fields for "Name:" and "City:". To the right of the browser is the Microsoft Edge DevTools console. It shows the following error message in the "Console" tab:

```
localhost - DevTools - Microsoft Edge
Elements Console Errors Warnings Info Logs Preset Log Target _top: localhost:4200 Filter logs (Ctrl+F)
localhost:4200 (1,1)
HTML1300: Navigation occurred.
ERROR TypeError: Unable to get property 'name' of undefined or null reference
ERROR CONTEXT [Object Object]
Angular is running in the development mode. Call enableProdMode() to enable the production mode. vendor.js (69834,9)
ERROR TypeError: Unable to get property 'name' of undefined or null reference
ERROR CONTEXT [Object Object]
[WDS] Live Reloading enabled.
vendor.js (110206,5)
```

We can make Angular not to throw an exception by adding the **Safe Navigation Operator** in the interpolation.

```
<h3>User 1</h3>
<p>Name: {{user1?.name}}</p>
<p>City: {{user1?.city}}</p>

<h3>User 2</h3>
<p>Name: {{user2?.name}}</p>
<p>City: {{user2?.city}}</p>
```



The screenshot shows a browser window with the URL `localhost:4200`. The page displays the same "User 1" and "User 2" sections as before. Under "User 1", the fields are filled with "Name: Smith" and "City: Hyderabad". Under "User 2", the fields are empty. To the right of the browser is the Microsoft Edge DevTools console. It shows the following messages in the "Console" tab:

```
localhost - DevTools - Microsoft Edge
Elements Console Errors Warnings Info Logs Preset Log Target _top: localhost:4200 Filter logs (Ctrl+F)
localhost:4200 (1,1)
HTML1300: Navigation occurred.
Angular is running in the development mode. Call enableProdMode() to enable the production mode. vendor.js (69834,9)
[WDS] Live Reloading enabled.
```

We can also use **ngIf** directive to achieve the same functionality

```
<h3>User 1</h3>
<p *ngIf="user1">Name: {{user1.name}}</p>
<p *ngIf="user1">City: {{user1.city}}</p>
```

```
<h3>User 2</h3>
<p *ngIf="user2">Name: {{user2.name}}</p>
<p *ngIf="user2">City: {{user2.city}}</p>
```

The other alternative is using “**&& operator**”

```
<h3>User 1</h3>
<p>Name: {{user1 && user1.name}}</p>
<p>City: {{user1 && user1.city}}</p>
```

```
<h3>User 2</h3>
<p>Name: {{user2 && user2.name}}</p>
<p>City: {{user2 && user2.city}}</p>
```

Both **ngIf** and **&& Operator** approaches are valid only but it will become complex when the property path is long. Say for example if we want to protect **null** or **undefined** for a longer path such as **obj1.obj2.obj3.obj4** in those it is not advisable to use **ngIf** or **&& operator** and it is better to go with **Safe Navigation Operator**.

Notes on Interpolation

1. Interpolation is one way from component to View
2. Binding source is a Template expression
3. Interpolation Expression must result in a string. If we return an object it will not work

Property Binding

In Angular, another binding mechanism exists, which is called Property Binding. Property Binding is also a **one-way data binding** technique. Property binding allows us to bind **Property of a view element** to the value of template expression or you can say bind a property of a DOM element to a field which is a defined property in our component TypeScript code. Actually Angular internally converts string interpolation into property binding.

For Example:

[Property] = "expression"

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

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

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>
```



Property binding is preferred over string interpolation because it has shorter and cleaner code. String interpolation should be used when you want to simply display some dynamic data from a component on the view between tags like h1, h2, p, span, and div etc...

```
<h2>{{ title }}</h2> <!-- String Interpolation -->
<img [src]="imgUrl" /> <!-- Property Binding -->
```

For Example – ``

However, we can use string interpolation here like ``, but property binding is always a lot cleaner and shorter syntax to bind image source.

In general, recommendation is when you want to simply display some dynamic data from a component on the view between tags like h1, h2, p, span, and div etc..., use string interpolation.

Suppose if you will use property binding for just displaying some value between h2 headings; the syntax will be –
`<h2 [textContent] = "title"></h2>`

Here, we are binding the text content property of h2 DOM element with the title property that we defined in our component. You can see that this syntax obviously is longer than string interpolation.

Note: String Interpolation and Property binding both are one-way binding. Means, if field value in the component changes, Angular will automatically update the DOM. But any changes in the DOM will not be reflected back in the component.

String Interpolation vs Property Binding

String Interpolation and Property binding both are used for same purpose i.e. one-way databinding. But the problem is how to know which one is best suited for your application.

Here, we compare both in the terms of Similarities, Differences and the output you receive.

Similarities b/w String Interpolation and Property Binding

String Interpolation and Property Binding doth are about one-way data binding. They both flow a value in one direction from our components to HTML elements.

String Interpolation

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: `
    <h1>{{ fullName }}</h1>
  `
})
export class AppComponent {
  fullName: string = 'John Miller';
}
```

You can see in the above example, Angular takes value of the fullName property from the component and inserts it between the opening and closing `<h1>` element using curly braces used to specify interpolation.

Property Binding

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: `
    <h1 [innerHTML]='fullName'></h1>
  `
})
export class AppComponent {
  fullName: string = 'John Miller';
}
```

In Property binding, see how Angular pulls the value from fullName property from the component and inserts it using the html property innerHTML of `<h1>` element.

Both examples for string interpolation and property binding will provide the same result.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Difference between String interpolation and Property Binding

String Interpolation is a special syntax which is converted to property binding by Angular. It's a convenient alternative to property binding.

When you need to concatenate strings, you must use interpolation instead of property binding.

Example:

```
@Component({
  selector: 'my-app',
  template: `<div>
    <h1>{{'Welcome to ' + info}}</h1>
  </div>`
})
export class AppComponent {
  info: string = 'Interpolation';
}
```

Property Binding is used when you have to set an element property to a non-string data value.

Example:

In the following example, we disable a button by binding to the Boolean property `isDisabled`.

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: `<div>
    <button [disabled]='isDisabled'>Click Me</button>
  </div>`
})
export class AppComponent {
  isDisabled: boolean = true;
}
```

If you use interpolation instead of property binding, the button will always be disabled regardless `isDisabled` class property value is true or false.

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: `<div>
    <button disabled='{{isDisabled}}'>Click Me</button>
  </div>`
})
export class AppComponent {
  isDisabled: boolean = true/false;
}
```

Note: So where you have to use string expression use interpolation and when you are dealing with non-string expression using property binding.



Attribute Binding

Attribute binding is used to bind an attribute property of a view element. Attribute binding is mainly useful where we don't have any property view with respect to an HTML element attribute.

Let's consider an example where we are trying to bind a value to the colspan property of the element.

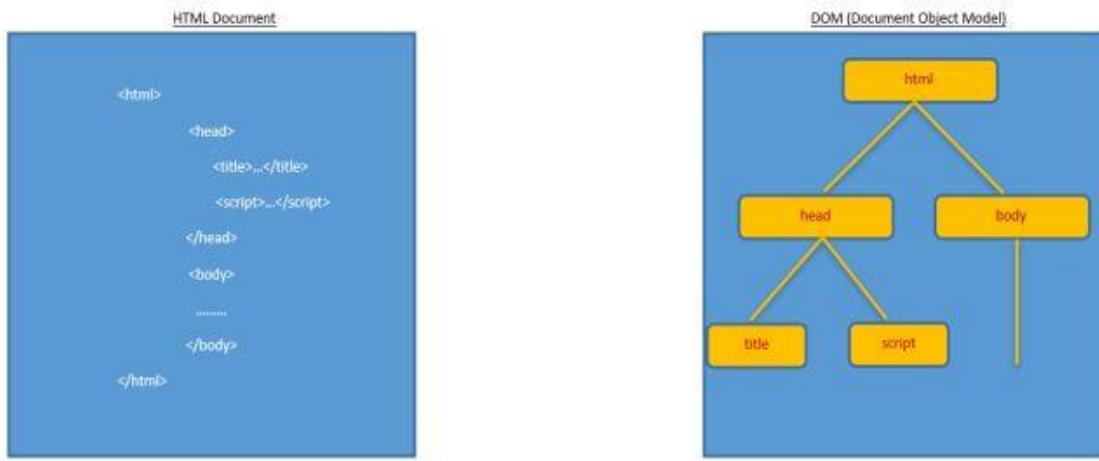
```
<h2>Attribute Binding Example</h2>
<table border="1" width="100%">
  <tr>
    <td [colspan]="col">Title</td>
  </tr>
  <tr>
    <td>1</td>
    <td>2</td>
  </tr>
</table>
```

This will throw an error in our console window i.e. "Template parse errors: Can't bind to 'colspan' since it isn't a known native property".

Difference between HTML and DOM (Document Object Model)

DOM is a model of objects that represent the structure of a document. It's indeed a hierarchical structure of HTML elements in memory.

HTML is a markup language that is used to represent the DOM in the text. It's indeed a textual representation of DOM in HTML document.



That means when your browser parses your HTML document, it creates a tree of objects in memory that is called DOM. Here is one of the most important things we need to know -- most of the attributes of HTML elements have the one-to-one mapping with the properties of DOM objects (as shown in the below picture).



Just like src, the attribute of the element has the one-to-one mapping with src property of the DOM element.

However, there are a few exceptions like we have some attributes of HTML elements that do not have representation as properties in DOM.

Differences between HTML Attribute and DOM Property

Angular attribute binding is required in those cases when HTML attribute has no corresponding DOM property. Find some differences between HTML attribute and DOM property.

- Attributes are defined by HTML and properties are defined by DOM.
- The responsibility of HTML attributes is just to initialize DOM properties. And then later DOM properties can change but HTML attributes cannot.
- There are some HTML attributes that have corresponding DOM property and those DOM properties have corresponding HTML attributes such as id.
- There are also some HTML attributes that do not have corresponding DOM property such as colspan.
- There are also some DOM properties that do not have corresponding HTML attributes such as textContent.
- The HTML attribute value contains initial value whereas DOM property value contains current value.

For Example:

```
<td [colspan]="col"></td>
```

Here, colspan is an example of the same. So when your browser parses this HTML and creates a DOM object for this `<td>` element and that DOM object does not have a property called colspan, that's why we get that error in the console.

Same here, there are some properties in the DOM that do not have a representation in HTML.

For Example:

```
<h1 [textContent]="headerValue" ></h1>
```

Here textContent is a property of DOM that does not have a representation in the HTML.

Now the conclusion is when using property binding, we should remember that we are actually binding to a property of DOM object and not an attribute of an HTML element.

We can only use property binding and interpolation for binding the properties, not attributes. We need separate attribute binding to create and bind to attributes.

Attribute binding syntax is like property binding. In property binding, we only specify the element between brackets. But in the case of attribute binding, it starts with the prefix `attr`, followed by a dot (.), and the name of the attribute. You then bind the attribute value using an expression that resolves to a string.

Let's consider an example where we are creating a table and setting the colspan attribute of the element. Here, we are setting the colspan to 3 by binding value to `attr.colspan` attribute property.

File Name: app.component.ts

```
import { Component } from "@angular/core";

@Component({
  selector: 'app-root',
  template: `
    <div>
      <table>
        <tr><td [attr.colspan]="col">Three</td></tr>
        <tr><td>1</td><td>2</td><td>3</td></tr>
      </table>
    </div>
  `
})

export class AppComponent {
  col:number = 3;
}
```

Attribute binding is used for those attributes of HTML element which do not have representation in DOM. Property binding does not work for such attributes.

1. Find the attribute binding using bracket [].

```
<td [attr.colspan] = "col"></td>
```

2. Attribute binding using bind- keyword.

```
<td bind-attr.colspan = "col"></td>
```

3. Attribute binding using interpolation.

```
<td attr.colspan = "{{col}}></td>
```

Class Binding:

We use CSS classes to give an impressive look and feel to our applications. It is very important to add or remove CSS classes at runtime to maintain high user experience in the application. It is a very common thing for any front-end developer to do so either in JavaScript or jQuery.

Let's see how to do the same in Angular?

Class binding is used to set a class property of a view element. We can add and remove the CSS class names from an element's class attribute with class binding. You can add CSS Classes conditionally to an element, hence creating a dynamically styled element.

CSS class binding is same as property binding. But we need to add a prefix **class**. with CSS class name in class binding. The CSS class added by class binding will override the existing CSS class properties if any property will be common. CSS class binding removes CSS class when expression returns false and adds it when expression returns true. To add or remove more than one CSS class dynamically we should use NgClass directive. Angular also provides DOM **className** property to add/remove classes to and from the element.

Class binding with **className**:

The **className** is the property name of HTML Element. Hence we can make use of property binding to assign the class name to any HTML element.

The following example assigns CSS Class red to the p element.

```
<style>
.red{
  color:red;
}
</style>

<p [className]="'red'">This is Red</p>
```

You can also add more than one class by separating them as following:

```
<style>
.red{
  color:red;
}
.size{
  font-size: 25px;
}
</style>

<p [className]="'red size'">This is Red</p>
```

HTML Class attribute:

You can also add class using the normal HTML way.

```
<p class="red">This is Red</p>
```

But, mixing both **class** and **[className]** results in removal of **class** attribute. You cannot use both.

```
<p class="red" [className]="'size'">This is Red</p>
```

Conditionally apply Classes:

We can also bind the class name dynamically.

To do that first create a variable in your component class.

```
className: string = 'red size';
```

And then use it in the Template as shown below:

```
<p [className]="className">This is Red</p>
```

You can create a function, which returns the class based on some condition.

```
getClass() {
  return 'red';
}
```

And then use it in the template as shown below:

```
<p [className]="getClass()">This is Red Class</p>
```

The following example uses the Conditional (Ternary) Operator.

```
hasError(error:boolean) {
  if(error==true)
    return true;
  else
    return false;
}
```

```
<style>
.red{
  color:red;
}
.green{
  color:green;
}
</style>
```

```
<p [className]="hasError(false) ? 'red' : 'green'">This is Text</p>
```

Class binding with class:

Class binding is same as property binding with the difference that we need to prefix class name with **class** followed by a dot(.). The syntax of using class binding is as follows.

```
<div [class.<className>]="condition">Class Binding Example</div>
```

Where

className is name of the class, which you want to bind to.

condition must return true or false. A return value of true adds the class and a false removes the class.

Here we are using bracket [] for class binding. We can also use **bind-** keyword with the target. Class binding also works with interpolation.

Examples:

```
export class AppComponent {
  isReq: boolean = true;
}
```

app.component.html:

```
<style>
.required{
  color: green;
  font-size: 30px;
}
.optional {
  color: red;
  background-color: cyan;
  font-family: cursive;
}
</style>

<div [class.required]="isReq">Class Binding !</div>
<div bind-class.required="isReq">Class Binding !</div>
<div class.required="{{isReq}}>Class Binding !</div>
```

In the above code snippet, we have defined **isReq** a component property which is **Boolean**. In the above class binding, we control our class dynamically. If the value of **isReq** is true then only the CSS class **required** will be applied in the **<div>** element.

In normal HTML coding CSS class is used as follows.

```
<div class="required">Class Binding !</div>
```

In the following example, the class **required** and **optional** both are added to the **div** element.

```
<div [class.required]="true" [class.optional]="true">This is Text</div>
```

Now to understand more about class binding, here creating a method in component as follows.

```
isOptional(data) {
  if (data == 'yes') {
    return true;
  }
  else {
    return false;
  }
}
```

In the above method when we pass "yes", method will return true otherwise false. Let see different scenarios now. Here we are using HTML element class attribute as well as angular class binding. We are adding two CSS classes that is **required** and **optional**.

1. The component method **isOptional('yes')** returns true, so **<div>** tag will have both the CSS classes. The CSS properties of optional class will override the CSS properties of required class if there are any common properties.

```
<div class="required" [class.optional]="isOptional('yes')">Class Binding !</div>
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

2. Here **optional** class will not include in <div> because **isOptional('no')** will return **false** value.

```
<div class="required" [class.optional]="isOptional('no')">Class Binding !</div>
```

3. Here we consider a scenario in which we are using both CSS classes using regular HTML element class attribute. Now using angular class binding we can on/off CSS classes. If **isOptional()** method returns true, then both the CSS classes will work and if **isOptional()** method returns false, then optional CSS class will be removed.

```
<div class="required optional" [class.optional]="isOptional('no')">Class Binding !</div>
```

In above code only **required** CSS class will work.

Let's see another real time example:

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title:string = 'Student List';
  students = [
    {
      name:"John",
      age:21,
      gender:"Male"
    },
    {
      name:"Alina",
      age:23,
      gender:"Female"
    },
    {
      name:"David",
      age:25,
      gender:"Male"
    },
    {
      name:"Smith",
      age:24,
      gender:"Male"
    },
    {
      name:"Peter",
      age:27,
      gender:"Male"
    }
];
}
```

app.component.html:

```

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

<style>
td,th{
  padding: 10px;
  margin: 10px;
  font-weight: bold;
  font-family: Tahoma;
}

.headingRow{
  color: white;
  background-color: black
}

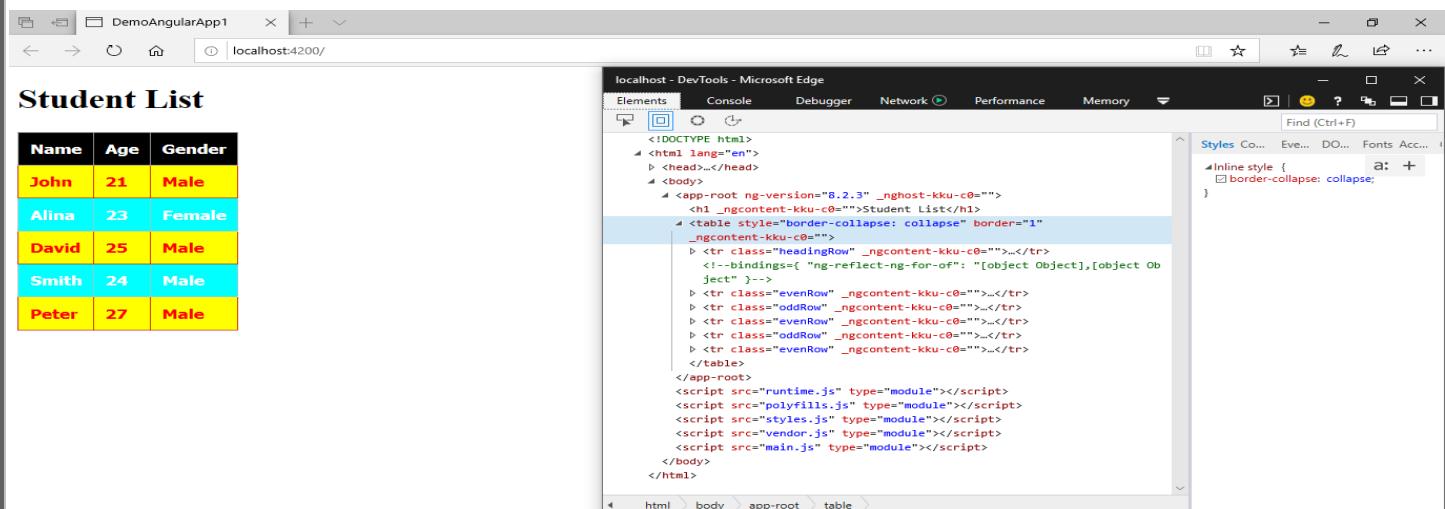
.oddRow{
  color:white;
  background-color: cyan;
}

.evenRow {
  color: red;
  background-color: yellow;
}
</style>

<table border="1" style="border-collapse: collapse">
<tr class="headingRow">
  <th>Name</th>
  <th>Age</th>
  <th>Gender</th>
</tr>
<tr *ngFor='let student of students;let i = index' class="evenRow oddRow" [class.evenRow]="i%2==0" [class.oddRow] ="i%2!=0">
  <td>{{student.name}}</td>
  <td>{{student.age}}</td>
  <td>{{student.gender}}</td>
</tr>
</table>

```

Output:



The screenshot shows a browser window with the URL localhost:4200. The page displays a table titled "Student List" with five rows of data. The table has three columns: Name, Age, and Gender. The rows alternate colors: even rows are yellow and odd rows are cyan. The DevTools Elements tab is open, showing the DOM structure of the page. The table is wrapped in an `<app-root>` element. The `<table>` tag has a `border="1"` attribute and a `style="border-collapse: collapse"` inline style. The `<tr>` elements for the header and data rows have classes like `headingRow`, `oddRow`, and `evenRow` applied via Angular's `*ngFor` directive.

Name	Age	Gender
John	21	Male
Alina	23	Female
David	25	Male
Smith	24	Male
Peter	27	Male

Class binding with NgClass:

NgClass is used to add and remove CSS classes on an HTML element. We can bind several CSS classes to NgClass simultaneously that can be added or removed. There are different ways to bind CSS classes to NgClass that are using string, array and object. CSS classes are assigned to NgClass as property binding using bracket i.e. [ngClass] as attribute of any HTML element. We can assign one or more than one CSS classes to HTML element using NgClass. When we use NgClass with object then adding and removing CSS classes are performed on the basis of returned Boolean value of an expression. If the value of expression is true then respective CSS class will be added otherwise that CSS class will be removed from HTML element at run time.

NgClass

The **ngClass** directive adds and removes CSS classes on an HTML element. The syntax of the ngClass is as shown below.

```
<element [ngClass]="expression">...</element>
```

Where

element is the DOM element to which class is being applied

expression is evaluated and the resulting classes are added/removed from the element. The expression can be in various formats like **string**, **array** or an **object**. Let us explore all of them with example

Create CSS Classes

For our example, we are creating four CSS classes as follows.

```
<style>
  .one {
    color: green;
  }
  .two {
    font-size: 20px;
  }
  .three {
    color: red;
  }
  .four {
    font-size: 15px;
  }
</style>
```

NgClass with String

We have two CSS classes named as **.one** and **.two**. Now to bind CSS class to NgClass, just create a string and inside it we need to refer our CSS class names enclosed by single quote ('). Find the code below. Here we are binding a single CSS class to NgClass.

```
<p [ngClass]="'one'">
  Using NgClass with String.
</p>
```

In case we want to add more than one CSS classes then add it separated by space.

```
<p [ngClass]="'one two'">
  Using NgClass with String.
</p>
```

We can also use ngClass with ngFor using string as follows. The given CSS classes will be applied to HTML element in each iteration.

app.component.ts:

```
export class AppComponent {
  cities:string[] = ["Hyderabad", "Banglore", "Chennai", "Pune", "Mumbai"];
}
```

app.component.html:

```
<ul>
  <li *ngFor="let city of cities" [ngClass]="'one two'">
    {{city}}
  </li>
</ul>
```

NgClass with Array

Here we will discuss how to use NgClass with array of CSS classes. Suppose we have two CSS classes named as **.one** and **.two**. We need to add our CSS classes within bracket [] separated by comma (,) and enclosed by single quotes ('').

```
<p [ngClass]=["'one', 'two']">
  Using NgClass with Array.
</p>
```

ngClass with array can also be used with **ngFor** as follows:

```
<ul>
  <li *ngFor="let city of cities" [ngClass]=["'one', 'two']">
    {{city}}
  </li>
</ul>
```

NgClass with Object

Here we will discuss NgClass with object using braces { }. It will be in key and value pair format separated by colon (:).
key: CSS class name.

value: An expression that returns Boolean value.

Here key will be CSS class name and value will be an expression that will return Boolean value. CSS will be added at run time in HTML element only when if expression will return true. If expression returns false then the respective CSS class will not be added. Suppose we have four CSS classes named as **.one**, **.two**, **.three** and **.four**. Now find the below code snippet.

```
<p [ngClass]={'one': true, 'three': false }">
  Using NgClass with Object.
</p>
```

In the above code snippet we are using two CSS classes. The expression value of CSS class **.one** is true. So this CSS class will be added at runtime. Now check next CSS class used in above code snippet that is **.three** and its expression value is false. So this CSS class will be removed at run time from HTML element.

ngClass with object can also be used with ngFor. Find the code snippet below. Here we are using index variable and dividing it by 2 in every iteration. When the result is 0, we are using CSS class as **.one** and if result is 1 then we are using CSS class as **.three**.

```
<ul>
  <li *ngFor="let city of cities; let i = index" [ngClass]="{'one': i%2==0, 'three':i%2==1}">
    {{city}}
  </li>
</ul>
```

Now find the below code snippet. Here we are using **even** variable. This variable returns **true** if iteration number is even otherwise **false**.

```
<ul>
  <li *ngFor="let city of cities; let flag = even;">
    <p [ngClass]="{{'one':flag, 'two':!flag, 'three':!flag, 'four':!flag}}>{{city}}</p>
  </li>
</ul>
```

In the above code snippet, while iteration if **flag** value is **true** then CSS classes **.one** and **.two** will be added to the HTML element otherwise CSS classes **.three** and **.four** will be added to HTML element at run time.

NgClass with Component Method:

Here we will use a component method with NgClass. We will create a component method that will return a set of CSS classes. We can dynamically add or remove CSS classes from our set within the method. Find the component method used in our example.

```
export class AppComponent {
  getCSSClasses(flag:string) {
    let cssClasses;
    if(flag == 'nightMode') {
      cssClasses = {
        'one': true,
        'two': true
      }
    } else {
      cssClasses = {
        'two': true,
        'four': false
      }
    }
    return cssClasses;
  }
}
```

We will observe that we can create a set of classes that will returned by method. In this way in one call of method we can assign more than one CSS classes to **NgClass**. We can easily add or remove CSS classes in our set dynamically within component method. We will call our component method with **NgClass** as below.

```
<div [ngClass]="getCSSClasses('nightMode')">
  Using NgClass with Component Method.
</div>
```

In the above code we are calling **getCSSClasses()** method that will return a set of CSS classes to **NgClass** on the basis of specified argument.

Dynamically updating Class names:

We can dynamically change the CSS Classes from the component.

Using strings

To do that first create a string variable **classNames** in your component code and assign the class names to it as shown below.

```
classNames: string= 'one two';
```

You can refer to the **classNames** in your template as shown below:

```
<div [ngClass]="classNames">
    Welcome to ngClass !!!
</div>
```

Using arrays

Instead of string variable, you can create an **array of string** as shown below:

```
classNames: string[] = ['one', 'two'];
```

And, then use it in **ngClass** directive as shown below:

```
<div [ngClass]="classNames">
    Welcome to ngClass !!!
</div>
```

Using JavaScript object:

Create a class as shown below in your component

```
class CssClass {
  one: boolean= true;
  two: boolean= true;
}
```

Next, create the instance of the **CssClass** in the component as shown below. You can change the value of the property true as false dynamically.

```
cssClass: CssClass = new CssClass();
```

And then refer to the **cssClass** in your template.

```
<div [ngClass]="cssClass">
    Welcome to ngClass !!!!!
</div>
```

Style Binding:

Style binding is used to set a style of a view element. We can set the inline styles of a HTML element using the style binding in angular.

Like with attribute and class binding, style binding syntax is like property binding. In property binding, we only specify the element between brackets. But in case of style binding, it starts with the prefix **style**, followed by a **dot (.)** and the **name of the style**. You then bind the style value with CSS style name like the **style.style-name**.

Syntax for style binding:

```
[style.style-property] = "style-value"
```

Here `style` is a prefix and `style-property` is a name of a CSS style property. `style` prefix and style property are concatenated using dot (`.`). Style property binding can be achieved with bracket `[]`, `bind-` keyword and interpolation `{{}}`. Now find the code to use style binding.

For Example, to set the **color** of **p** element.

```
<p [style.color]="'red'">This is Red</p>
<p bind-style.color="'red'">This is Red</p>
<p style.color="{{'red'}}">This is Red</p>
```

Setting the background color of a paragraph:

```
<p [style.background-color]="'grey'">Some paragraph with grey background color</p>
<p bind-style.background-color="'grey'">Some paragraph with grey background color</p>
<p style.background-color="{{'grey'}}">Some paragraph with grey background color</p>
```

Setting the border style of a button.

```
<button [style.border]="'5px solid green'">Save</button>
 
<button bind-style.border="'5px solid red'">Update</button>
 
<button style.border="{{'5px solid blue'}}">Cancel</button>
```

Another way is to create the `getColor()` method in component and use it in the template as shown below.

app.component.ts:

```
export class AppComponent {
    getColor() {
        return 'aqua';
    }
}
```

app.component.html:

```
<button [style.background-color]= "getColor()">Button 1</button>
<button bind-style.background-color= "getColor()">Button 2</button>
<button style.background-color= "{{getColor()}}">Button 3</button>
```

Conditionally setting the styles:

app.component.ts:

```
export class AppComponent {
    result: number = 50;
}
```

app.component.html:

```
<p [style.color] = "result > 30 ? 'blue' : 'green'"> Hello Color World! </p>
<p bind-style.color = "result > 30 ? 'blue' : 'green'"> Hello Color World! </p>
<p style.color = "{{result > 30 ? 'blue' : 'green'}}"> Hello Color World! </p>
```

In our style binding example we are using conditional operator. When the condition `result > 30` returns true then color style will be set to **blue** otherwise **green**. Let us know about conditional operator.

Conditional (ternary) Operator

Conditional operator that is also called ternary operator, is used as a short cut of if else statement. Find the syntax as below.

```
condition ? expr1 : expr2
```

When the **condition** is true then **expr1** will be returned otherwise **expr2** will be returned.

Setting the units

The styles like **font-size**, **width** etc... have unit extension. The following example conditionally sets the **font-size** in "px" unit:

```
<button [style.fontSize.px]="'20'">Big Button</button>
<button bind-style.fontSize.px="'20'">Big Button</button>
<button style.fontSize.px="{{'20'}}">Big Button</button>
```

The style property name can be written in either **dash-case (font-size)**, as shown in above example, or **camelCase (fontSize)** as shown below.

```
<button [style.fontSize.px]="'20'">Big Button</button>
<button bind-style.fontSize.px="'20'">Big Button</button>
<button style.fontSize.px="{{'20'}}">Big Button</button>
```

Setting multiple styles

To change the multiple styles, we need to add each one separately as shown below.

app.component.ts:

```
export class AppComponent {
  status:string = "success";

  getColor() {
    return 'aqua';
  }
}
```

app.component.html:

```
<p [style.color]="getColor()"
  [style.fontSize.px]="'20'"
  [style.backgroundColor]="status=='error' ? 'red': 'blue'">
  Paragraph with multiple styles
</p>
```

Note: The style binding is the easy way to set a single style of a HTML element. Although you can use it to set several inline styles as shown in the above example, but the better way is to use the **ngStyle** directive for multiple inline styles.

Style binding with ngStyle:

NgStyle directive is used to set many inline styles dynamically. Setting styles using **NgStyle** works as **key:value** pair. **key** is the style property name and **value** is the style value.

ngStyle Syntax

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

Where

element is the DOM element to which style is being applied

styleNames are style names (ex: 'font-size', 'color' etc). with an optional suffix (ex: 'top.px', 'font-style.em'),

styleExp is the expression, which is evaluated and assigned to the **styleNames**

We can add more than one key value pairs **'styleNames': styleExp** each separated by comma.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

In the following example, `p` element gets the style of `font-size` of `20px`.

```
<p [ngStyle] = "{'font-size': '20px'}">Paragraph with Font size to 20px</p>
```

The units (for example `px`, `em`) are prefixed to the `styleName`.

Syntax:

```
<element [ngStyle] = "{'styleName.unit': expressionValue}" ...></element>
```

Example:

```
<p [ngStyle] = "{'font-size.em': '3'}">Paragraph with Font size to 3 em</p>
```

NgStyle with Background Image:

To load background image the style property `background-image` is used. Its value is given in the following syntax.

```
url('assets/Images/Angular.gif')
```

In inline style binding we can use it as follows.

```
<div [style.background-image] = "url(\\"assets/Images/Angular.jpg\\")">  
    Background Image Style  
</div>
```

And while using with `NgStyle` the **key:value** pair is created as following.

```
<div [ngStyle] = "{'background-image': 'url(\\"assets/Images/Angular.jpg\\")'}">  
    Background Image Style  
</div>
```

Note: We need to use backslash (\) in background image url otherwise it will throw error.

ngStyle multiple attributes:

We can change multiple style as shown in the following example

```
<p [ngStyle] = "{'color': 'purple',  
    'font-size': '20px',  
    'font-weight': 'bold'}">  
    Multiple Styles  
</p>
```

The JavaScript object is assigned to the `ngStyle` directive containing multiple properties. Each property name of the object acts as a style name. The value of the property is the value of the style.

Using object from Component:

app.component.ts:

```
export class AppComponent {  
    styles: Styles = new Styles();  
}
```

```
class Styles {  
    'color': string = 'blue';  
    'font-size.px': number = 20;  
    'font-weight': string = 'bold';  
}
```

app.component.html:

```
<div [ngStyle] = "styles">Div Element !!!</div>
```

Real Time Example:

app.component.ts:

```

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title:string = 'DemoAngularApp1';
  result:number = 50;
  colorFlag:boolean = false;
  isSmall:boolean = true;
  isBackgroundRed:boolean = false;
  small:number = 10;
  big:number = 15;
  num:number = 10;

  isRed(num:number):boolean {
    if (num > 10) {
      return false;
    } else {
      return true;
    }
  }

  allRequiredStyles(styleSet:string) {
    let myStyles={};
    if(styleSet == 'one') {
      myStyles = {
        'color': this.colorFlag ? 'black' : 'yellow',
        'font-size.em': this.isSmall ? this.small/5 : this.big/5,
        'background-
image': !this.isBackgroundRed ? 'url(\`assets/Images/red.gif\')' : 'url(\`assets/Images/green.gif\`)
"';
      };
    }
    else if(styleSet == 'two') {
      myStyles = {
        'color': !this.colorFlag ? 'black' : 'yellow',
        'font-size.em': !this.isSmall ? this.small/5 : this.big/5,
        'background-
image': this.isBackgroundRed ? 'url(\`assets/Images/red.gif\')' : 'url(\`assets/Images/green.gif\`)
"';
      };
    }
    else {
      myStyles = {
        'background-color': this.colorFlag ? 'cyan' : 'grey',
        'font-size.%': !this.isSmall ? this.small * 10: this.big * 10
      };
    }
    return myStyles;
  }
}

```

app.component.html:

```

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

<p [style.color] = "result > 30 ? 'blue' : 'green'">Hello Color World!</p>
<p bind-style.color = "result > 30 ? 'blue' : 'green'">Hello Color World!</p>
<p style.color = "{{result > 30 ? 'blue' : 'green'}}">Hello Color World!</p>

<button
  [style.background-color] = "colorFlag ? 'cyan' : 'grey'"
  [style.color] = "isRed(8) ? 'red' : 'black'">
  Button
</button>

<div [style.fontSize.em] = "isSmall ? small/8 : big/8">Font Size Test with em</div>
<div [style.fontSize.px] = "!isSmall ? small : big">Font Size Test with px</div>
<div [style.fontSize.pt] = "isSmall ? small : big">Font Size Test with pt</div>
<div [style.fontSize.%] = "!isSmall ? small * 10: big * 10">Font Size Test with %</div>

<div [style.background-
image] = "isBackgroundRed ? 'url(\`assets/Images/red.gif\')' : 'url(\`assets/Images/green.gif\`)'>
  Style Binding Example
</div>

<br/><br/>

<div [ngStyle]="allRequiredStyles('one')">
  NgStyle Example
</div>

<br/><br/>

<div bind-ngStyle="allRequiredStyles('two')">
  NgStyle Example
</div>

```

Output:



NgStyle Example

NgStyle Example

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Event Binding:

Event binding is another of the data binding techniques available in Angular. The Interpolation, Property binding, Attribute binding, Class binding and Style binding are one-way bindings from component to view whereas Event binding is a data binding from an element (view) to a component.

Event Binding is used to perform an action in the component when the user does action in the view. Data obtained from user actions such as keystrokes, mouse movements, clicks, and touches can be bound to component property using event binding. In other words, when a user interacts with an application in the form of a keyboard movement, a mouse click, or a mouse over, it generates an event. These events need to be handled to perform some kind of action. This is where event binding comes into picture.

In the event binding, target will be an event name. Target event is enclosed within parenthesis () or can be used with canonical form using prefix **on-** to perform event binding. Target event is written to the left side of equal sign. Target events can be click, change, mouseover, mouseout, keydown, keyup etc. In angular framework we can alias our event name as well as we can create custom events. To access current changes by user, we use event object **\$event**. It will be DOM event object if target event is a native DOM element event. It has properties as **target** and **target.value**. Event binding using parenthesis () is achieved as follows.

```
<button (click)="isValid=true">True</button>
```

And event binding using **on-** keyword is achieved as follows.

```
<button on-click="isValid=true">True</button>
```

Call component method on event binding.

```
<button (click)="ClickMe()">Click Me</button>
```

```
<input (change)="changeText($event.target.value)" />
```

Input Event Binding

input event is a DOM event which is fired synchronously when the value of **<input>** or **<textarea>** element is changed. Here we will bind the component property with **input** event. We will use event object **\$event** to get the current value entered by user. For the example find a component property.

```
message:string = 'Hello World';
```

For the initial value of our **<input>** element we are using property binding and assigning component property value. For event binding we need to enclose event name by parenthesis as **(input)** or by using **on-** keyword as **on-input**. Now find input event binding using () .

```
<input [value]="message" (input)="message=$event.target.value" />
```

We can also use **on-** keyword as follows:

```
<input [value]="message" on-input="message=$event.target.value" />
```

Achieve same goal by calling a component method using event binding. Create a method in component.

```
setMessage(data:string) {
  this.message = data;
}
```

Use **input** event to call the method

```
<input [value]="message" (input)="setMessage($event.target.value)" />
```

Here we are passing the current input data entered by user. If we print the message as:

```
{{message}}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



Then we will observe that whenever we write any text in `<input>` element, at the same time value of `message` will also be changed. Here `$event` gives the current value entered by user. We access it by using `$event.target.value`.

In our code we are assigning its value to `message` as follows:

```
message=$event.target.value
```

Click Event Binding:

Here we will understand event binding using `click` event. For the demo we are creating two buttons. We have initialized a component property as follows:

```
isValid:boolean = true;
```

Now find the event binding using `<button>` element as an example.

```
<button (click)="isValid=true">True</button>
<button (click)="isValid=false">False</button>
```

When we click **True** button, the component property `isValid` will be assigned with value `true` and when we click **False** button then `isValid` will be assigned with value `false`. If we are using `isValid` property in our code anywhere else, accordingly there will be change in DOM. For the example we are using it with `ngIf` as follows.

```
<p *ngIf="isValid">
    Data is valid.
</p>
<p *ngIf="!isValid">
    Data is not valid.
</p>
```

On the click of **True** button, the first `ngIf` will run and on the click of **False** button second `ngIf` will run.

KeyDown and KeyUp Event Binding:

Here we will discuss event binding using `keydown` and `keyup` event using `<input>` element. For the example we are using parenthesis () as well as `on-` keyword.

```
<input on-keydown="message='Key Down'" (keyup)="message='Key Up'" />
```

Here `message` is a component property and if we print it as follows:

```
{{message}}
```

Then on key down we will get message **Key Down** and on key up we will get message **Key Up**.

Change Event Binding:

Here we will discuss `change` event binding using `<input>` element. On value change we will call a method which we have defined in our component as follows.

```
changeText(mytext:string) {
    this.message = mytext;
}
```

Now find the `<input>` element which is using `change` event binding.

Using parenthesis () .

```
<input (change)="changeText($event.target.value)" />
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Using **on-** keyword.

```
<input on-change="changeText($event.target.value)" />
```

For the example we are printing data using **innerHTML** in **<p>** element as follows.

```
<p [innerHTML]="message" > </p>
```

Now use interpolation.

```
<p> {{message}} </p>
```

When user changes text value then after value change completion, **changeText()** method will be called and current user input value will be passed to the method that will be assigned to a component property **message**. And hence we will observe value change within **<p>** element for every value change in **<input>** element.

Change Event Binding with Dropdown List:

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title:string = 'DemoAngularApp1';

  cities:string[] = ["Select", "Hyderabad", "Banglore", "Mumbai"];

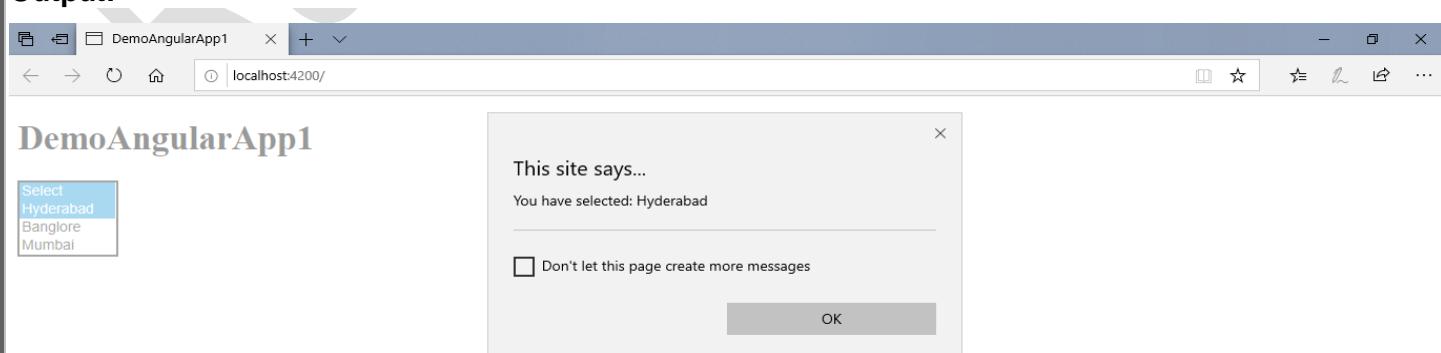
  getValue(city)
  {
    if(city!="Select")
      alert("You have selected: " + city);
    else
      alert("Please select a city");
  }
}
```

app.component.html:

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

<b>Select City: </b>
<select (change)="getValue($event.target.value)">
  <option *ngFor="let city of cities">{{city}}</option>
</select>
```

Output:



Printing Selected City on a page instead of displaying in an alert box:

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title:string = 'DemoAngularApp1';

  cities:string[] = ["Select", "Hyderabad", "Banglore", "Mumbai"];

  selectedCity:string="";

  getValue(city)
  {
    if(city!="Select")
      this.selectedCity = "You have selected: " + city;
    else
      this.selectedCity = "Please select a city";
  }
}
```

app.component.html:

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

<b>Select City: </b>
<select (change)="getValue($event.target.value)">
  <option *ngFor="let city of cities">{{city}}</option>
</select>

<p>{{selectedCity}}</p>
```

Output:



DemoAngularApp1

Select City:

You have selected: Hyderabad



DemoAngularApp1

Select City:

Please select a city

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Printing Selected City on a page with style:

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title:string = 'DemoAngularApp1';
  cities:string[] = ["Select", "Hyderabad", "Bangalore", "Mumbai"];
  selectedCity:string="";
  isSelectedCity:boolean=false;

  getValue(city)
  {
    if(city!="Select"){
      this.selectedCity = city;
      this.isSelectedCity=true;
    }
    else{
      this.isSelectedCity=false;
      this.selectedCity = "Please select a city";
    }
  }
}
```

app.component.html:

```
<h1>{{title}}</h1>
<b>Select City: </b>
<select (change)="getValue($event.target.value)">
  <option *ngFor="let city of cities">{{city}}</option>
</select>
<br />
<div>
  <span *ngIf="isSelectedCity; then condition1 else condition2">
  </span>
  <ng-template #condition1><p style="color: green;">{{"You have selected: " + selectedCity}}</p></ng-template>
  <ng-template #condition2><p style="color: red;">{{selectedCity}}</p></ng-template>
</div>
```

Output:



DemoAngularApp1

Select City:

You have selected: Hyderabad



DemoAngularApp1

Select City:

Please select a city

Event Bubbling:

All the DOM events bubble up to the DOM tree unless a handler prevents further bubbling. This is just standard event propagation mechanism in DOM. Let's understand this with the help of an example.

Let's proceed with the button example. We have a div wrapper around the button in component HTML and div has also a click event handler just to log some message if div is clicked.

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title:string = 'DemoAngularApp1';

  ButtonClick(){
    console.log("Button is Clicked!!!");
  }
  DivClick(){
    console.log("Div is Clicked!!!");
  }
}
```

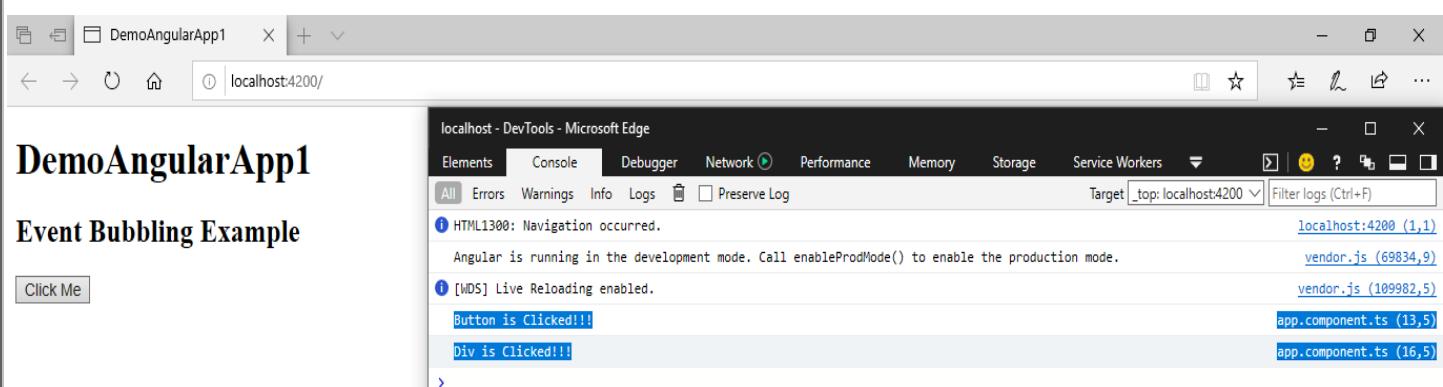
app.component.html:

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

<h2>Event Bubbling Example</h2>

<!-- Event Bubbling -->
<div (click)="DivClick()">
  <button (click)="ButtonClick()">Click Me</button> <!-- Event Binding -->
</div>
```

Output:



The screenshot shows a Microsoft Edge browser window with the URL localhost:4200. The page content is "DemoAngularApp1" with a heading "Event Bubbling Example" and a button labeled "Click Me". Below the browser is the Microsoft Edge DevTools console. The logs show the following entries:

- HTML1300: Navigation occurred.
- Angular is running in the development mode. Call enableProdMode() to enable the production mode.
- [WDS] Live Reloading enabled.
- Button is Clicked!!!
- Div is Clicked!!!

Now let's see the output. When we click the button, there are two messages logged in the console. One is from the handler of the click event of the button and second is from the handler of the click event of the div.

This is what we call "Event Bubbling", an event bubbles up the event of its parent element.

If we would have another div or another element as a wrapper around of the div element and we handle the click event of that element, our button event will bubble up against the events of entire DOM tree.

To stop this event bubbling, we use **stopPropagation** method.

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title:string = 'DemoAngularApp1';

  ButtonClick($event){
    $event.stopPropagation(); // Stop Event Bubbling
    console.log("Button is Clicked!!!");
  }
  DivClick(){
    console.log("Div is Clicked!!!");
  }
}
```

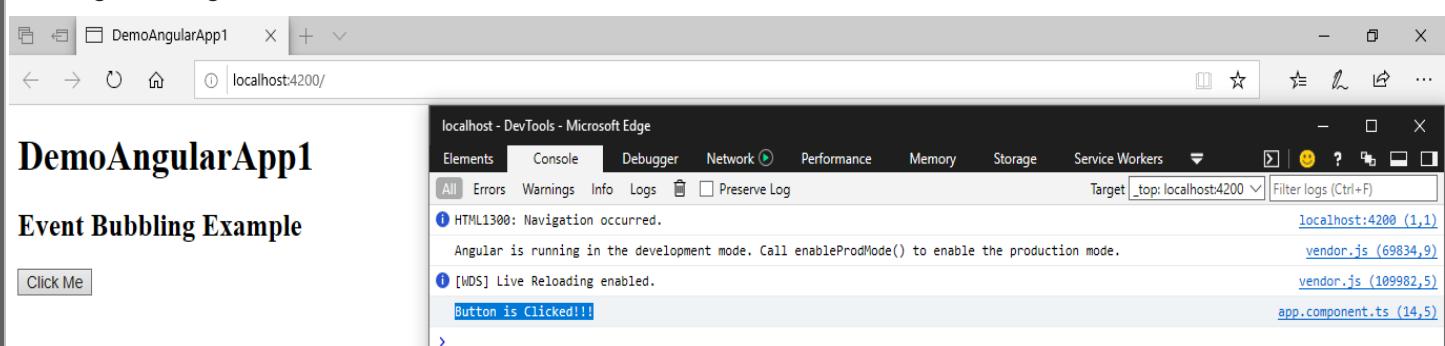
app.component.html:

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

<h2>No Event Bubbling Example</h2>

<div (click)="DivClick()">
  <button (click)="ButtonClick($event)">Click Me</button> <!-- Event Binding -->
</div>
```

And now, when we click on the 'Click Me' button, only the **ButtonClick** method will be executed and will see only a message coming from the handle of the click event of the button.



The screenshot shows a Microsoft Edge browser window with the title "DemoAngularApp1". Below it, a Microsoft Edge DevTools window titled "localhost - DevTools - Microsoft Edge" is open. The DevTools Console tab shows the following log entries:

- HTML1300: Navigation occurred.
- Angular is running in the development mode. Call enableProdMode() to enable the production mode.
- [WDS] Live Reloading enabled.
- Button is Clicked!!!

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Event Filtering:

Angular provides a feature called Event Filtering. Let's understand this with the help of an example. Suppose we have a textbox and we want to log the text (which is entered in the textbox) in the console on pressing enter. Below is a traditional way to implement it.

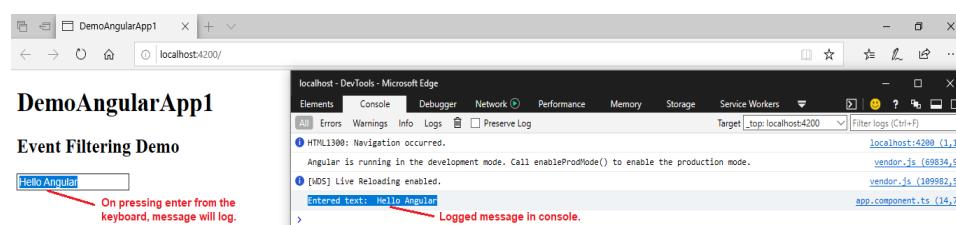
app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title:string = 'DemoAngularApp1';

  onPressEnter($event){
    if($event.keyCode==13){
      console.log("Entered Text: ",$event.target.value);
    }
  }
}

app.component.html:
<h1>{{title}}</h1>
<h2>Event Filtering Example</h2>
<input (keyup)="onPressEnter($event)" />
```

Output:



Angular provides a shorter and cleaner way to implement the same. Let's implement it with Angular event filtering.
app.component.html:

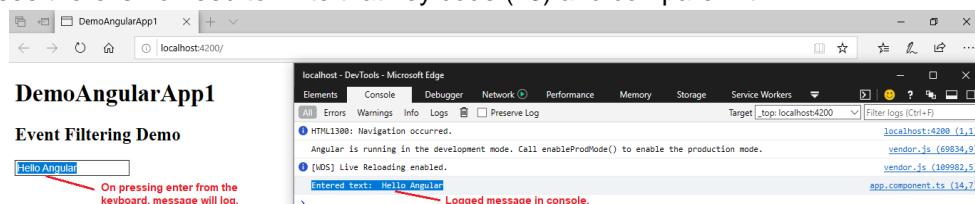
<h2>Event Filtering

```
<input (keyup.enter)="onPressEnter($event)" />
```

App.component.ts:

```
export class AppComponent {
  title:string = 'DemoAngularApp1';
  onPressEnter($event){
    console.log("Entered text: ",$event.target.value);
  }
}
```

And now, you can see there is no need to write that key code (13) and compare with.





Two Way Data Binding:

In Angular Framework, the most used and important data binding techniques are known as Two-Way Data Binding.

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.

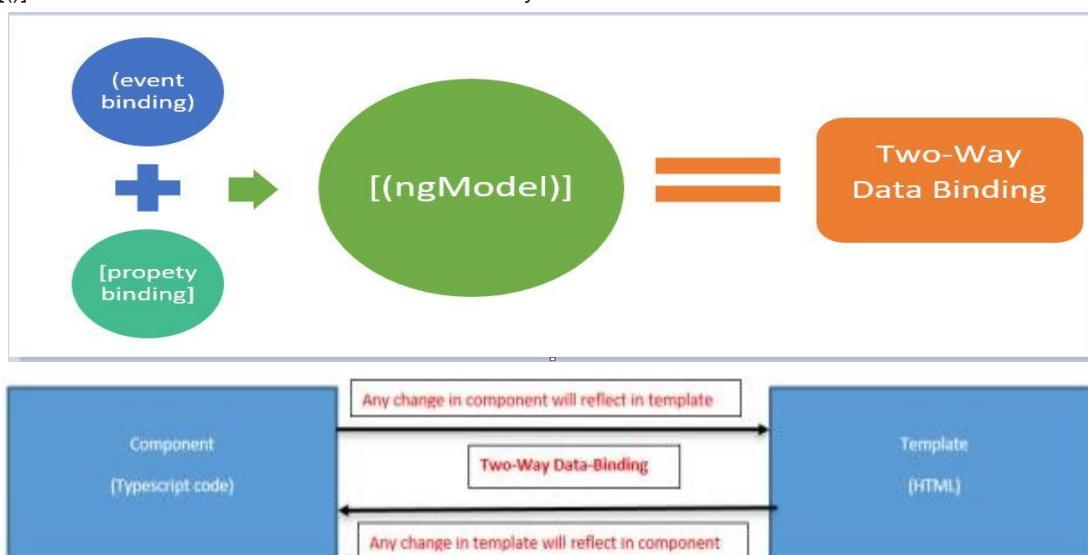
Two-way binding is mainly used in the input type field or any form element where the user can provide input values from the browser or provides any value or changes any control value through the browser. On the other side, the same is automatically updated into the component variables and vice versa.

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.

To Specify **Two Way Data Binding**, We need to use the **ngModel** directives as shown below:

<input [(ngModel)]='name'></input>

We use **[]** since it is actually a property binding, and parentheses **()** are used for the event binding concept i.e. the notation of two-way data binding is **[()]**. This is now known as **"Banana in the Box"** syntax.



ngModel performs both **property binding** and **event binding**. Actually, the **property binding** of the **ngModel** (i.e. **[(ngModel)]**) performs the activity to update the input element with a value. Whereas **(ngModel)** (**((ngModelChange)** event) instructs the outside world when any change occurred in the DOM Element.

The below example demonstrates the implementation of two-way binding. In this example, we define a string variable called **firstName** and assign that variable with a Textbox control. So, whenever we change any content in the textbox, the value of the variable will be changed automatically.

The **ngModel** directive is not part of the **Angular Core library**. It is part of the **FormsModule library**. You need to import the **FormsModule package** into your **Angular module** i.e. **app.module.ts** as follows:

app.module.ts:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule, AppRoutingModule, FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Open the app.component.ts and add the firstName variable

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title:string = 'DemoAngularApp1';

  firstName:string = "Angular";
}
```

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

app.component.html:

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

<div>
  <b>Enter First Name: </b>
  <input [(ngModel)]="firstName" type="text"/>
</div>

<p><b>Your First Name: </b>{{firstName}}</p>
```

Run the project and see that as you modify the first name, the component class model (firstName property) is automatically updated and displayed back to view. ngModel will trigger the input event and update the value of input textbox in our view and when we will change the value of the textbox, it will automatically update in the "firstName" property of our component and vice-versa (as shown with the help of including string interpolation inside a paragraph).

Before Changing the First Name:



DemoAngularApp1

Enter First Name: Angular

Your First Name: Angular

After Changing the First Name:



DemoAngularApp1

Enter First Name: RakeshSoftNet

Your First Name: RakeshSoftNet

Two-Way Data Binding Without ngModel:

To understand ngModel directive working, let's see how we can achieve two-way data binding without using ngModel directive. To do that, we need to use:

- **Property binding** to bind the expression to the value property of the input element. In this example, we are binding firstName variable expression to value property.
- **Event binding** to emit an input event on the input element. Yes, there is an input event that will be fired whenever the user inputs the input element. Using event binding, the input event would be bound to an expression.

Using property binding and the event binding, two-way data binding can be achieved as shown in the listing below:

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title:string = 'DemoAngularApp1';

  firstName:string = "Angular";
}
```

app.component.html:

```
<h1>{{title}}</h1>
<div>
  <b>Enter First Name: </b>
  <input [value]="firstName" (input)="firstName=$event.target.value" />
</div>
<p><b>Your First Name: </b>{{firstName}}</p>
```

Output:

Before Changing the First Name:



DemoAngularApp1

Enter First Name: Angular

Your First Name: Angular

After Changing the First Name:



DemoAngularApp1

Enter First Name: RakeshSoftNet

Your First Name: RakeshSoftNet

Hyderabad's Best Institute for .NET

Like the ngModel directive example in this example, when typing into the input element, the input element's value will be assigned to firstName variable and displayed back to the view. We are implementing two-way data binding without using ngModel using the code shown in the below image:

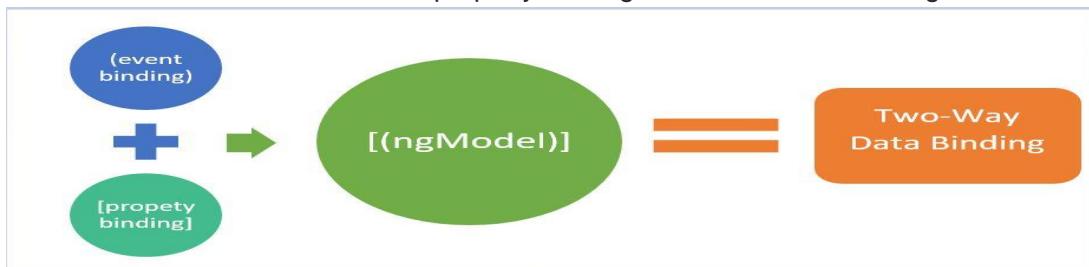
```
<div>
  <b>Enter First Name: </b>
  <input [value]="firstName"
         (input)="firstName=$event.target.value" />
</div>

<p><b>Your First Name: </b>{{firstName}}</p>
```

Let's understand a few important things here:

- [value] = "firstName" is the property binding. We are binding the value property of the input element with the variable (or expression) firstName.
- (input) = "expression" is the event binding. Whenever the input event is fired, the expression will be executed.
- "firstName = \$event.target.value" is an expression that assigns the entered value to the firstName variable.
- The firstName variable can be accessed inside the AppComponent class.

So far, we have seen two-way data binding with ngModel and without ngModel. We can conclude that the directive ngModel is nothing but a combination of property binding and event binding. Event binding is denoted using small brackets and property binding is denoted using square [] brackets. If you notice, the syntax of ngModel is [(ngModel)]. It suggests it is a combination of both event and property binding as shown in below image.



Important Note:

Using two-way binding we can display a data property as well as update that property when user makes changes. We can achieve it in component element and HTML element both. Two-way binding uses the syntax as [] or bindon- keyword. Two-way binding uses the syntax of property binding and event binding together. Property binding uses the syntax as bracket [] or bind- and event binding uses the syntax as parenthesis () or on- and these bindings are considered as one-way binding. Two-way binding works in both direction setting the value and fetching the value.

Using []

[(ngModel)] = "firstName"

Using bindon-

bindon-ngModel = "firstName"

If we want to use NgModel as property binding and event binding separately then we need to use ngModel and ngModelChange as follows:

<input [ngModel] = "firstName" (ngModelChange) = "firstName=\$event"/>

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Two-Way Binding between Components:

Here we will discuss example for two-way data binding between components. We will discuss a scenario in which a string property of parent component will be sent to child component. In child component that property will be updated using text box. Parent component will listen the changes. We will see step by step two-way binding example between the two components.

1. Create a string property in `app.component.ts`.

```
data:string = 'Hello World';
```

2. Create input and output property in `message.component.ts`.

```
@Input() message: string;
@Output() messageChange = new EventEmitter<string>();
```

We will observe that output property name is **input property name + "Change"**.

3. In `message.component.html` we will update `message` using a text box.

```
<input [value]="message" (input)="update($event.target.value)"/>
```

4. For any input in text box, `update()` method will be called.

```
update(val: string) {
    this.message = val;
    this.messageChange.emit(this.message);
}
```

In the above code we are calling `emit()` method.

5. In `app.component.html` we are performing two-way binding.

```
<app-message [(message)]="data"></app-message>
```

Using **bindon-** we can achieve the same goal.

```
<app-message bindon-message="data"></app-message>
```

If we do not use two-way binding then same goal can be achieved using component property binding and custom event binding.

```
<app-message [message]="data" (messageChange)="data=$event"></app-message>
```

Now find the files used in our example.

message.component.ts:

```
import { Component, Input, Output, EventEmitter } from '@angular/core';
@Component({
  selector: 'app-message',
  templateUrl: './message.component.html'
})
export class MessageComponent {
  @Input() message: string;
  @Output() messageChange = new EventEmitter<string>();

  update(val: string) {
    this.message = val;
    this.messageChange.emit(this.message);
  }
}
```

message.component.html:

```
<h3>Message Component</h3>
<p>Message: <input [value]="message" (input)="update($event.target.value)" /></p>
```

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title:string = 'DemoAngularApp1';

  data:string = 'Hello World';
}
```

app.component.html:

```
<h1>{{title}}</h1>
<h3>App Component</h3>
<p>Data: <input [value]="data" (input)="data=$event.target.value" /></p>
<app-message [(message)]="data"></app-message>
```

OR

```
<h1>{{title}}</h1>
<h3>App Component</h3>
<p>Data: <input [(ngModel)]="data" /></p>
<app-message [(message)]="data"></app-message>
```

Output:



DemoAngularApp1

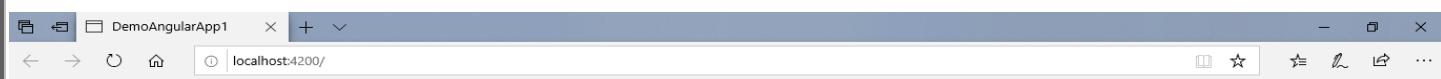
App Component:

Data:

Message Component:

Message:

Now if we change the data of App Component, it will reflect the message in Message Component and the same thing does vice-versa.



DemoAngularApp1

App Component:

Data:

Message Component:

Message:

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Two-Way Binding with Input and Output Property Aliasing:

We will understand here how to use two-way binding between components when `@Input` and `@Output` decorators are using property name aliasing. The property name pattern will be followed in alias name for two-way binding. Now find the code that are using `@Input` and `@Output` decorators.

message.component.ts:

```
import { Component, Input, Output, EventEmitter } from '@angular/core';
@Component({
  selector: 'app-message',
  templateUrl: './message.component.html'
})
export class MessageComponent {
  @Input('myMessage') message: string;
  @Output('myMessageChange') messageChange = new EventEmitter<string>();

  update(val: string) {
    this.message = val;
    this.messageChange.emit(this.message);
  }
}
```

Look at the code of component. The name pattern for two-way binding is being followed in aliases of properties but not in actual property names. The alias for input property `message` is `myMessage` and the alias for output property `messageChange` is `myMessageChange`. Here event name alias is input property name alias adding **Change** as suffix. So we can perform two-way binding with these aliases.

message.component.html:

```
<h3>Message Component:</h3>
<p>Message: <input [value]="message" (input)="update($event.target.value)" /></p>
```

Now we are performing two-way binding in `app.component.html`.

```
<h1>{{title}}</h1>
<h3>App Component:</h3>
<p>Data: <input [value]="data" (input)="data=$event.target.value" /> </p>
<app-message [(myMessage)]="data"></app-message>
```

In `app.component.ts` we are initializing our `data` property.

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

```
export class AppComponent {
  title:string = 'DemoAngularApp1';

  data:string = 'Hello World';
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Two-Way Binding between Components to Change Style:

For the example of two-way binding, now we will provide an example in which we have two buttons as plus (+) and minus (-). On click of these buttons we will change the font size of a text. Here we are using component element two-way binding.

`app.component.ts` will initialize a font size in `textSize` property and it will send it to `textsize.component.ts`. Using plus and minus button we will increase and decrease its value that will also get updated in `app.component.ts`.

To achieve it we have created the property for size with some initial value in `app.component.ts`.

```
textSize:number = 20;
```

In `app.component.html` we are calling `textsize.component.ts` by creating component element with two-way binding as follows.

```
<div>
  <app-textsize [(cdTextSize)]="textSize"></app-textsize>
  <p [style.fontSize.px]="textSize">Hello World!</p>
</div>
```

Now find the `textsize.component.ts` file and its HTML template file.

textsize.component.ts:

```
import {Component, EventEmitter, Input, Output} from '@angular/core';
@Component({
  selector: 'app-textsize',
  templateUrl: './textsize.component.html'
})
export class TextSizeComponent {
  @Input() cdTextSize : number;
  @Output() cdTextSizeChange = new EventEmitter<number>();
  plus() {
    this.cdTextSize = this.cdTextSize + 1;
    this.emitSize();
  }
  minus() {
    this.cdTextSize = this.cdTextSize - 1;
    this.emitSize();
  }
  emitSize() {
    this.cdTextSizeChange.emit(this.cdTextSize);
  }
}
```

textsize.component.html:

```
<button (click)="plus()"> + </button>
&nbsp;
<button (click)="minus()"> - </button>
```

Output:



DemoAngularApp1



Hello World!

Two-Way Binding with NgModel in Select Box:

Here we will provide example for two-way binding in which we will show how to use **NgModel** with HTML select box. The scenario is that an array of color will be populated by select box. A default color initialized by component property will be selected initially in select box and a sample text is using that color. On change of color in select box, the color of text will change.

selectbox.component.ts:

```
import {Component, Input} from '@angular/core';
@Component({
  selector: 'app-select',
  templateUrl: './selectbox.component.html'
})
export class SelectBoxComponent {
  @Input() cdColors : Array<string>
  myColor:string = 'GREEN';
}
```

selectbox.component.html:

```
<!--<select [value] ="myColor" (change)="myColor=$event.target.value">-->
<select [(ngModel)] ="myColor">
  <option *ngFor = "let color of cdColors" [value]="color">
    {{color}}
  </option>
</select>

<p [style.color]="myColor"> Hello World!</p>
```

Here we are using **NgModel** for two-way binding. We will observe that **myColor** is initially providing a default color as selected in select box and on change of color selection the value of **myColor** is getting changed. If we do not want to use **NgModel** we will do as follows.

```
<select [value] ="myColor" (change)="myColor=$event.target.value">
```

We are using **selectbox.component.ts** in **app.component.html** as follows.

```
<div>
  <app-select [cdColors] ="colors"></app-select>
</div>
```

app.component.ts is defining the array property **colors**.

```
colors:string[] = ['RED', 'GREEN', 'YELLOW'];
```

Output:



DemoAngularApp1

GREEN ▾

Hello World!

Two-Way Binding between Components using Object Property

Here we will use an object property for two-way binding between components. We have a class as follows:

employee.ts:

```
export class Employee {
    constructor(public id : number, public name : string) {
    }
}
```

Now our scenario is that we will change a given text as uppercase and lowercase on click of two radio button. In `app.component.ts` we are creating an object of `Employee` class. Using two-way binding we will send `name` property of `Employee` class to `case.component.ts`. In this component we change the text as lowercase and uppercase on the click of radio button and then send this changed data to `name` property of `Employee` class in `app.component.ts`. Now find the code to change lower and upper case.

case.component.ts:

```
import {Component, EventEmitter, Input, Output} from '@angular/core';
@Component({
    selector: 'app-case',
    templateUrl: './case.component.html'
})
export class CaseComponent {
    @Input() myName : string;
    @Output() myNameChange = new EventEmitter<string>();

    changeCase(val:string) {
        if (val == 'upper') {
            this.myName = this.myName.toUpperCase();
        }
        else {
            this.myName = this.myName.toLowerCase();
        }
        this.myNameChange.emit(this.myName);
    }
}
```

case.component.html:

```
<input type="radio" id="upper" name="case" (click)="changeCase('upper')"/>
<label for="upper">Upper Case</label>
<br />
<input type="radio" id="lower" name="case" (click)="changeCase('lower')"/>
<label for="lower">Lower Case</label>
```

We are performing two-way binding in `app.component.html` as follows:

```
<div>
    <app-case [(myName)] = "emp.name"> </app-case>
    <p><b>Name: </b>{{emp.name}} </p>
</div>
```

In `app.component.ts` we are instantiating `Employee` class as follows:

```
emp = new Employee(1, 'Rakesh Singh');
```

Output:



DemoAngularApp1

○ Upper Case
○ Lower Case

Name: Rakesh Singh

Two-Way Binding with NgModel using Object Property

Here we will use **NgModel** for two-way binding with object property. Our scenario is that we will create text box which will display value of **name** property of **Employee** class and on change of text box value the **name** property of **Employee** class will also get updated.

uppercase.component.ts:

```
import {Component} from '@angular/core';
import {Employee} from './employee';

@Component({
  selector: 'app-uppercase',
  templateUrl: './uppercase.component.html'
})

export class UpperCaseComponent {
  employee = new Employee(100, 'Rakesh Singh');

  toUpper(val:string) {
    this.employee.name = val.toUpperCase();
  }
}
```

uppercase.component.html:

```
<input [(ngModel)]="employee.name"/> {{employee.name}}
<br/>
<input [ngModel]="employee.name" (ngModelChange)="toUpper($event)"/> {{employee.name}}
```

If we use **ngModelChange** for event binding then we can call our component method and can pass data to it. Now we will use **uppercase.component.ts** in **app.component.html** as follows:

app.component.html:

```
<div>
  <app-uppercase></app-uppercase>
</div>
```

Output:



DemoAngularApp1

Rakesh Singh	Rakesh Singh
Rakesh Singh	Rakesh Singh

If we do changes in the first textbox it will update the same to second textbox and print as well but if we do changes in second textbox it will convert into uppercase and update the same to first textbox and print as well.



DemoAngularApp1

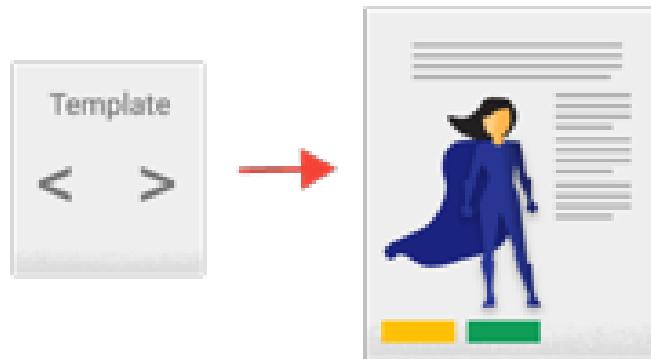
RAKESH SINGH	RAKESH SINGH
RAKESH SINGH	RAKESH SINGH

Templates & Views in Angular:

What is Angular Template and View?

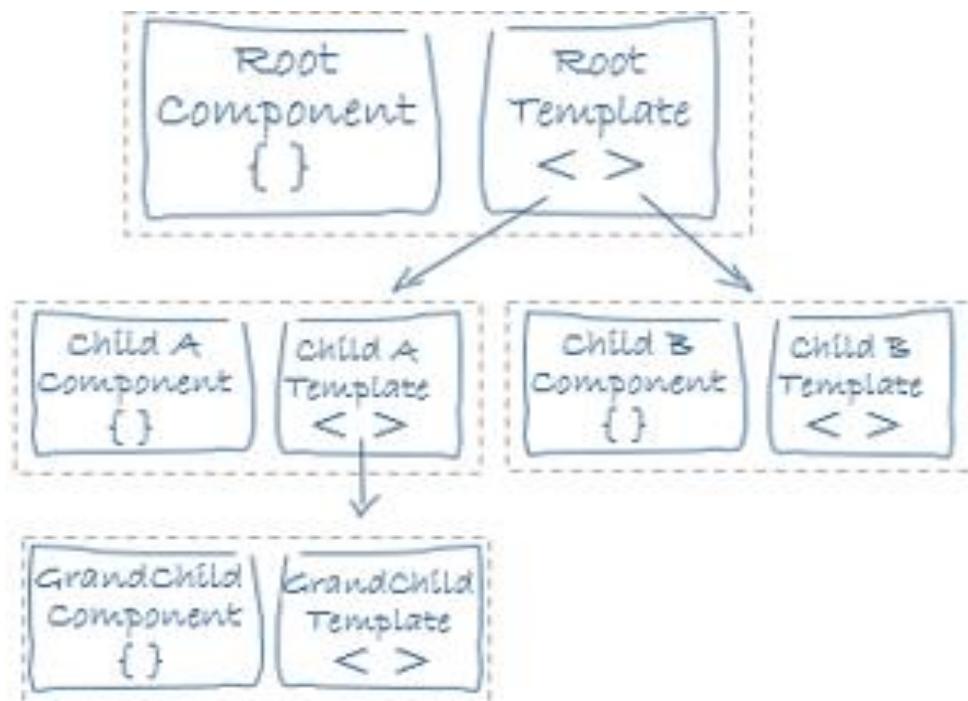
A template is an **HTML snippet** that tells Angular how to render the component in angular application.

The template is immediately associated with a component defines that component's **view**.



Angular View Hierarchy:

The component can also contain a **view hierarchy**, which have embedded views, defined or associated with other components.



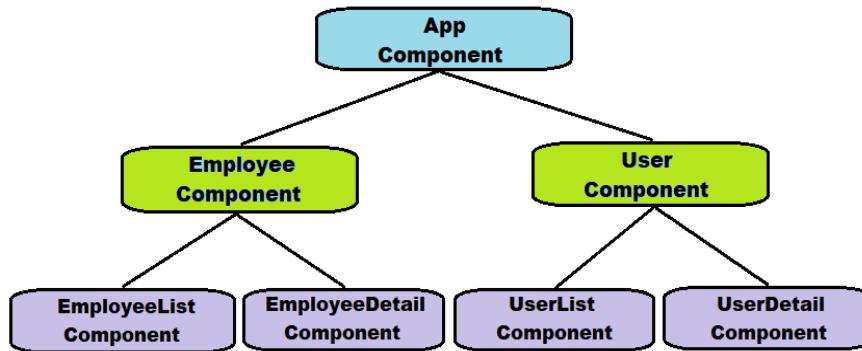
A view hierarchy can include views from components in the same NgModule, but it can also include views from components that are defined in different NgModules.

Key points to consider about **view hierarchy** are as follows:

- It is tree of related views that can act as one independent unit.
- The root view is often called as **component's host view**.
- It plays an important role in Angular change detection.

View Hierarchy Example

The image below shows the view hierarchy for an application managing Employee and User.



- The App Component is at the root level and is called as **host view** containing **Employee** and **User** components.
- The **Employee Component** acts as a host view for its child components **Employee List** and **Employee Detail** which will have their respective views.
- Likewise **User Component** further hosts two child components containing views for **User List** and **User Detail** components respectively.

Here each component in the hierarchy may have a view associated with it.

Types of Templates:

There are two ways of defining template in an angular component.

1. Inline Template:

The inline template is defined by placing the HTML code in backticks (`) and is linked to the component metadata using the **template** property of **@Component** decorator e.g.

```

import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-sample',
  template: `
    <p>
      sample works!
    </p>
  `,
  styleUrls: ['./sample.component.css']
})
export class SampleComponent implements OnInit {
  constructor() { }
  ngOnInit() {
  }
}
  
```

Can't we include the HTML code of template within single or double quotes?

Yes, you can include the HTML code of template within a pair of either single quotes or double quotes as long as your HTML code is in a single line as shown below. Here, we are using single quotes.

template: '<p> sample works! </p>'

When to use backticks (`) instead of either single quotes or double quotes?

When you have multiple lines of HTML code, then you need to use the backticks (`); otherwise, you will get a compile-time error.

To define the inline template using **@angular/cli**, use the command below:

```
> ng generate component Sample --inlineTemplate=true
```

When true, includes template inline in the component.ts file. By default, an external template file is created and referenced in the component.ts file. Default: false Aliases: -t or -it

2. Template File (External File):

The template is defined in a separate HTML file and is linked to the component metadata using the `@Component` decorator's `templateUrl` property e.g.

sample.component.ts:

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

@Component({
  selector: 'app-sample',
  templateUrl: './sample.component.html',
  styleUrls: ['./sample.component.css']
})

export class SampleComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }
}
```

sample.component.html:

```
<p>
  sample works!
</p>
```

When to use templateUrl in Angular applications?

When you have a complex view, then it recommended by Angular to create that complex view in an external HTML file instead of an inline template. The Angular component decorator provides a property called `templateUrl`. This property takes the path of an external HTML file.

Angular Template vs TemplateUrl and when do we need to use one over the other?

According to Angular, when you have a complex view (i.e. a view with more than 3 lines), then go with `templateUrl` (use external file); otherwise, use the `template` (inline HTML) property of the component decorator.

When you use an inline template, then you will lose the Intellisense support, code-completion, and formatting features of Visual Studio Code. But with an external view template, you will get the Intellisense support, code-completion, and formatting features of Visual Studio Code.

There is a convenience factor to having the TypeScript code and the associated HTML in the same file because it is easier to see how the two are related to each other.

To define the external template using `@angular/cli`, use the command below:

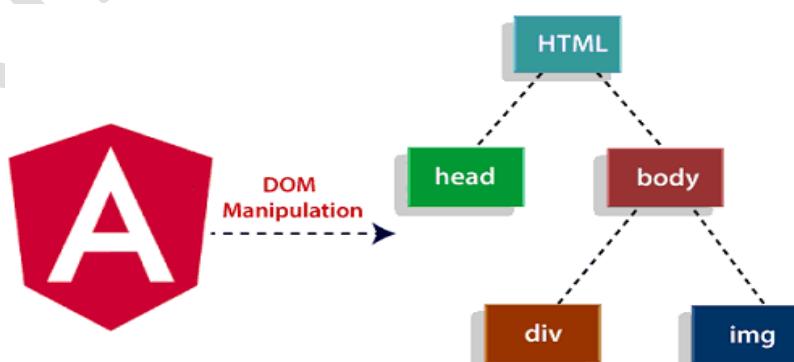
```
> ng generate component Sample
```

Directives in Angular:

The directive is another main building block of the Angular framework. Using directive, we can create different reusable UI controls of the web application in which we can define design as well as business logic and those controls can be used in different components in the applications.

What is the Directive?

A directive manipulates the DOM by changing the appearance, behavior, or layout of DOM elements. It also helps you to extend HTML.



Directives just like Component are one of the core building blocks in the Angular framework to build applications.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Angular Directive is basically a class with a `@Directive` decorator. You might be wondering what are decorators? **Decorators** are functions that modify JavaScript classes. Decorators are used for attaching metadata to classes, it knows the configuration of those classes and how they should work.

You would be surprised to know that a component is also a directive-with-a-template. A `@Component decorator` is actually a `@Directive decorator` extended with template-oriented features. Whenever Angular renders a directive, it changes the DOM according to the instructions given by the directive. Directives acts as an HTML markers on any DOM element (like as an attribute or element name or CSS based style class) which instructed Angular's HTML compiler to attach a specified behaviour on that particular DOM element or to transform that DOM element.

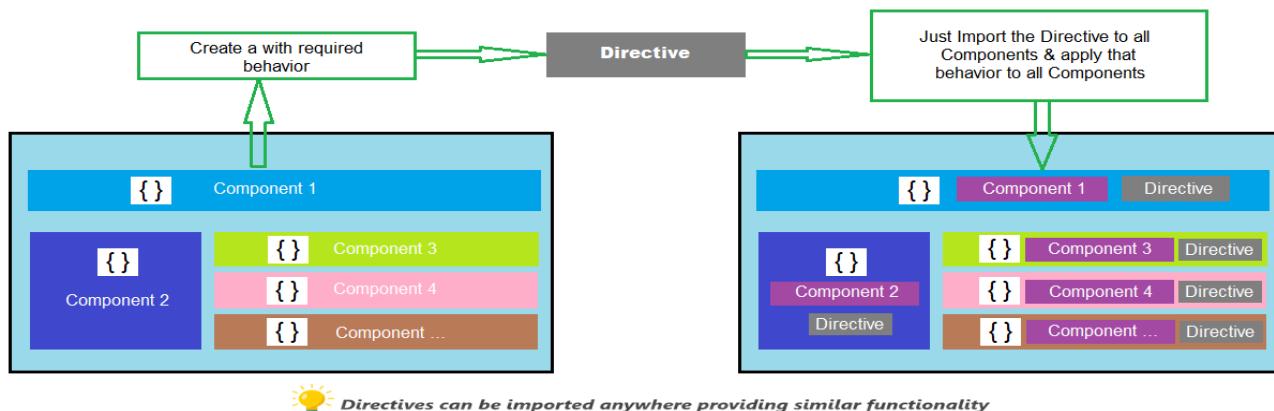
Basic Concept of Directives

In Angular Framework, one of the most important elements is Directives. And if we analyse the directive in brief, then we will discover that the main building block of the Angular framework which is known as Component is basically a directive. So, in a simple word, each and every Angular component is just a directive with a custom HTML template. So, in real word when defining a component as the main building block of Angular application, actually we want to say that directives are the main building blocks of Angular applications.

In general, a directive is a TypeScript based function that executes whenever Angular compiler identified it within the DOM element. Directives are used to provide or generate new HTML based syntax which will extend the power of the UI in an Angular Application. Each directive must have a selector name just like the same as a component – either that name can be from Angular predefined patterns like `ng-if` or a custom developer-defined name which can be any name but need to indicate the main purpose of the directive. Also, every directive can act as an element or an attribute or a class or a comment in the HTML section.

Why Directives Required?

You might be wondering why we need Angular Directives. Now take a look at the below image, if you want a similar functionality in all the components for an example fade-in and fade-out functionality, you can take two approaches. The common approach would be, you can explicitly write the code in all the components for the required behavior, but it would be tedious and complex. Alternatively, like a function in a programming language, you can write the code and later you can call it anytime whenever you want that behavior of that function. Similarly, you can create a directive and write the behavior inside it. Then, wherever you need that behaviour, you can import the directive.



In Angular Framework, Directives always ensure the high-level of reusability of the UI controls throughout the application. With the help of Directives, we can develop UIs with many movable parts and at the same time, we can streamline the development flow for the engineers. The main reason for using directives in any Angular applications are:

- Reusability** – In an Angular application, the directive is a self-sufficient part of the UI. As a developer, we can reuse the directive across the different parts of the application. This is very much useful in any large-scale applications where multiple systems need the same functional elements like search box, date control, etc.
- Readability** – Directive provides much more readability for the developers to understand the production functionality and data flow.
- Maintainability** – One of the main use of directive in any application is the maintainability. We can easily decouple the directive from the application and replace the old one with a new one directives.

Component vs Directives:

The comparison between Component and directives are as below –

Component	Directives
A component is defined with the @Component decorator	A Directive is defined with the @Directive decorator
A component is a directive that uses a shadow DOM to create encapsulated visual behavior called a component. Components are typically used to create UI widgets.	Directive mainly used to provide new behavior within the existing DOM elements.
With the help of the component, we can break down the application in multiple small parts.	With the help of the directive, we can design any type of reusable component.
In the browser DOM, only one component can be activated as a parent component. Other components will act like a child component in that case.	Within a single DOM element, any no of directives can be used.
@View decorator or templateUrl template is mandatory in the component.	Directives don't use View.
Component is used to define pipes.	You can't define Pipes in directive.
ViewEncapsulation can be define in components because they have views.	Directive don't have views. So you can't use ViewEncapsulation in directive.

@Directive Decorator Metadata:

@Directive decorator is used to defining any class as an Angular Directive. We can always define any directive to specify the custom behavior of the elements within the DOM. @Directive () metadata is contained below-mentioned major options:

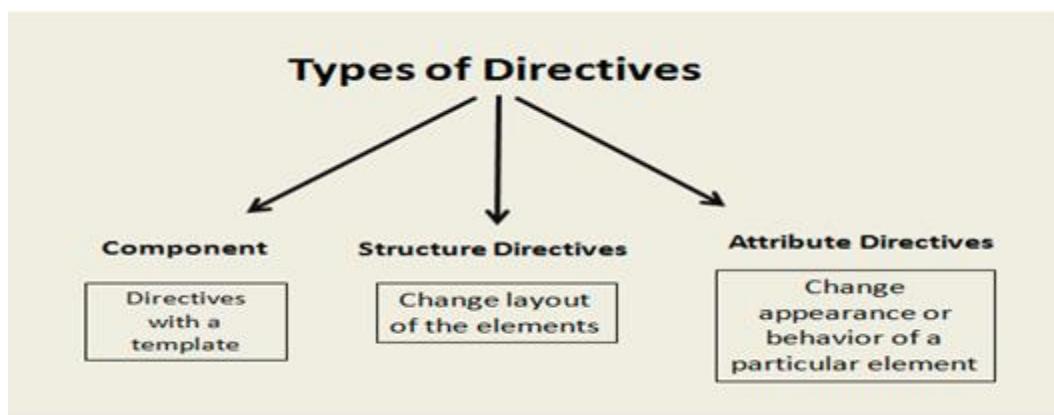
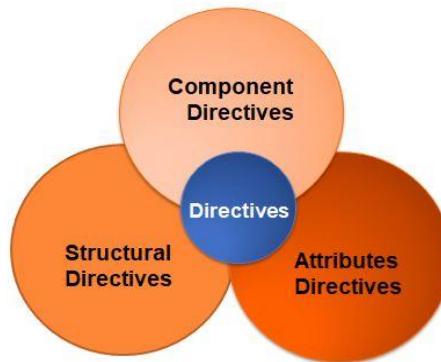
```
@Directive({
  selector?: string
  inputs?: string[]
  outputs?: string[]
  providers?: Provider[]
  exportAs?: string
  queries?: {...}
  host?: {...}
})
```

- selector:** The selector property is used to identify the directive within the HTML template. Also, with the help of this property, we can initialize the directive from the DOM elements
- inputs:** It is used to provide the set of data-bound input properties of the directives.
- outputs:** It is used to enumerates any event properties from the directives.
- providers:** It is used to inject any provider type like service or components within the directives.
- exportAs:** It is used to define the name of the directives which can be used to assign a directive as a variable.
- queries:** It Configures the queries that will be injected into the directive.
- host:** It maps class properties to host element bindings for properties, attributes, and events, using a set of key-value pairs.

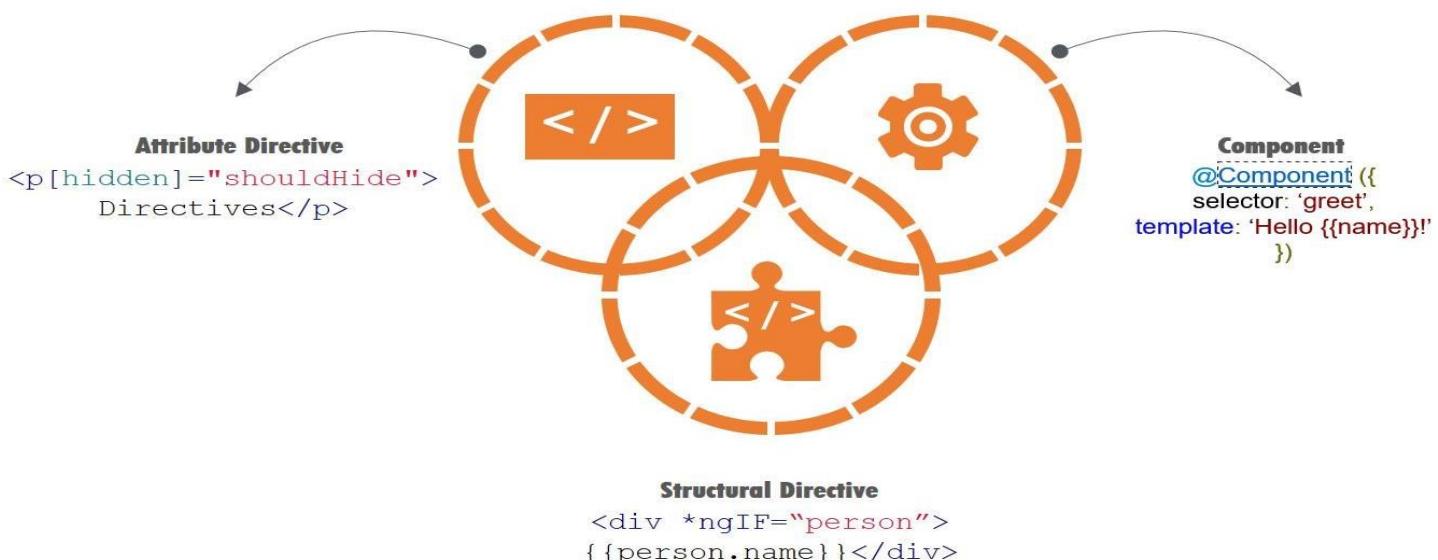
Types of Directives:

There are three main types of directives in Angular:

- **Component:** Directives with templates.
- **Structural Directives:** Directives that change the behavior of a component or element by affecting the template.
- **Attribute Directives:** Directives that change the behavior of a component or element but don't affect the template.



Directives in Angular



Component Directive:

Component directive, is nothing but a simple class which is decorated with the `@Component` decorator. Component directive is used to specify the HTML templates. It has structure design and the working pattern of how the component should be processed, instantiated and used at runtime. It is the most commonly-used directive in any Angular project.

Above, we discussed the component, that should be the Component Directive.

- `sample.component.css`: contains all the CSS styles for the component.
- `sample.component.html`: contains all the HTML code used by the component to display itself.
- `sample.component.ts`: contains all the code used by the component to control its behavior.

sample.component.ts:

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

```
@Component({
  selector: 'app-sample',
  templateUrl: './sample.component.html',
  styleUrls: ['./sample.component.css']
})
```

```
export class SampleComponent implements OnInit {
```

```
  constructor() { }
```

```
  ngOnInit() {
  }
}
```

sample.component.html:

```
<p>
  sample works!
</p>
```

sample.component.css:

specify the stylesheet for html template if needed.

Structural Directive:

Structural Directive changes the DOM layout design and structure. The main usage of Structural Directive is for manipulating and making run-time changes on the DOM elements. In other words, structural directives can change the DOM layout by adding, removing and manipulating DOM elements. You can easily recognize all structural directives which are preceded by asterisk (*) symbol.

Built-in structural directive - `ngIf`, `ngFor`, and `ngSwitch`

- **ngIf** is used to create or remove a part of DOM tree depending on a condition.
- **ngFor** is used to customize data display. It is mainly used for display a list of items using repetitive loops. In other words, it is used to iterate over the list of items or collection of items.
- **ngSwitch** is like the JavaScript switch. It can display *one* element from among several possible elements, based on a *switch condition*. Angular puts only the *selected* element into the DOM.



ngIf:

ngIf is a directive that is used to add an element subtree to the DOM on the basis of true value of an expression. If the value of expression is false then the element subtree will be removed from the DOM.

To use ngIf we need to prefix it with asterisk (*) as *ngIf. Hiding and displaying element subtree using CSS visibility property is not same as work done by ngIf. CSS visibility property does not remove element subtree from DOM, whereas ngIf removes the element subtree from DOM for false value of expression. CSS visibility property does not physically remove the element subtree when hiding, so it continues to consume resources and memory that will affect the performance. In case of ngIf it physically removes element subtree, so it does not consume resources and memory which in turn results into better performance. ngIf is used with HTML elements as well as component elements.

ngIf is used with HTML elements as follows:

```
<div *ngIf="isValid"> Data is valid. </div>
```

ngIf is used with component elements as follows:

```
<app-emp *ngIf="emp" [name]="emp.name"></app-emp>
```

ngIf with HTML Elements:

Here we will understand using ngIf with HTML elements. ngIf is an angular directive that is used to add an element subtree for true value of expression. NgIf is used as *ngIf="expression". Here if "expression" value is either false or null, in both cases the element subtree will not be added to DOM. Now find the below code snippet.

```
<p *ngIf="isValid">
  Data is Valid.
</p>
<p *ngIf="!isValid">
  Data is not Valid.
</p>
```

Here isValid is a component property that has been initialized as follows:

isValid:boolean = true;

Hence the output will be

Data is valid

And !isValid will return false, so *ngIf="!isValid" will not add element in DOM.

NgIf with NgFor and NgClass

Here we will discuss how to use ngIf with ngFor and ngClass. Suppose we have two CSS classes as .one and .two. To use it with alternating row using ngIf, we can do it as follows:

```
<div *ngFor="let id of ids">
  Id is {{id}}
  <div *ngIf="id%2 == 0">
    <div [ngClass]="'one'">Even Number</div>
  </div>
  <div *ngIf="id%2 == 1">
    <div [ngClass]="'two'">Odd Number</div>
  </div>
</div>
```

In the above code snippet ids has been initialized in component as an array of numbers as follows:

```
ids:number[] = [1,2,3,4,5];
```

ngFor will iterate the given array and hence id%2 will return 0 and 1 alternatively. If the value is 0 then element with CSS class .one will be added to DOM otherwise the element with CSS class .two will be added to DOM.

ngIf with Enum:

TypeScript **enum** can also be used with **ngIf**. To understand this we are creating an **enum** as following:

numEnum.ts:

```
export enum NumEnum {
  ONE = 1,
  TWO = 2,
  THREE = 3
}
```

We will import this file in our component as given below:

```
import {NumEnum} from './numEnum';
```

Now we need to create a component property and will assign our **NumEnum** to it. Find the line of code.

```
myNumEnum = NumEnum;
```

In this way we are ready to use **enum** with **ngIf** in our code. Find the code snippet.

```
<div *ngIf="myNumEnum.TWO > 1">
  {{myNumEnum.TWO}} is greater than 1.
</div>
```

In our enum **myNumEnum.TWO** has value as 2. So the expression **myNumEnum.TWO > 1** will return true. Hence the output will be as follows.

2 is greater than 1.

ngIf Null Check:

For the null value of expression assigned to **ngIf**, it does not add element tree in DOM. So it is useful while accessing property from object to avoid error. For the example find the class.

employee.ts:

```
export class Employee {
  constructor(public id : number, public name : string) {
  }
}
```

We will import it in our component as follows:

```
import {Employee} from './employee';
```

Now we will define two variables of class **Employee** type in our component as given below:

```
emp1 = new Employee(101, 'David');
emp2 : Employee;
```

Here we will notice that **emp1** has been instantiated but **emp2** has not been instantiated. Now find the code in HTML template.

```
<div *ngIf="emp1">
  Id:{{emp1.id}} Name: {{emp1.name}}
</div>
<div *ngIf="emp2">
  Id:{{emp2.id}} Name: {{emp2.name}}
</div>
```

In the above code snippet before accessing object property, we are checking it by **ngIf**. The first **ngIf** will add the element in DOM but second will not because of null value of **emp2**. Hence expression **{{emp2.id}}** and **{{emp2.name}}** will not execute and no chance of error because of null value of **emp2**.

ngIf with Component Elements:

Here we will use `ngIf` with component elements. For the example we are creating a child component as given below.

message.component.ts:

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

@Component({
  selector: 'app-message',
  template: '<p> Hello {{name}} </p>'
})

export class MessageComponent {
  @Input() name : string;
}
```

app.component.html:

```
import { Component } from '@angular/core';
import {Employee} from './employee';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
```

```
export class AppComponent {
  title:string = 'DemoAngularApp1';

  emp1 = new Employee(101, 'David');
  emp2 : Employee;
}
```

app.component.html:

```
<app-message *ngIf="emp1" [name]="emp1.name"></app-message>
```

Here we are performing component property binding and using `ngIf`. Now if `emp1` will be null or undefined, the child component with selector `app-message` will not execute.

Nglf-Then-Else:

Angular 4 has introduced `ngIf` with `then` and `else`. `ngIf` conditionally includes a template based on the value of expression. It adds or removes the HTML element in DOM layout. If the expression value is true then `ngIf` renders the `then` template in its place and if the expression value is false then `ngIf` renders the `else` template in its place. `then` template is the inline template of `ngIf` unless bound to a different value and `else` template is blank unless it is bound. It is not necessary to use `then` and `else` with `ngIf`. In simple way `ngIf` is used to show conditionally inline template. If condition in `ngIf` is false and it is necessary to show a template then we use `else` and its binding point is `<ng-template>`. Usually `then` is the inline template of `ngIf` but we can change it and make non-inline using a binding and binding point will be `<ng-template>`. Because `then` and `else` are bindings, the template references can change at runtime.



<ng-template> Element:

<ng-template> is an angular element for rendering HTML. It is never displayed directly. Suppose we have following code in our HTML template.

```
<ng-template>
  <p>Hello World!</p>
</ng-template>
```

When the code runs then the code written inside <ng-template> will not be displayed. It writes a comment <!-->.

<ng-template> is used by structural directive such as nglf.

Using nglf:

ngIf is an structural directive that can add or remove host element and its descendants in DOM layout at run time. It conditionally shows the inline template. ngIf works on the basis of true and false result of given expression. If condition is true, the elements will be added into DOM layout otherwise they will be removed. ngIf can be used in different ways.

```
<div *ngIf="condition">...</div>
<ng-template [ngIf]="condition"><div>...</div></ng-template>
```

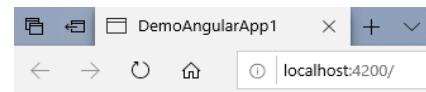
Now we will have an example of nglf directive.

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title:string = 'DemoAngularApp1';
  isValid: boolean = true;
  changeValue(valid: boolean) {
    this.isValid = valid;
  }
}
```

app.component.html:

```
<h3>Using ngIf</h3>
<div>
  <input id="rdb1" type="radio" name="rdb" [checked]="isValid" (click)="changeValue(true)">
  <label for="rdb1">True</label>
  <input id="rdb2" type="radio" name="rdb" (click)="changeValue(false)">
  <label for="rdb2">False</label>
</div>
<br />
<div *ngIf="isValid">
  <b>Data is Valid.</b>
</div>
<div *ngIf="!isValid">
  <b>Data is Invalid.</b>
</div>
```



DemoAngularApp1

Using ngIf

True False

Data is Invalid.

In the above code two <div> host element is using ngIf. On the basis of true and false condition, the respective <div> will be added or removed from the DOM.

Using `ngIf` with `Else`:

In this example we will use `ngIf` with `else`. In our application `else` is used when we want to display something for false condition. The `else` block is used as follows:

```
<div *ngIf="condition; else elseBlock">...</div>
<ng-template #elseBlock>...</ng-template>
```

For `else` block we need to use `<ng-template>` element. It is referred by a template reference variable. `ngIf` will use that template reference variable to display `else` block when condition is false. The `<ng-template>` element can be written anywhere in the HTML template but it will be readable if we write it just after host element of `ngIf`. In the false condition, the host element will be replaced by the body of `<ng-template>` element. Now find the example of `ngIf` with `else`.

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title:string = 'DemoAngularApp1';
  isValid: boolean = true;
  age: number = 12;

  changeValue(valid: boolean) {
    this.isValid = valid;
  }
}
```

app.component.html:

```
<h1>{{title}}</h1>
<h3>Using NgIf with Else</h3>
<b>Example-1:</b><br/><br/>
<div>
  <input id="rdb1" type="radio" name="rb" [checked]="isValid" (click)="changeValue(true)">
  <label for="rdb1">True</label>
  <input id="rdb2" type="radio" name="rb" (click)="changeValue(false)">
  <label for="rdb2">False</label>
</div>
<br />
<div *ngIf="isValid; else elseBlock">
  <b>Data is Valid.</b>
</div>
<ng-template #elseBlock>
  <div>
    <b>Data is Invalid.</b>
  </div>
</ng-template>
<br/><b>Example-2:</b><br/><br/>
<div>Enter Age: <input type="text" [(ngModel)]='age' /></div><br/>
<div *ngIf="age < 18; else elseAgeBlock">
  <b>Not Eligible to Vote.</b>
</div>
<ng-template #elseAgeBlock>
  <b>Eligible to Vote.</b>
</ng-template>
```

In the above code, we are using two examples of `ngIf` directive. When condition is true, the host element with its descendants will be added in DOM layout and if condition is false then `<ng-template>` code block will run in the place of host element.

Output:



DemoAngularApp1

Using NgIf with Else

Example-1:

True False

Data is Valid.

Example-2:

Enter Age:

Not Eligible to Vote.

Form with storing the value locally:

```
<div *ngIf="condition as value; else elseBlock">{{value}}</div>
<ng-template #elseBlock>Content to render when condition is false.</ng-template>
```

Using NgIf with Then and Else:

In this example we will use `NgIf` with `then` and `else`. `then` template is the inline template of `NgIf` and when it is bound to different value then it displays `<ng-template>` body having template reference value as `then` value.

`ngIf` with `then` and `else` is used as follows:

```
<div *ngIf="condition; then thenBlock else elseBlock"></div>
<ng-template #thenBlock>...</ng-template>
<ng-template #elseBlock>...</ng-template>
```

When condition is true, then the `<ng-template>` with reference variable `thenBlock` is executed and when condition is false then the `<ng-template>` with reference variable `elseBlock` is executed.

The value of `thenBlock` and `elseBlock` can be changed at run time. We can have more than one `<ng-template>` for `then` and `else` block and at runtime we can switch to those `<ng-template>` by changing the value of `thenBlock` and `elseBlock`. At one time only one `<ng-template>` for `thenBlock` or `elseBlock` will run.

Example:

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  preserveWhitespaces: true
})
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

```

export class AppComponent implements OnInit{
  title:string = 'DemoAngularApp1';
  isValid:boolean = true;
  age:number = 12;
  myThenBlock: TemplateRef<any> = null;
  myElseBlock: TemplateRef<any> = null;

  @ViewChild('firstThenBlock',{static: true})
  firstThenBlock: TemplateRef<any> = null;
  @ViewChild('secondThenBlock',{static: true})
  secondThenBlock: TemplateRef<any> = null;

  @ViewChild('firstElseBlock',{static: true})
  firstElseBlock: TemplateRef<any> = null;
  @ViewChild('secondElseBlock',{static: true})
  secondElseBlock: TemplateRef<any> = null;

  ngOnInit() {
    this.myThenBlock = this.firstThenBlock;
    this.myElseBlock = this.firstElseBlock;
  }

  changeValue(valid: boolean) {
    this.isValid = valid;
  }

  toggleThenBlock() {
    this.myThenBlock = this.myThenBlock === this.firstThenBlock ? this.secondThenBlock : this.firstThenBlock;
  }
  toggleElseBlock() {
    this.myElseBlock = this.myElseBlock === this.firstElseBlock ? this.secondElseBlock : this.firstElseBlock;
  }
}

```

app.component.html:

```

<h1>{{title}}</h1>
<h3>Using NgIf with Then and Else</h3>
<b>Example-1:</b><br/><br/>
<div>
  <input id="rdb1" type="radio" name="rdb" [checked]="isValid" (click)="changeValue(true)">
  <label for="rdb1">True</label>
  <input id="rdb2" type="radio" name="rdb" (click)="changeValue(false)">
  <label for="rdb2">False</label>
</div><br/>

```

```

<div *ngIf="isValid; then thenBlock; else elseBlock"> </div>
<ng-template #thenBlock>
  <div><b>Data is Valid.</b></div>
</ng-template>
<ng-template #elseBlock>
  <div><b>Data is Invalid.</b></div>
</ng-template>
<br/><b>Example-2:</b><br/><br/>
<div>Enter Age: <input type="text" [(ngModel)]='age'> </div><br/>
<div>
  <button type="button" (click)= "toggleThenBlock()">Toggle Then Block</button>
  <button type="button" (click)= "toggleElseBlock()">Toggle Else Block</button>
</div>
<br/>
<div *ngIf="age > 17; then myThenBlock; else myElseBlock"></div>
<ng-template #firstThenBlock>
  <div> <b>First Then Block: Eligible to Vote.</b></div>
</ng-template>
<ng-template #secondThenBlock>
  <div> <b>Second Then Block: Eligible to Vote.</b></div>
</ng-template>

<ng-template #firstElseBlock>
  <div> <b>First Else Block: Not Eligible to Vote</b></div>
</ng-template>
<ng-template #secondElseBlock>
  <div> <b>Second Else Block: Not Eligible to Vote</b> </div>
</ng-template>

```

We have created two scenarios to use `ngIf` with `then` and `else`. In the first scenario we have one `<ng-template>` for `then` and one `<ng-template>` for `else` block. In the second scenario we have more than one `<ng-template>` for `then` and more than one `<ng-template>` for `else` block. But at one time only one `<ng-template>` will run. In the second scenario we have buttons to change the value of `then` and `else` binding point.

Using `@ViewChild` decorator we get the instance of `TemplateRef` for a given template reference variable. In the `toggleThenBlock()` we are switching the value of `then` and in the `toggleElseBlock()` we are switching the value of `else`.



DemoAngularApp1

Using NgIf with Then and Else

Example-1:

True False

Data is Valid.

Example-2:

Enter Age:

First Else Block: Not Eligible to Vote



ngFor:

The *ngFor directive is used to repeat a portion of HTML template once per each item from an iterable list (Collection). The ngFor is an Angular structural directive and is similar to ng-repeat in AngularJS. Some local variables like index, first, last, odd and even are exported by *ngFor directive.

In other words we can say *ngFor is a directive that iterates over collection of data. It is used to customize data display. Whenever there is change in collection of data at runtime, that will be updated in data display iterated by ngFor directive. ngFor provides local variables that will help to get index of current iteration, detect if element in iteration is first or last and odd or even. ngFor is used with HTML element as well as component element. ngFor with HTML element iterates the template within it. ngFor with component element iterates the child component HTML template. ngFor can be used with asterisk(*) or ng-template element. Any change in contents in iteration is propagated to DOM.

Syntax of ngFor

The simplified syntax for the ngFor is as shown below:

```
<li *ngFor="let item of items;"> .... </li>
```

*ngFor

The syntax starts with *ngFor. The * here represents the Angular template syntax.

let item of items;

Here the items are a collection that we need to show to the user. The let keyword creates a local variable named item. You can then use this variable to reference the individual item in the items collection. You can use this variable anywhere inside your template.

How to use ngFor Directive?

To Use ngFor, First, you need to create a block of HTML elements, which can display a single item of the items collection. Then use the ngFor directive to tell angular to repeat that block of HTML elements for each item in the list.

Example of ngFor:

First, you have to create an angular Application. After that open the app.component.ts and add the following code.

The following Code contains a list of employees in a employees array. We will build a template to display these employees in a tabular form.

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title: string = 'Employee List';
  employees: Employee[] = [
    {EmpId: 101, EmpName: "Smith", EmpJob: "Programmer", EmpSalary: 18000, DeptName: "IT"}, 
    {EmpId: 102, EmpName: "David", EmpJob: "HR Manager", EmpSalary: 25000, DeptName: "HR"}, 
    {EmpId: 103, EmpName: "Peter", EmpJob: "Sales Manager", EmpSalary: 22000, DeptName: "Sales"}, 
    {EmpId: 104, EmpName: "Martin", EmpJob: "Marketing Manager", EmpSalary: 28000, DeptName: "Marketing"}, 
    {EmpId: 105, EmpName: "Fleming", EmpJob: "Finance Manager", EmpSalary: 35000, DeptName: "Finance"}
  ];
}
class Employee{
  EmpId : number;
  EmpName : string;
  EmpJob : string;
  EmpSalary : number;
  DeptName: string
}
```

The next step is to create HTML template. Open the app.component.html and add the following code:

app.component.html:

```
<h1>{{title}}</h1>
<table border="1" style="border-collapse: collapse; width: 100%;">
  <thead>
    <tr>
      <th>Emp Id</th>
      <th>Emp Name</th>
      <th>Emp Job</th>
      <th>Emp Salary</th>
      <th>Dept Name</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let employee of employees;">
      <td>{{employee.EmpId}}</td>
      <td>{{employee.EmpName}}</td>
      <td>{{employee.EmpJob}}</td>
      <td>{{employee.EmpSalary}}</td>
      <td>{{employee.DeptName}}</td>
    </tr>
    <tr *ngIf="!employees || employees.length==0">
      <td colspan="5" style="color: red;">
        No Employees to Display !!!
      </td>
    </tr>
  </tbody>
</table>
```

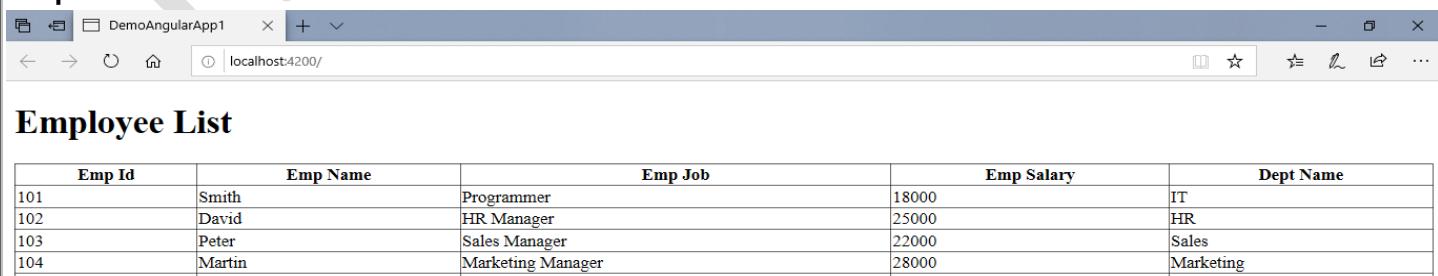
The most important part of the code is:

```
<tr *ngFor="let employee of employees;">
  <td>{{employee.EmpId}}</td>
  <td>{{employee.EmpName}}</td>
  <td>{{employee.EmpJob}}</td>
  <td>{{employee.EmpSalary}}</td>
  <td>{{employee.DeptName}}</td>
</tr>
```

Our purpose is to display the list of employees. Hence we build a tr element of a table, which can display a single employee. We want the tr element to be repeated for each item in the employees collection. Hence we apply ngFor directive to the tr element of the table.

The “let employee of employees” creates the local variable employee. You can use the variable to create the template. When you run the application using ng serve --o command and you should be able to see the list of employees in tabular form displayed in your browser.

Output:



Emp Id	Emp Name	Emp Job	Emp Salary	Dept Name
101	Smith	Programmer	18000	IT
102	David	HR Manager	25000	HR
103	Peter	Sales Manager	22000	Sales
104	Martin	Marketing Manager	28000	Marketing
105	Fleming	Finance Manager	35000	Finance

Local Variables in ngFor:

ngFor also provides several values to help us manipulate the collection. We can assign the values of these exported values to the local variable and use it in our template.

The list of exported values provided by ngFor directive:

index: Provides the index for current loop iteration. Index starts from 0.

first: Provides Boolean value. It returns true if the element is first in the iteration otherwise false.

last: Provides Boolean value. It returns true if the element is last in the iteration otherwise false.

even: Provides Boolean value. For every index of elements in the iteration, if even then returns true otherwise false.

odd: Provides Boolean value. For every index of elements in the iteration, if odd then returns true otherwise false.

Example:

Create Class, Component and HTML Template.

Find the TypeScript file and HTML template used in our example.

user.ts:

```
export class User {
  constructor(public name: string, public age: number) {
  }
}
```

user.component.ts:

```
import { Component } from '@angular/core';
import { User } from './user';
```

```
@Component({
  selector: 'app-user',
  templateUrl: './user.component.html'
})
```

```
export class UserComponent {
  users: User[] = [
    {name: 'Rajesh', age: 20},
    {name: 'Mahesh', age: 22},
    {name: 'Naresh', age: 25},
    {name: 'Lokesh', age: 28},
    {name: 'Suresh', age: 30}
  ];
}
```

A good feature of **ngFor** is that **whenever there is change in `users` array at runtime, that will be updated in data display iterated by **ngFor****. In our component we have array of `User` class as `users`. We will iterate this array using **ngFor**. If a user is added or removed in `users` array at runtime, that will be updated in display. If at any index in `users` array, the instance of `User` is updated then that will also be updated in display iterated by **ngFor**. Now find the HTML template that will show how to iterate data using **ngFor**.

user.component.html:

```

<b>*ngFor Example</b>
<ul>
  <li *ngFor="let user of users">
    {{user.name}} - {{user.age}}
  </li>
</ul>

<b>ngFor with &lt;ng-template&gt; element</b>
<ul>
  <ng-template ngFor let-user [ngForOf] = "users" let-i = "index">
    <li>Row {{i}} : {{user.name}} - {{user.age}}</li>
  </ng-template>
</ul>

<b>index variable example</b>
<p *ngFor="let user of users; let i = index">
  Row {{i}} : Name: {{user.name}}, Age: {{user.age}}
</p>

<b>first and last variable example</b>
<div *ngFor="let user of users; let i = index; let f=first; let l=last;">
  Row {{i}} : Name: {{user.name}} & Age: {{user.age}}, is first row: {{f}}, is last row: {{l}}
</div>

<br/><b>even and odd variable example</b>
<div *ngFor="let user of users; let i = index; let e=even; let o=odd;">
  Row {{i}} : Name: {{user.name}} & Age: {{user.age}}, is even row: {{e}}, is odd row: {{o}}
</div>

<br/><b>ngFor with component element using *ngFor</b>
<app-emp *ngFor="let user of users" [emp]="user"> </app-emp>
```

```

<br/><br/><b>ngFor with component element using &lt;ng-template&gt; element</b>
<ng-template ngFor let-user [ngForOf] = "users">
  <app-emp [emp] = "user"></app-emp>
</ng-template>
```

employee.component.ts:

```

import {Component, Input} from '@angular/core';
import {User} from './user';
@Component({
  selector: 'app-emp',
  template: `
    <br/> {{emp.name}} - {{emp.age}}
  `
})
export class EmployeeComponent {
  @Input() emp : User;
}
```

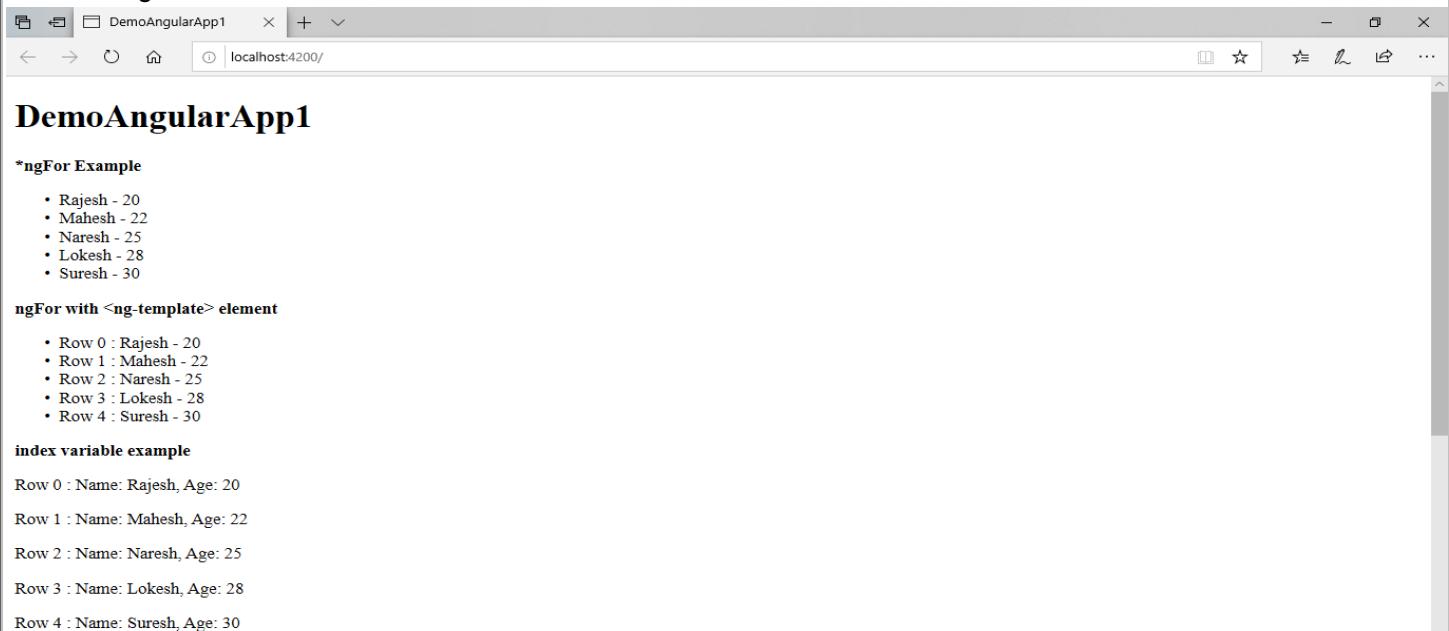
Now, add the UserComponent in AppComponent to use as following:

app.component.html:

```

<h1>{{title}}</h1>
<app-user></app-user>
```

When you run the application using ng serve --o command and you should be able to see the output in your browser as following:



DemoAngularApp1

***ngFor Example**

- Rajesh - 20
- Mahesh - 22
- Naresh - 25
- Lokesh - 28
- Suresh - 30

ngFor with <ng-template> element

- Row 0 : Rajesh - 20
- Row 1 : Mahesh - 22
- Row 2 : Naresh - 25
- Row 3 : Lokesh - 28
- Row 4 : Suresh - 30

index variable example

Row 0 : Name: Rajesh, Age: 20
 Row 1 : Name: Mahesh, Age: 22
 Row 2 : Name: Naresh, Age: 25
 Row 3 : Name: Lokesh, Age: 28
 Row 4 : Name: Suresh, Age: 30

first and last variable example

Row 0 : Name: Rajesh & Age: 20, is first row: true, is last row: false
 Row 1 : Name: Mahesh & Age: 22, is first row: false, is last row: false
 Row 2 : Name: Naresh & Age: 25, is first row: false, is last row: false
 Row 3 : Name: Lokesh & Age: 28, is first row: false, is last row: false
 Row 4 : Name: Suresh & Age: 30, is first row: false, is last row: true

even and odd variable example

Row 0 : Name: Rajesh & Age: 20, is even row: true, is odd row: false
 Row 1 : Name: Mahesh & Age: 22, is even row: false, is odd row: true
 Row 2 : Name: Naresh & Age: 25, is even row: true, is odd row: false
 Row 3 : Name: Lokesh & Age: 28, is even row: false, is odd row: true
 Row 4 : Name: Suresh & Age: 30, is even row: true, is odd row: false

ngFor with component element using *ngFor

Rajesh - 20
 Mahesh - 22
 Naresh - 25
 Lokesh - 28
 Suresh - 30

ngFor with component element using <ng-template> element

Rajesh - 20
 Mahesh - 22
 Naresh - 25
 Lokesh - 28
 Suresh - 30

ngFor with HTML Elements

Here we will discuss ngFor with HTML elements. ngFor directive is used with asterisk (*) and template directive as well as <ng-template> tag. First find how to use ngFor with asterisk (*).

```
<li *ngFor="let user of users">
  {{user.name}} - {{user.age}}
</li>
```

We need to add asterisk (*) as prefix with **ngFor** as ***ngFor**. Here **users** is a collection of **User** instances in our example. To define a variable we need to use **let** keyword. We have defined here one variables **user**. For every iteration an element from **users** collection is assigned to the variable **user** which is accessed within the tag. ***ngFor** can be used with HTML tags such as **<div>**, ****, **<p>** etc...

Find the output.

- Rajesh - 20
- Mahesh - 22
- Naresh - 25
- Lokesh - 28
- Suresh - 30

In case we do not want to use asterisk(*), we can use `<ng-template>` tag for `ngFor` as follows:

```
<b>ngFor with &lt;ng-template&gt; element</b>
<ul>
  <ng-template ngFor let-user [ngForOf] = "users" let-i = "index">
    <li>Row {{i}} : {{user.name}} - {{user.age}}</li>
  </ng-template>
</ul>
```

let-user and let-i : Using prefix **let-**, we declare variable. Here `user` and `i` are variables.

[`ngForOf`] : [] is used to assign a source value into a target within any tag. `ngForOf` is the property of `ngFor` class.

In `ngForOf` we assign our array to iterate.

Find the output.

ngFor with `<ng-template>` element

- Row 0 : Rajesh - 20
- Row 1 : Mahesh - 22
- Row 2 : Naresh - 25
- Row 3 : Lokesh - 28
- Row 4 : Suresh - 30

ngFor with index

index gives the index for current loop iteration.

```
<b>index variable example</b>
<p *ngFor="let user of users; let i = index">
  Row {{i}} : Name: {{user.name}}, Age: {{user.age}}
</p>
```

Find the output.

index variable example

```
Row 0 : Name: Rajesh, Age: 20
Row 1 : Name: Mahesh, Age: 22
Row 2 : Name: Naresh, Age: 25
Row 3 : Name: Lokesh, Age: 28
Row 4 : Name: Suresh, Age: 30
```

Real Time Example of index to display users data in tabular form along with S.No column value:

```
<table border="1" style="border-collapse: collapse;">
<tr>
  <th>S.No</th>
  <th>Name</th>
  <th>Age</th>
</tr>
<tr *ngFor="let user of users; let i = index">
  <td>{{i + 1}}</td>
  <td>{{user.name}}</td>
  <td>{{user.age}}</td>
</tr>
</table>
```

Find the output.

S.No	Name	Age
1	Rajesh	20
2	Mahesh	22
3	Naresh	25
4	Lokesh	28
5	Suresh	30

NgFor with first and last

first and **last** are Boolean values. **first** returns true for the element of first iteration otherwise false. **last** returns true for the element of last iteration otherwise false.

```
<b>first and last variable example</b>
<div *ngFor="let user of users; let i = index; let f=first; let l=last;">
    Row {{i}} : Name: {{user.name}} & Age: {{user.age}}, is first row: {{f}}, is last row: {{l}}
</div>
```

Find the output.

first and last variable example

```
Row 0 : Name: Rajesh & Age: 20, is first row: true, is last row: false
Row 1 : Name: Mahesh & Age: 22, is first row: false, is last row: false
Row 2 : Name: Naresh & Age: 25, is first row: false, is last row: false
Row 3 : Name: Lokesh & Age: 28, is first row: false, is last row: false
Row 4 : Name: Suresh & Age: 30, is first row: false, is last row: true
```

Formatting first & last rows:

```
<style>
    .first { background-color: yellow; }
    .last { background-color: lightpink; }
</style>
<table border="1" style="border-collapse: collapse;">
    <tr>
        <th>S.No</th>
        <th>Name</th>
        <th>Age</th>
    </tr>
    <tr *ngFor="let user of users; let i = index; let first = first; let last = last" [ngClass]="{ first: first, last: last }">
        <td>{{i + 1}}</td>
        <td>{{user.name}}</td>
        <td>{{user.age}}</td>
    </tr>
</table>
```

Find the output

S.No	Name	Age
1	Rajesh	20
2	Mahesh	22
3	Naresh	25
4	Lokesh	28
5	Suresh	30

This will add a CSS class named **first** to the first element of the list, and a CSS class named **last** to the last element of the list.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

NgFor with even and odd:

even and **odd** are Boolean values. **even** returns true for element if iteration number is even otherwise false. **odd** returns true for element if iteration number is odd otherwise false.

```
<b>even and odd variable example</b>
<div *ngFor="let user of users; let i = index; let e=even; let o=odd;">
  Row {{i}} : Name: {{user.name}} & Age: {{user.age}}, is even row: {{e}}, is odd row: {{o}}
</div>
```

Find the output.

even and odd variable example

Row 0 : Name: Rajesh & Age: 20, is even row: true, is odd row: false
 Row 1 : Name: Mahesh & Age: 22, is even row: false, is odd row: true
 Row 2 : Name: Naresh & Age: 25, is even row: true, is odd row: false
 Row 3 : Name: Lokesh & Age: 28, is even row: false, is odd row: true
 Row 4 : Name: Suresh & Age: 30, is even row: true, is odd row: false

Formatting odd & even rows:

Another very common functionality needed when building tables is to be able to stripe a table by adding a different css class to the even or odd rows.

Let's say that to the above table we want to add a CSS class **even** if the row is even and the CSS class **odd** if the row is odd.

We can use the odd & even values to format the odd & even rows alternatively. To do that create two local variable odd & even. Assign the values of odd & even values to these variables using the let statement. Then use this to change the class name to either odd or even as following:

```
<style>
  .odd { background-color: red; color:white; }
  .even { background-color: blue; color:yellow; }
</style>

<table border="1" style="border-collapse: collapse;">
  <tr>
    <th>S.No</th>
    <th>Name</th>
    <th>Age</th>
  </tr>
  <tr *ngFor="let user of users; let i = index; let odd = odd; let even = even" [ngClass]="{ odd: odd, even: even }">
    <td>{{i + 1}}</td>
    <td>{{user.name}}</td>
    <td>{{user.age}}</td>
  </tr>
</table>
```

Find the output.

S.No	Name	Age
1	Rajesh	20
2	Mahesh	22
3	Naresh	25
4	Lokesh	28
5	Suresh	30

ngFor with Component Property Binding:

Here we will provide example of **NgFor** using component element in component property binding. We perform component property binding between two components. There will be a parent component and a child component. In component property binding we copy a component property value from parent to child component property.

Here in our example parent component is **user.component.ts**. Now we will create a child component as follows:

employee.component.ts:

```
import {Component, Input} from '@angular/core';
import {User} from './user';
@Component({
  selector: 'app-emp',
  template:
    `  
 {{emp.name}} - {{emp.age}}
```

)

```
export class EmployeeComponent {
  @Input() emp : User;
}
```

We will use **ngFor** with component property binding in **user.component.html** as follows.

```
<b>ngFor with component element using *ngFor</b>
<app-emp *ngFor="let user of users" [emp]="user"> </app-emp>
```

For each and every elements of **users**, component element will execute child component. For every iteration the value of **emp** will be assigned by **user**. Output will be as given below.

ngFor with component element using *ngFor

```
Rajesh - 20
Mahesh - 22
Naresh - 25
Lokesh - 28
Suresh - 30
```

Expanding *ngFor into template:

We will understand how angular transform the ***ngFor** into **template**. In our example we are using ***ngFor** as given below.

```
<app-emp *ngFor="let user of users" [emp]="user"> </app-emp>
```

Find ***ngFor** transformation into template.

Angular expands ***ngFor** into **<ng-template>** element

```
<b>ngFor with component element using <ng-template> element</b>
<ng-template ngFor let-user [ngForOf]="users">
  <app-emp [emp]="user"></app-emp>
</ng-template>
```

It is up to our choice to use **<ng-template>** element in our code in place of *****

In all the both cases that is ***** and **<ng-template>** element, output will be the same.

ngFor with component element using <ng-template> element

```
Rajesh - 20
Mahesh - 22
Naresh - 25
Lokesh - 28
Suresh - 30
```

ngFor Change Propagation:

For any change in contents of iteration, `ngFor` performs the corresponding changes to DOM.

So, when the contents of the iterator changes, `ngFor` makes the corresponding changes to DOM as given below.

1. If an item is added then new instance of template is added in DOM.
2. If an item is removed then its respective template is removed from DOM.
3. If items are reordered, respective templates are reordered in DOM.
4. If no change for an item, its respective template in DOM will be unchanged.

Angular uses object identity to maintain the entire above task of change propagation. If we want that Angular should not use object identity for change propagation but use user provided identity, then we can achieve it using `trackBy`. It assigns a function that accepts `index` and `item` as function arguments.

We can use `trackBy` with HTML elements as following.

```
<li *ngFor="let person of persons; trackBy: trackByFn"> -- </li>
```

`trackByFn` can be defined to return an identity as following:

```
trackByFn(index: number, item: any) {
    // return any id
}
```

Why use trackBy with ngFor directive:

`ngFor` directive may perform poorly with large lists.

A small change to the list like, adding a new item or removing an existing item may trigger several DOM manipulations.

Let's look at an example, suppose we are having an "Employee Component" in the project.

Now consider this below code in `employee.component.ts` file.

Note:

1. The constructor() initializes the "employees" property with 4 employee objects.
2. `getEmployees()` method returns another list of 5 employee objects (The 4 existing employees plus a new employee object)

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-employee',
  templateUrl: './employee.component.html',
  styleUrls: ['./employee.component.css']
})
export class EmployeeComponent {
  employees: any[];
  constructor() {
    this.employees = [
      {EmpId: 101, EmpName: "Smith", EmpJob: "Programmer", EmpSalary: 18000, DeptName: "IT"},
      {EmpId: 102, EmpName: "David", EmpJob: "HR Manager", EmpSalary: 25000, DeptName: "HR"},
      {EmpId: 103, EmpName: "Peter", EmpJob: "Sales Manager", EmpSalary: 22000, DeptName: "Sales"},
      {EmpId: 104, EmpName: "Martin", EmpJob: "Marketing Manager", EmpSalary: 28000, DeptName: "Marketing"}
    ];
  }
  getEmployees(): void {
    this.employees = [
      {EmpId: 101, EmpName: "Smith", EmpJob: "Programmer", EmpSalary: 18000, DeptName: "IT"},
      {EmpId: 102, EmpName: "David", EmpJob: "HR Manager", EmpSalary: 25000, DeptName: "HR"},
      {EmpId: 103, EmpName: "Peter", EmpJob: "Sales Manager", EmpSalary: 22000, DeptName: "Sales"},
      {EmpId: 104, EmpName: "Martin", EmpJob: "Marketing Manager", EmpSalary: 28000, DeptName: "Marketing"},
      {EmpId: 105, EmpName: "Fleming", EmpJob: "Finance Manager", EmpSalary: 35000, DeptName: "Finance"}
    ];
  }
}
```

Now look at this code in employee.component.html

```

<table border="1" style="border-collapse: collapse; width: 100%;">
  <thead>
    <tr>
      <th>Emp Id</th>
      <th>Emp Name</th>
      <th>Emp Job</th>
      <th>Emp Salary</th>
      <th>Dept Name</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let employee of employees;">
      <td>{{employee.EmpId}}</td>
      <td>{{employee.EmpName}}</td>
      <td>{{employee.EmpJob}}</td>
      <td>{{employee.EmpSalary}}</td>
      <td>{{employee.DeptName}}</td>
    </tr>
    <tr *ngIf="!employees || employees.length==0">
      <td colspan="5" style="color: red;">
        No Employees to Display !!!
      </td>
    </tr>
  </tbody>
</table>
<br />
<button (click)='getEmployees()'>Refresh Employees</button>
```

Now let's check what we are exactly doing. At the moment we are not using trackBy with ngFor directive. So what above code will do is that:

- When the page initially loads we see the 4 employees.
- When we click “Refresh Employees” button we see the fifth employee also.
- It will look like it has just added the additional row for the fifth employee. But that's not true, it effectively destroyed all the <tr> and <td> elements of all the employees and recreated them.
- To confirm this launch your browser with developer tools by pressing F12.
- Then Click on the “Elements” tab and expand the <table> and then <tbody> elements.
- At this point again click the “Refresh Employees” button and you will notice all the <tr>elements are briefly highlighted. This indicates that they are destroyed and recreated.

Initial page load screen with 4 employees objects:

Emp Id	Emp Name	Emp Job	Emp Salary	Dept Name
101	Smith	Programmer	18000	IT
102	David	HR Manager	25000	HR
103	Peter	Sales Manager	22000	Sales
104	Martin	Marketing Manager	28000	Marketing

When we click on "Refresh Employees" button:

Emp Id	Emp Name	Emp Job	Emp Salary	Dept Name
101	Smith	Programmer	18000	IT
102	David	HR Manager	25000	HR
103	Peter	Sales Manager	22000	Sales
104	Martin	Marketing Manager	28000	Marketing
105	Fleming	Finance Manager	35000	Finance

We are adding only one record, but it will render the entire list again.

In the above screen, you can check that when the button is clicked it will add the fifth object of an employee but along with that it will destroy previously added objects and recreate them. You can see in your browser that they all are highlighted.

This happened because by default angular keeps track of these Employee objects using their object references and when we click the button we are returning new object references(by calling getEmployees() method). So angular does not know whether it is old objects collection or not and that's why it destroys old list and then recreates them.

This can cause a problem when we are dealing with a large number of objects or list and performance issues will arise. So to avoid this we can use **trackBy** with ngFor directive.

So now we will add a trackBy function in **employee.component.ts**.

The **trackBy function** takes the index and the current item as arguments and returns the unique identifier by which that item should be tracked. In our case, we are tracking by Employee code.

```
trackByEmpCode(index: number, employee: any): number {
    return employee.EmpId;
}
```

updated **employee.component.ts**

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-employee',
  templateUrl: './employee.component.html',
  styleUrls: ['./employee.component.css']
})
export class EmployeeComponent {
  employees: any[];
  constructor() {
    this.employees = [
      { EmpId: 101, EmpName: "Smith", EmpJob: "Programmer", EmpSalary: 18000, DeptName: "IT" },
      { EmpId: 102, EmpName: "David", EmpJob: "HR Manager", EmpSalary: 25000, DeptName: "HR" },
      { EmpId: 103, EmpName: "Peter", EmpJob: "Sales Manager", EmpSalary: 22000, DeptName: "Sales" },
      { EmpId: 104, EmpName: "Martin", EmpJob: "Marketing Manager", EmpSalary: 28000, DeptName: "Marketing" }
    ];
  }
  getEmployees(): void {
    this.employees = [
      { EmpId: 101, EmpName: "Smith", EmpJob: "Programmer", EmpSalary: 18000, DeptName: "IT" },
      { EmpId: 102, EmpName: "David", EmpJob: "HR Manager", EmpSalary: 25000, DeptName: "HR" },
      { EmpId: 103, EmpName: "Peter", EmpJob: "Sales Manager", EmpSalary: 22000, DeptName: "Sales" },
      { EmpId: 104, EmpName: "Martin", EmpJob: "Marketing Manager", EmpSalary: 28000, DeptName: "Marketing" },
      { EmpId: 105, EmpName: "Fleming", EmpJob: "Finance Manager", EmpSalary: 35000, DeptName: "Finance" }
    ];
  }
  trackByEmpCode(index: number, employee: any): number {
    return employee.EmpId;
  }
}
```

And also make the following change in employee.component.html

Notice that along with ngFor we also specified trackBy.

```
<tr *ngFor="let employee of employees; trackBy:trackByEmpCode">
```

employee.component.html:

```
<table border="1" style="border-collapse: collapse; width: 100%; ">
  <thead>
    <tr>
      <th>Emp Id</th>
      <th>Emp Name</th>
      <th>Emp Job</th>
      <th>Emp Salary</th>
      <th>Dept Name</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let employee of employees; trackBy:trackByEmpCode">
      <td>{{employee.EmpId}}</td>
      <td>{{employee.EmpName}}</td>
      <td>{{employee.EmpJob}}</td>
      <td>{{employee.EmpSalary}}</td>
      <td>{{employee.DeptName}}</td>
    </tr>
    <!--Or Using <ng-template>-->
    <ng-template ngFor let-employee [ngForOf]="employees"
      [ngForTrackBy]="trackByEmpCode">
      <tr>
        <td>{{employee.EmpId}}</td>
        <td>{{employee.EmpName}}</td>
        <td>{{employee.EmpJob}}</td>
        <td>{{employee.EmpSalary}}</td>
        <td>{{employee.DeptName}}</td>
      </tr>
    </ng-template>
    <tr *ngIf="!employees || employees.length==0">
      <td colspan="5" style="color: red;">
        No Employees to Display !!!
      </td>
    </tr>
  </tbody>
</table>
<br />
<button (click)='getEmployees()'>Refresh Employees</button>
```

Now run the application:

The screenshot shows a browser window titled "DemoAngularApp1" displaying an "Employee List" page at "localhost:4200". The page features a table with five columns: Emp Id, Emp Name, Emp Job, Emp Salary, and Dept Name. A new row with values (105, Fleming, Finance Manager, 35000, Finance) is highlighted with a red box. Below the table is a button labeled "Refresh Employees". To the right, the browser's developer tools are open, showing the DOM tree. A red box highlights the entire `<tbody>` element. Inside this box, another red box highlights a specific `<tr>` element representing the new employee entry. The developer tools also show the CSS styles applied to the page, including a style for the `.ngcontent` pseudo-class and a style for the `tbody` element.

Employee List

Emp Id	Emp Name	Emp Job	Emp Salary	Dept Name
101	Smith	Programmer	18000	IT
102	David	HR Manager	25000	HR
103	Peter	Sales Manager	22000	Sales
104	Martin	Marketing Manager	28000	Marketing
105	Fleming	Finance Manager	35000	Finance

Refresh Employees

Now it will add only `<tr>`, not entire list.

```
<!DOCTYPE html>
...<html lang="en"> == $0
  <head></head>
  <body>
    <app-root _ngcontent-lfx-c0 ng-version="8.2.3">
      <h1 _ngcontent-lfx-c0>Employee List</h1>
      <app-employee _ngcontent-lfx-c0 _ngcontent-lfx-c1>
        <table _ngcontent-lfx-c1 border="1" style="border-collapse: collapse; width: 100%;">
          <thead _ngcontent-lfx-c1></thead>
          <tbody _ngcontent-lfx-c1>
            <!--bindings={
              "ng-reflect-ng-for-of": "[object Object],[object Object]",
              "ng-reflect-ng-for-track-by": "trackByEmpCode(index, employee)"
            }-->
            <tr _ngcontent-lfx-c1></tr>
            <tr _ngcontent-lfx-c1></tr>
            <tr _ngcontent-lfx-c1></tr>
            <tr _ngcontent-lfx-c1></tr>
            <tr _ngcontent-lfx-c1></tr>
            <!--bindings={
              "ng-reflect-ng-if": "false"
            }-->
          </tbody>
        </table>
        <br _ngcontent-lfx-c1>
        <button _ngcontent-lfx-c1>Refresh Employees</button>
      </app-employee>
    </app-root>
    <script src="runtime.js" type="module"></script>
    <script src="polyfills.js" type="module"></script>
    <script src="styles.js" type="module"></script>
    <script src="vendor.js" type="module"></script>
    <script src="main.js" type="module"></script>
  </body>
</html>
```

At this point, run the application and check the developer tools. When you click “**Refresh Employees**” the first time, only the row of the fifth employee is highlighted indicating only that `<tr>` element is added. When you click furthermore, nothing is highlighted. It means that none of the `<tr>` elements are destroyed or added as the employees collection has not changed. Even now we get different object references when we click “**Refresh Employees**” button, but as Angular is now **tracking** employee objects using the employee code instead of object references, the respective DOM elements are not affected.

In case we are fetching data from server, we should certainly use `trackBy` or `ngForTrackBy` to change the default behavior (i.e. change propagation by object identity). Suppose a scenario that iterator of `ngFor` is produced from RPC to server and that RPC is re-run then object identity may change. So `ngFor` will re-draw the template even as there is no change in any object value. That impacts the performance and hence in this case we should use `trackBy` or `ngForTrackBy` function to change the default tracking id.



ngSwitch Directive:

ngSwitch is similar to switch statement of C# and Java.

NgSwitch is an angular structural directive that displays one element from a possible set of elements based on some condition.

ngSwitch is a directive which is bound to an expression. ngSwitch is used to display one element tree from a set of many element trees based on some condition. It uses three keywords as follows:

ngSwitch: It is applied to the container element with a switch expression. We bind an expression to it that returns the switch value. It uses property binding.

ngSwitchCase: Defines the element for matched value. We need to prefix it with asterisk (*). 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: We can also apply the ngSwitchDefault directive. It defines the default element when there is no match. We need to prefix it with asterisk (*). 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.

ngSwitch uses ngSwitchCase and ngSwitchDefault. ngSwitch uses ngSwitchCase keyword to define a set of possible element trees and ngSwitchDefault is used to define default value when expression condition does not match to any element tree defined by ngSwitchCase. ngSwitch is used as property binding such as [ngSwitch] with bracket []. To define possible set of elements, we need to add asterisk (*) as prefix to the switch case keywords as *ngSwitchCase and *ngSwitchDefault. Whenever ngSwitch finds a match evaluated by expression then the respective element defined by ngSwitchCase is added to DOM and if no match is found then the element defined by ngSwitchDefault is added to DOM.

Syntax of ngSwitch:

```
<container_element [ngSwitch]="switch_expression">
  <inner_element *ngSwitchCase="match_expression_1">...</inner_element>
  <inner_element *ngSwitchCase="match_expression_2">...</inner_element>
  <inner_element *ngSwitchCase="match_expression_3">...</inner_element>
  <inner_element *ngSwitchDefault>...</element>
</container_element>
```

ngSwitch Examples:

Component Class (app.component.ts)

Create a variable/property named num in your component class:

```
num: number = 0;
```

Component Template (app.component.html)

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

Now let us examine the code in detail

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

This is a simple input box bound to **num** variable in the component

`<div [ngSwitch]="num">`

The ngSwitch directive is attached to the div element. It is then bound to the expression "num". The expression is evaluated and matched to the expression bound to ngSwitchCase.

`<div *ngSwitchCase="'1'">One</div>`

Next, we have few ngSwitchCase attached to the div element. If is then bound to the expression "1", "2" etc. This expression is evaluated and compared with the expression bound to ngSwitch (i.e. "num"). If it matches, the div element is shown to the user, else it will be removed from the DOM

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

Finally, we have the ngSwitchDefault directive, which is attached to the element div, but not to bound any expression. If none of the ngSwitchCase expressions matches, then the div element attached to ngSwitchDefault is shown and all other elements are destroyed from the DOM.

ngSwitch Example ngSwitch Example

Input Value : Input Value :
This is Default Five

ngSwitch with ngFor and ngClass:

Find the code snippet which is using **ngSwitch** with **ngFor** and **ngClass**. Here **ngSwitch** is using interpolation `{{ }}`.

Component Class (app.component.ts):

Create an array property named **values** in your component class:

`values: number[] = [1,2,3,4,5];`

Component Style (app.component.css):

```
.even {
  color: green;
}
.odd {
  color: red;
}
```

Component Template (app.component.html):

```
<div *ngFor="let value of values">
  Value is {{value}} &
  <span ngSwitch="{{value % 2}}>
    <span *ngSwitchCase="'0'" [ngClass]="'even'">It is Even.</span>
    <span *ngSwitchCase="'1'" [ngClass]="'odd'">It is Odd.</span>
    <span *ngSwitchDefault>Nothing</span>
  </span>
</div>
```

Here **values** is an array defined in component which has numbers. For each iteration we are getting array element in the variable **value**. Now we are using this variable in **ngSwitch** and dividing it by 2. For 0 value the element with CSS class `.even` will be executed and for 1 the element with CSS class `.odd` will be executed.

ngSwitch Example

Value is 1 & It is Odd.
Value is 2 & It is Even.
Value is 3 & It is Odd.
Value is 4 & It is Even.
Value is 5 & It is Odd.

Another Example:

Component Class (app.component.ts):

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

  items: Item[] = [
    {name: 'One', value: 1},
    {name: 'Two', value: 2},
    {name: 'Three', value: 3}
  ];

  selectedValue: number= 1;
}

class Item {
  name: string;
  value: number;
}
```

Component Template: (app.component.html):

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

<select [(ngModel)]="selectedValue">
  <option value="0">Select Option</option>
  <option *ngFor="let item of items;" [value]="item.value">{{item.name}}</option>
</select>
&nbsp;
<span [ngSwitch]="selectedValue">
  <span *ngSwitchCase="'1'">One is Selected</span>
  <span *ngSwitchCase="'2'">Two is Selected</span>
  <span *ngSwitchCase="'3'">Three is Selected</span>
  <span [ngStyle]="{{'color':'red'}}" *ngSwitchDefault>No Option Selected</span>
</span>
```

Output:

ngSwitch Example ngSwitch Example

One is Selected No Option Selected

NgSwitch with Enum:

Here we will discuss how to use **NgSwitch** with TypeScript **enum**. Suppose we have an **enum** as below.

directionEnum.ts:

```
export enum DirectionEnum {
  East, West, North, South
}
```

Now we need to import this **enum** in our component as follows:

```
import {DirectionEnum} from './directionEnum';
```

In our component class we will assign our **DirectionEnum** to a component property as below.

```
direction = DirectionEnum;
```

Here **direction** is a component property that will be used to access **enum** values. Hence we are ready to use **enum** in our angular HTML template as follows:

```
<h1>{{title}}</h1>
<div [ngSwitch]="myDirection">
  <b *ngSwitchCase="direction.East">East Direction</b>
  <b *ngSwitchCase="direction.West">West Direction</b>
  <b *ngSwitchCase="direction.North">North Direction</b>
  <b *ngSwitchCase="direction.South">South Direction</b>
  <b *ngSwitchDefault>No Direction</b>
</div>
```

Here **myDirection** is a component property and this will assign the value to **ngSwitch**.

For example if we assign it as

```
myDirection: DirectionEnum = DirectionEnum.North;
```

Then the following line of code will execute.

```
<b *ngSwitchCase="direction.North">North Direction</b>
```

Output:

ngSwitch Example

North Direction

Expanding *ngSwitchCase and *ngSwitchDefault into <ng-template> element:

Angular expands ***ngSwitchCase** and ***ngSwitchDefault** into **<ng-template>** element.

Expand into **<ng-template>** element:

```
<div [ngSwitch]="myDirection">
  <ng-template [ngSwitchCase]="direction.East"> <b> East Direction</b> </ng-template>
  <ng-template [ngSwitchCase]="direction.West"> <b> West Direction</b> </ng-template>
  <ng-template [ngSwitchCase]="direction.North"> <b>North Direction</b> </ng-template>
  <ng-template [ngSwitchCase]="direction.South"> <b>South Direction</b> </ng-template>
  <ng-template ngSwitchDefault> <b> No Direction</b> </ng-template>
</div>
```

In this way if we do not want to use asterisk (*) we can use **<ng-template>** tag for **ngSwitchCase** and **ngSwitchDefault**.

Here we can understand that **ngSwitch** does not define content. It only controls a collection of templates. That is why angular does not add asterisk (*) as prefix with **ngSwitch**.

Real Time Example:

Component Class (app.component.ts):

```

import { Component } from '@angular/core';
import { DirectionEnum } from './directionEnum';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title:string = 'ngSwitch Example';

  studentList: Array<any> = new Array<any>();

  constructor(){
    this.studentList = [
      { StudentId: 1, Name: 'Smith', Course: 'B.Sc.', Grade: 'A' },
      { StudentId: 2, Name: 'John', Course: 'B.A.', Grade: 'B' },
      { StudentId: 3, Name: 'David', Course: 'B.Com.', Grade: 'A' },
      { StudentId: 4, Name: 'Peter', Course: 'B.Sc.', Grade: 'C' },
      { StudentId: 5, Name: 'Fleming', Course: 'MBA', Grade: 'B' },
      { StudentId: 6, Name: 'Jacob', Course: 'M.Sc.', Grade: 'B' },
      { StudentId: 7, Name: 'Donald', Course: 'MBA', Grade: 'A' },
      { StudentId: 8, Name: 'Martin', Course: 'M.Tech', Grade: 'C' },
      { StudentId: 9, Name: 'Alina', Course: 'MCA', Grade: 'D' },
      { StudentId: 10, Name: 'Ronald', Course: 'B.Tech', Grade: 'A' }
    ];
    //Sort The Array List by Grade Property
    this.studentList.sort((a,b) => (a.Grade > b.Grade) ? 1 : ((a.Grade < b.Grade) ? -1 : 0));
  }
}

```

Template (app.component.html):

```

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

<table border="1" style="width:100%;border-collapse: collapse;">
  <thead>
    <tr>
      <td>Student Id</td>
      <td>Student Name</td>
      <td>Course</td>
      <td>Grade</td>
    </tr>
  </thead>

```

```

<tbody>
  <tr *ngFor="let student of studentList;" [ngSwitch]="student.Grade">
    <td>
      <span *ngSwitchCase="'A'" [ngStyle]="{{'color':'green'}}">{{student.StudentId}}</span>
      <span *ngSwitchCase="'B'" [ngStyle]="{{'color':'blue'}}">{{student.StudentId}}</span>
      <span *ngSwitchCase="'C'" [ngStyle]="{{'color':'orange'}}">{{student.StudentId}}</span>
      <span *ngSwitchDefault [ngStyle]="{{'color':'red'}}">{{student.StudentId}}</span>
    </td>
    <td>
      <span *ngSwitchCase="'A'" [ngStyle]="{{'color':'green'}}">{{student.Name}}</span>
      <span *ngSwitchCase="'B'" [ngStyle]="{{'color':'blue'}}">{{student.Name}}</span>
      <span *ngSwitchCase="'C'" [ngStyle]="{{'color':'orange'}}">{{student.Name}}</span>
      <span *ngSwitchDefault [ngStyle]="{{'color':'red'}}">{{student.Name}}</span>
    </td>
    <td>
      <span *ngSwitchCase="'A'" [ngStyle]="{{'color':'green'}}">{{student.Course}}</span>
      <span *ngSwitchCase="'B'" [ngStyle]="{{'color':'blue'}}">{{student.Course}}</span>
      <span *ngSwitchCase="'C'" [ngStyle]="{{'color':'orange'}}">{{student.Course}}</span>
      <span *ngSwitchDefault [ngStyle]="{{'color':'red'}}">{{student.Course}}</span>
    </td>
    <td>
      <span *ngSwitchCase="'A'" [ngStyle]="{{'color':'green'}}">{{student.Grade}}</span>
      <span *ngSwitchCase="'B'" [ngStyle]="{{'color':'blue'}}">{{student.Grade}}</span>
      <span *ngSwitchCase="'C'" [ngStyle]="{{'color':'orange'}}">{{student.Grade}}</span>
      <span *ngSwitchDefault [ngStyle]="{{'color':'red'}}">{{student.Grade}}</span>
    </td>
  </tr>
</tbody>
</table>

```

Output:



ngSwitch Example

Student Id	Student Name	Course	Grade
1	Smith	B.Sc.	A
3	David	B.Com.	A
7	Donald	MBA	A
10	Ronald	B.Tech	A
2	John	B.A.	B
5	Fleming	MBA	B
6	Jacob	M.Sc.	B
4	Peter	B.Sc.	C
8	Martin	M.Tech	C
9	Alina	MCA	D

Attribute Directives:

Attribute directives manipulate the DOM by changing its behavior and appearance.

We use attribute directives to apply conditional style to elements, show or hide elements or dynamically change the behavior of a component according to a changing property.

As the name suggests, it is used to change the attributes of the existing html element. In Angular there are many built in attribute directives. Some of the most commonly used attribute directives are ngModel, ngClass, ngStyle

Commonly used Attribute Directives:

- ngModel
- ngClass
- ngStyle

ngModel Attribute Directive:

The ngModel attribute directive is used to achieve the two-way data binding. We have already covered ngModel attribute directive while working with Data Binding (Two Way Binding) in Angular earlier. Please refer to it for more details.

Basic Example:

Component Class: (app.component.ts):

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title:string = 'ngModel Directive';
}
```

Bind the Component Class Property to View and vice-versa too using **ngModel** attribute directive (Called Two Way Data Binding).

Template (app.component.html)

```
<b>Title: </b> <input type="text" [(ngModel)]="title" />
<br /><br />
<b>Title Value Is: </b> {{title}}
```

ngModel Directive

Title:

Title Value Is: ngModel Directive

ngClass Attribute Directive:

The ngClass is used to add or remove the CSS classes from an HTML element. So we can say this attribute is used to change the class attribute of the element in DOM or the component to which it has been attached. Using the ngClass one can create dynamic styles in HTML template.

The ngClass attribute directive is used to achieve the class binding. We have already covered ngClass attribute directive while working with Class Binding in Angular earlier. Please refer to it for more details.

Example of ngClass: We can define ngClass in different ways:

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

NgClass with String:

```
<p [ngClass]="'class1'">Using NgClass with String.</p>
```

In case we want to add more than one CSS classes then add it separated by space.

```
<p [ngClass]="'class1 class2'">Using NgClass with String.</p>
```

We can also use `ngClass` with `ngFor` using string as follows. The given CSS classes will be applied to HTML element in each iteration.

```
<div *ngFor="let user of users" [ngClass]="'class1 class2'">
  {{user}}
</div>
```

NgClass with Array:

Here we will discuss how to use `NgClass` with array of CSS classes. Suppose we have two CSS classes named as `.class1` and `.class2`. We need to add our CSS classes within bracket `[]` separated by comma (,) and enclosed by single quotes (').

```
<p [ngClass]=["'class1', 'class2'"]>
  Using NgClass with Array.
</p>
```

`ngClass` with array can also be used with `ngFor` as follows:

```
<div *ngFor="let user of users" [ngClass]=["'class1', 'class2'"]>
  {{user}}
</div>
```

NgClass with Object:

Here we will discuss `NgClass` with object using braces `{ }`. It will be in **key** and **value** pair format separated by colon (:).

key: CSS class name.

value: An expression that returns Boolean value.

Here **key** will be CSS class name and **value** will be an expression that will return Boolean value. CSS will be added at run time in HTML element only when if expression will return **true**. If expression returns **false** then the respective CSS class will not be added. Suppose we have four CSS classes named as `.class1`, `.class2`, `.class3` and `.class4`. Now find the below code as following:

```
<p [ngClass]={"'class1': true, 'class3': false }">
  Using NgClass with Object.
</p>
```

In the above code we are using two CSS classes. The expression value of CSS class `.class1` is **true**. So this CSS class will be added at runtime. Now check next CSS class used in above code that is `.class3` and its expression value is **false**. So this CSS class will be removed at run time from HTML element.

`ngClass` with object can also be used with `ngFor`. Find the code below. Here we are using `index` variable and dividing it by 2 in every iteration. When the result is 0, we are using CSS class as `.class1` and if result is 1 then we are using CSS class as `.class3`.

```
<div *ngFor="let user of users; let i = index">
  <div [ngClass]={"'class1': i%2==0, 'class3':i%2==1}"> {{user}} </div>
</div>
```

Now find the below code. Here we are using `even` variable. This variable returns **true** if iteration number is even otherwise **false**.

```
<div *ngFor="let user of users; let flag = even;">
  <div [ngClass]={"'class1':flag, 'class2':flag, 'class3':!flag, 'class4':!flag}"> {{user}} </div>
</div>
```

In the above code, while iteration if `flag` value is **true** then CSS classes `.class1` and `.class2` will be added to the HTML element otherwise CSS classes `.class3` and `.class4` will be added to HTML element at run time.

NgStyle Attribute Directive:

Angular provides a built-in `ngStyle` attribute to modify the element appearance and behavior. `ngStyle` is used to change the multiple style properties of our HTML elements.

In other words we can say `ngStyle` directive is used to set many inline styles dynamically. Setting styles using `ngStyle` works as **key:value** pair. **key** is the style property name and **value** is the style value. We have already covered `ngStyle` attribute directive also while working with Style Binding in Angular earlier. Please refer to it for more details.

Example:

Component Class (app.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title:string = 'ngStyle Directive Example';
  applyGreenStyle: boolean = true;
  borderStyle = '1px solid black';
  appStyleGreen = {
    'color': 'green',
    'font-weight': 'bold',
    'font-size': '45px',
    'borderBottom': this.borderStyle,
    'padding': '10px'
  };
  appStyleBlue = {
    'color': 'blue',
    'font-weight': 'bold',
    'font-size': '45px',
    'borderBottom': this.borderStyle,
    'padding': '10px'
  };
  ChangeColor(): void {
    this.applyGreenStyle = !this.applyGreenStyle;
  }
}
```

Template (app.component.html):

```
<h1>{{title}}</h1>
<p style="padding: 10px"
 [ngStyle]="{'color': Red,'font-weight': 'Bold','font-size':'35px','borderBottom': borderStyle}">
   Assigning Inline Style to ngStyle
</p>
<p [ngStyle]="appStyleGreen">Set Style from the Component.</p>
<p> <input type="button" (click)="ChangeColor()" value="Change Color" /> </p>
<p [ngStyle]="applyGreenStyle==true?appStyleGreen:appStyleBlue">
   Change the Style based On Condition...
</p>
```

ngStyle Directive Example

Assigning Inline style to ngStyle

set style from the Component.

Change color

Change the Style based On Condition...



Custom Directives:

Angular provides three types of directive: component directive, attribute directive and structural directive. Component directive is used to create HTML template. Attribute directive changes the appearance and behavior of DOM element. Structural directive changes the DOM layout by adding and removing DOM elements. Here we will discuss how to create custom attribute and structural directives.

In angular we create these directives using `@Directive()` decorator. It has a `selector` metadata that defines the custom directive name. Angular also provides built-in directives. The built-in attribute directives are `ngModel`, `ngStyle`, `ngClass` etc. These directives change the appearance and behavior of HTML elements. The built-in structural directives are `ngFor`, `ngIf` and `ngSwitch` etc. These directives can add and remove HTML elements from the DOM layout. Custom directives are created using following syntax:

```
@Directive({
  selector: '[rsnDir]'
})
export class RSNDirective {
```

The directive name is `rsnDir` here. It should be enclosed within bracket `[]`. We can keep directive name as we want but it should be started with your company name or any other keyword but not with Angular keyword such as `ng`.

To behave our directive like attribute directive, we can use `ElementRef` to change appearance. To listen event we can use `@HostListener()` decorator.

To behave our directive like structural directive, we can use `TemplateRef` and `ViewContainerRef`. Now find the complete custom attribute directive and custom structural directive example step by step.

Custom Attribute Directives:

Angular custom attribute is created to change appearance and behavior of HTML element. Find the steps to create custom attribute directive.

1. Create a class decorated with `@Directive()`.
2. Assign the attribute directive name using `selector` metadata of `@Directive()` decorator enclosed with bracket `[]`.
3. Use `ElementRef` class to access DOM to change host element appearance.
4. Use `@Input()` decorator to accept user input in our custom directive.
5. Use `@HostListener()` decorator to listen events in custom attribute directive.
6. Configure custom attribute directive class in application module in the `declarations` metadata of `@NgModule` decorator.

To change appearance of HTML element in DOM, we need to use `ElementRef` within custom directive definition. `ElementRef` can directly access the DOM. We can use it directly without using directive but that can lead to XSS attacks. It is safer to use `ElementRef` within directive definition. Now let us create custom attribute directives.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

1. Creating Simple Attribute Directive:

We are creating a simple attribute directive that will change color and font size of a HTML element.

Directive Class (rsn-default-theme.directive.ts):

```
import { Directive, ElementRef, AfterViewInit } from '@angular/core';

@Directive({
  selector: '[rsnDefaultTheme]'
})
export class RSNDefaultThemeDirective implements AfterViewInit {
  constructor(private elRef: ElementRef) {}
  ngAfterViewInit(): void {
    this.elRef.nativeElement.style.color = 'blue';
    this.elRef.nativeElement.style.fontSize = '20px';
  }
}
```

`AfterViewInit` is the lifecycle hook that is called after a component view has been fully initialized. To use `AfterViewInit`, our class will implement it and override its method `ngAfterViewInit()`. Directives are declared in application module in the same way as we declare component. We need to configure our each and every directive in application module within the `declarations` block of `@NgModule` decorator.

```
import { RSNDefaultThemeDirective } from './rsn-default-theme.directive';
@NgModule({
  ...
  declarations: [
    ...
    RSNDefaultThemeDirective
  ]
})
export class AppModule { }
```

Now we are ready to use our custom directive in our application in any component template.

Custom directive selector name: `rsnDefaultTheme`

Example to use our custom directive:

`<p rsnDefaultTheme> rsnDefaultTheme Directive Example </p>`

The text of the `<p>` element will be blue with font size 20px.

Create Attribute Directive using `@Input()`:

Angular custom directive can also accept input from the user. To accept input within directive we need to declare a property decorated with `@Input()`. We must use property name same as selector name.

If we want to use different property name from selector name then use alias with `@Input()`. Now alias name will be same as selector name. We will create a custom directive that will accept input from the user. Here for the example we will accept color as a user input.

Custom Attribute Directive Example

[rsnDefaultTheme Directive Example](#)

Directive Class (rsn-color.directive.ts):

```
import { Directive, ElementRef, Input, AfterViewInit } from '@angular/core';

@Directive({
  selector: '[rsnColor]'
})
export class RSNCOLORDirective implements AfterViewInit {
  @Input() rsnColor: string;
  constructor(private elRef: ElementRef) {}
  ngAfterViewInit(): void {
    this.elRef.nativeElement.style.color = this.rsnColor;
  }
}
```

Look into the above code, selector name and property name are the same. If we want to use alias we can use as follows:

```
@Input('rsnColor') myColor: string;
```

Custom directive selector name: **rsnColor**

Now we are ready to use our custom directive in our application in any component template.

Example to use our custom directive:

Template (app.component.html)

```
<h4 [rsnColor]="titleColor"> rsnColor Directive Demo using Bracket [] </h4>
<h4 bind-rsnColor="titleColor"> rsnColor Directive Demo using bind- prefix </h4>
<h4 rsnColor="{{titleColor}}> rsnColor Directive Demo using Interpolation </h4>
```

titleColor defined as a property with default color in component class (app.component.ts):

```
titleColor: string = "Red";
```

The text of <h4> will be shown in the user provided color.

Final Output:

Custom Attribute Directive Example using **@Input()**

rsnColor Directive Demo using Bracket []

rsnColor Directive Demo using bind- prefix

rsnColor Directive Demo using Interpolation

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

If we want to accept more than one input then create more than one properties decorated with `@Input()`.

Directive Class (`rsn-custom-theme.directive.ts`):

```
import { Directive, ElementRef, Input, AfterViewInit } from '@angular/core';

@Directive({
  selector: '[rsnCustomTheme]'
})

export class RSNCustomThemeDirective implements AfterViewInit {
  @Input() tcolor: string;
  @Input() tsize: string;
  constructor(private elRef: ElementRef) {}

  ngAfterViewInit(): void {
    this.tcolor = this.tcolor || 'green';
    this.tsize = this.tsize || '20px';
    this.elRef.nativeElement.style.color = this.tcolor;
    this.elRef.nativeElement.style.fontSize = this.tsize;
  }
}
```

Custom directive selector name: `rsnCustomTheme`

Now we are ready to use our custom directive in our application in any component template.

Example to use our custom directive:

```
<div rsnCustomTheme tcolor="blue" tsize="30px">
  rsnCustomTheme Directive Example with Custom Settings
</div>
```

User is providing color and font size as an input. The text of `<div>` tag will be shown with the given color and font size.

Final Output:

Custom Attribute Directive Example using `@Input()` with Multiple Properties:

[rsnCustomTheme Directive Example with Custom Settings](#)

Create Attribute Directive using `@HostListener()`:

If we want to change element appearance in DOM on any event then we need to listen event in our custom directive. To listen event we will use Angular `@HostListener()` decorator in our custom directive. The event name will be assigned to `@HostListener()` decorator.

Here we are creating a custom attribute directive using `@HostListener()` decorator.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Directive Class (default-color-on-event.directive.ts):

```
import { Directive, ElementRef, HostListener } from '@angular/core';

@Directive({
  selector: '[defColOnEvent]'
})
export class DefaultColorOnEventDirective {
  constructor(private elRef: ElementRef) {
  }
  @HostListener('mouseover') onMouseOver() {
    this.changeColor('red');
  }
  @HostListener('mouseleave') onMouseLeave() {
    this.changeColor('black');
  }
  private changeColor(color: string) {
    this.elRef.nativeElement.style.color = color;
  }
}
```

Custom directive selector name: `defColOnEvent`

Now we are ready to use our custom directive in our application in any component template.

Example to use our custom directive:

```
<p defColOnEvent> defColOnEvent Directive Demo </p>
```

The color of text of `<p>` tag will change on mouse event. On mouse over event , color of text will be red and on mouse leave event, the color of text will be black.

We can also use `@HostListener()` with `@Input()` to get user input:

Directive Class:

```
import { Directive, ElementRef, HostListener, Input } from '@angular/core';

@Directive({
  selector: '[dynamicColOnEvent]'
})
export class DynamicColorOnEventDirective {
  @Input('dynamicColOnEvent') dynamicColor: string;
  @Input() defaultValue: string;
  constructor(private elRef: ElementRef) {
  }
  @HostListener('mouseover') onMouseOver() {
    this.changeBackgroundColor(this.dynamicColor || this.defaultValue);
  }
  @HostListener('mouseleave') onMouseLeave() {
    this.changeBackgroundColor('white');
  }
  private changeBackgroundColor(color: string) {
    this.elRef.nativeElement.style.backgroundColor = color;
  }
}
```

Custom directive selector name: `dynamicColOnEvent`

Now we are ready to use our custom directive in our application in any component template.

Example to use our custom directive:

```
<p [dynamicColOnEvent]="myColor" defaultValue="green">
    dynamicColOnEvent Directive Demo
</p>
```

`myColor` defined as a property with default color in component class (`app.component.ts`):

```
myColor: string = "Yellow";
```

On mouse over, the background color will be user provided color and on mouse leave the background color will be white. If we don't accept the user input color then on mouse over, green background color as default will be shown.

Custom Structural Directives:

Structural directive is used to change the DOM layout by adding and removing DOM elements. Find the steps to create custom structural directive.

1. Create a class decorated with `@Directive()`.
2. Assign the structural directive name using `selector` metadata of `@Directive()` decorator enclosed with bracket `[]`.
3. Create a setter method decorated with `@Input()`. We need to take care that the method name should be same as directive name.
4. Configure custom structural directive class in application module in the `declarations` metadata of `@NgModule` decorator.

To create custom structural directive we need to use `TemplateRef` and `ViewContainerRef` etc. that will help to change the DOM layout.

TemplateRef: It represents an embedded template that can be used to instantiate embedded views.

ViewContainerRef: It represents a container where one or more views can be attached.

Now let us create custom structural directives.

1. Structural Directive for if Condition:

We will create a structural directive that will add a layout in DOM for a `true` condition otherwise it will delete it from DOM.

Directive Class (`rsn-if.directive.ts`):

```
import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';
@Directive({
    selector: '[rsnIf]'
})
export class RSNIfDirective {
    constructor( public templateRef: TemplateRef<any>, public viewContainer: ViewContainerRef) {
    }
    @Input() set rsnIf(condition: boolean) {
        if (condition) {
            this.viewContainer.createEmbeddedView(this.templateRef);
        } else {
            this.viewContainer.clear();
        }
    }
}
```

Custom directive selector name: `rsnIf`

Now we are ready to use our custom directive in our application in any component template.

Example to use our custom directive:

```
<div *rsnIf="showRSNIf">
  <b>Hello World!</b>
</div>
<ng-template [rsnIf]="!showRSNIf">
  <div>
    <b>Not Available</b>
  </div>
</ng-template>
```

`showRSNIf` defined as a property with default boolean value either true/false in component class (app.component.ts):

```
showRSNIf: boolean = true;
```

When `showRSNIf` is true then **Hello World!** message will be shown otherwise the message will be **Not Available**.

2. Structural Directive for Loop:

We will create a structural directive that will add an element in DOM layout for the given number of time. It will work as a loop.

Directive Class (`rsn-loop.directive.ts`):

```
import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';
@Directive({
  selector: '[rsnLoop]'
})
export class RSNLoopDirective {
  constructor( public templateRef: TemplateRef<any>,
              public viewContainer: ViewContainerRef) { }
  @Input('rsnLoop') set loop(num: number) {
    for(var i=0; i < num; i++) {
      this.viewContainer.createEmbeddedView(this.templateRef);
    }
  }
}
```

Custom directive selector name: `rsnLoop`

Now we are ready to use our custom directive in our application in any component template.

Example to use our custom directive:

```
<ul>
  <li *rsnLoop="3">
    Hello, Good Morning !!!
  </li>
</ul>
```

The `` tag will be added 3 times in DOM layout.

Custom Structural Directive Example for Loop

- Hello, Good Morning !!!
- Hello, Good Morning !!!
- Hello, Good Morning !!!

3. Structural Directive with setTimeout()

Now we will create a structural directive that will add an element in DOM layout after a given time.

Directive Class (rsn-delay.directive.ts):

```
import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';
@Directive({
  selector: '[rsnDelay]'
})
export class RSNDelayDirective {
  constructor( public templateRef: TemplateRef<any>,
    public viewContainer: ViewContainerRef) { }
  @Input() set rsnDelay(delay: number) {
    this.viewContainer.clear();
    setTimeout(() =>
    {
      this.viewContainer.createEmbeddedView(this.templateRef);
    }, delay);
  }
}
```

Custom directive selector name: **rsnDelay**

Now we are ready to use our custom directive in our application in any component template.

Example to use our custom directive:

```
<div *rsnDelay="delayInSec">
  <b> Hello World! </b>
</div>
```

delayInSec defined as a property with default value as number in component class (app.component.ts):

```
delayInSec number = 500;
```

Once the time given by **delayInSec** is over, the message will be displayed.

Complete Example with Application Component and Module:

Component Class (app.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  txtsize = '25px';
  colors = ['CYAN', 'GREEN', 'RED'];
  myColor = '';
  titleColor='green'

  showRsnIf = false;
  showRsnDelay = false;
  delayInSec = 0;

  changeCondition(flag) {
    this.showRsnIf = flag;
  }
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Template (app.component.html):

```

<h2>Custom Attribute Directives Examples</h2>
<p rsnDefaultTheme> rsnDefaultTheme Directive Demo</p>
<h4 [rsnColor]="titleColor"> rsnColor Directive Demo</h4>
<div rsnCustomTheme> rsnCustomTheme Directive Demo with Default Settings</div>
<div rsnCustomTheme tcolor="blue" tsize="30px">rsnCustomTheme Directive Demo with Custom Settings
</div>
<p defColOnEvent> defColOnEvent Directive Demo </p>
<div>
  <select [(ngModel)] ="myColor">
    <option value='' selected disabled> Select Color </option>
    <option *ngFor = "let color of colors" [value] = "color">
      {{color}}
    </option>
  </select>
  <p [dynamicColOnEvent]="myColor" defaultValue="blue"> dynamicColOnEvent Directive Demo</p>
</div>
<h2>Custom Structural Directives Examples</h2>
<h3>rsnIf Directive</h3>
<div>
  <input type="radio" name="rad1" (click)= "changeCondition(true)"> True
  <input type="radio" name="rad1" [checked]= "!showRsnIf" (click)= "changeCondition(false)"> False
</div>
<br/>
<div *rsnIf="showRsnIf">
  <b>Hello World!</b>
</div>
<ng-template [rsnIf] ="!showRsnIf">
  <div>
    <b>Not Available</b>
  </div>
</ng-template>
<h3>rsnLoop Directive</h3>
<ul>
  <li *rsnLoop="3" >
    Hello, Good Morning !!!
  </li>
</ul>
<h3>rsnDelay Directive</h3>
<div>
  Show message after
  <input type="radio" name="rad2" (click)= "showRsnDelay= true; delayInSec = 2000"> 2 Seconds
  <input type="radio" name="rad2" (click)= "showRsnDelay= true; delayInSec = 5000"> 5 Seconds
  <input type="radio" name="rad2" (click)= "showRsnDelay= true; delayInSec = 7000"> 7 Seconds
</div>
<br/>
<div *rsnIf="showRsnDelay">
  <div *rsnDelay="delayInSec">
    <b> Hello World of Angular !!! </b>
  </div>
</div>
<br />

```

Module (app.module.ts):

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';

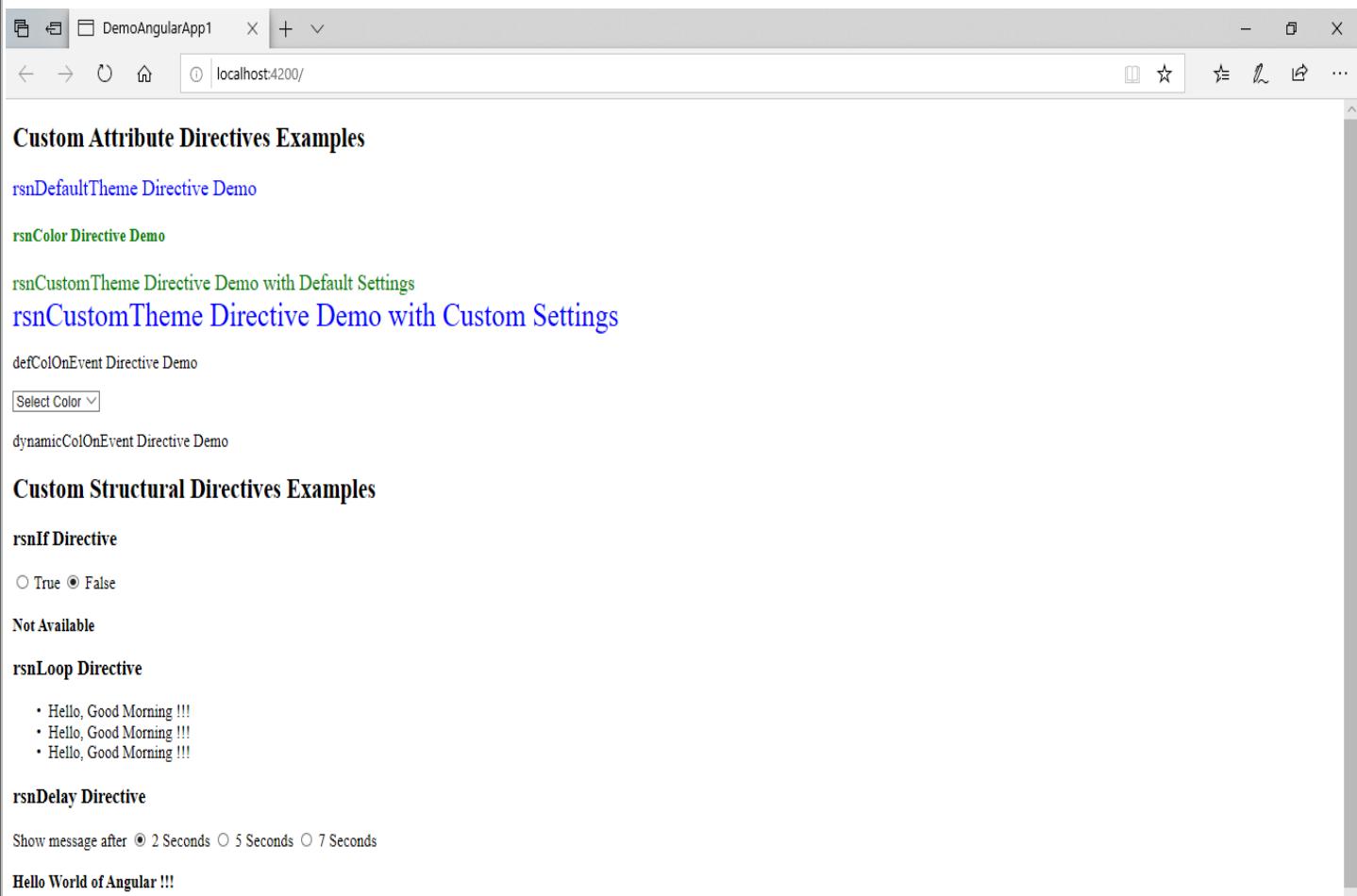
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

import { RSNDefaultThemeDirective } from './rsn-default-theme.directive';
import { RSNCOLORDirective } from './rsn-color.directive';
import { RSNCustomThemeDirective } from './rsn-custom-theme.directive';
import { DefaultColorOnEventDirective } from './default-color-on-event.directive';
import { DynamicColorOnEventDirective } from './dynamic-color-on-event.directive';
import { RSNIfDirective } from './rsn-if.directive';
import { RSNLoopDirective } from './rsn-loop.directive';
import { RSNDelayDirective } from './rsn-delay.directive';

@NgModule({
  declarations: [
    AppComponent,
    RSNDefaultThemeDirective,
    RSNCOLORDirective,
    RSNCustomThemeDirective,
    DefaultColorOnEventDirective,
    DynamicColorOnEventDirective,
    RSNIfDirective,
    RSNLoopDirective,
    RSNDelayDirective
  ],
  imports: [
    BrowserModule,
    FormsModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Final Output:



The screenshot shows a browser window titled "DemoAngularApp1" with the URL "localhost:4200/". The page displays several examples of Angular directives:

- Custom Attribute Directives Examples**
- [rsnDefaultTheme Directive Demo](#)
- [rsnColor Directive Demo](#)
- [rsnCustomTheme Directive Demo with Default Settings](#)
- [rsnCustomTheme Directive Demo with Custom Settings](#)
- [defColOnEvent Directive Demo](#)
- [Select Color](#) (a dropdown menu)
- [dynamicColOnEvent Directive Demo](#)
- Custom Structural Directives Examples**
- [rsnIf Directive](#)
- True False
- Not Available
- [rsnLoop Directive](#)
- Hello, Good Morning !!!
 - Hello, Good Morning !!!
 - Hello, Good Morning !!!
- [rsnDelay Directive](#)
- Show message after 2 Seconds 5 Seconds 7 Seconds
- Hello World of Angular !!!

Major Difference between Component, Attribute and Structural Directives?

Component	Attribute Directive	Structural Directives
Component directive is used to specify the template/html for the DOM Layout	Attribute directive is used to change/modify the behavior of the html element in the DOM Layout	Structural directive used to add or remove the html Element in the DOM Layout
Built in	Built in	Built in
@Component	NgModel, NgStyle, NgClass	*Nglf, *NgFor, *NgSwitch

Pipes in Angular:

The Angular Pipes helps us to format or transform data to display in our template. They are similar to Filters in AngularJS. So the Pipe is another important element of the Angular framework. Using pipe, we can decorate the data as per our desired format in the application. The main purpose of using pipes is to transform data within an HTML template. The Angular also allows us to create the Custom Pipe.

There are many circumstances where we may have to change the appearance of the data before presenting it to the user. The most common examples are dates. That is where **Angular Pipes** comes handy. They take the data as input and transforms that data to get the desired output.

What is Pipe in Details?

When we want to develop any application, we always start the application with a simple task: retrieve data, transform data, and then display the data in front of the user through the user interface. Retrieval of data from any type of data source totally depends on data service providers like Web Services, Web API, etc. So, once data arrives, we can push those raw data values directly to our user interface for viewing by the user. But sometimes, this is not exactly what happens. For example, in most use cases, users prefer to see a date in a simple format like 25/02/2019 rather than the raw string format Monday Feb 25 2019. So, it is clear from the above example that some values require editing before being viewed in the user interface. Also, that same type of transformation might be required by us in many different user interfaces. So, in this scenario, we think about some style type properties that we can create centrally and apply whenever we require it. So, for this purpose, the Angular framework introduced Angular pipes, a definite way to write display – value transformations that we can declare in our HTML.

In Angular, pipes are typescript classes that implement a single function interface, accept an input value with an optional parameter array, and return a transformed value. To perform the value transformation, we can implement any type of business logic as per our requirement. That pipe can be used in any UI to transform that particular type of data as per the desired result.

Basic Concept of Pipes:

Basically, pipes provide a sophisticated and handsome way to perform the tasks within the templates. Pipes make our code clean and structured. In Angular, Pipes accept values from the DOM elements and then return a value according to the business logic implemented within the pipes. So, pipes are one of the great features through which can transform our data into the UI and display. But we need to understand one thing very clearly, that pipes do not automatically update our model value. It basically performs the transformation of data and returns to the component. If we need to update the model data after transformation through pipes, then we need to update our model data manually.

Types of Pipes:

In Angular, pipes are categorised in two types i.e. **Pure Pipes** and **Impure Pipes**.

When you create a new pipe, it is **pure by default**. To make a pipe impure, set its pure flag to **false**.

```
@Pipe({
  name: 'pipeName',
  pure: false // default is set to true.
})
```

Pure pipe is fast.

Angular executes a pure pipe only when it detects a pure change to the input value.

A pure change is either a change to a primitive input value (String, Number, Boolean) or a changed object reference (Array, Date, Object).

A pure pipe is not executed if the input to the pipe is an object and only the property values of the object change but not the reference.

Why are Pure pipes fast?

An input for a pipe can either be a value type (String, Number, Boolean etc.) or a reference type (like Array, Date, Object etc.).

If the input to the pure pipe is a value type. Since value types store data directly in the memory slot comparing if a value type value has changed is very quick.

On the other hand, if the input to the pure pipe is a reference type, it is executed only if the reference of the input object has changed. Checking if an object reference has changed is much faster than checking if each of the object individual property values have changed.

So pure pipes are fast, but using them for filtering data is not a good idea because, the filtering may not work as expected if the source data is updated without a change to the object reference.

One way to make this work is by making the pipe impure. Impure pipes can significantly impact the performance of the application as they run on every change detection cycle. An impure pipe is called often, even for every keystroke or mouse movement, timer tick, and server response.

You have to think very carefully when you make a pipe impure. This is because an impure pipe is processed on every change, even when the source data does not change. They run unnecessarily when not required.

If you have an array with thousands of objects and each object in turn has dozens of properties. Now if we use an impure pipe to filter or sort this array and for some reason on the same page if we are also listening to the mouse move or keystroke event, then the pipe is unnecessarily executed for every mouse move or keystroke which can significantly degrade the performance of the application.

Implement an impure pipe with great care since an expensive, long-running pipe could affect performance and user experience. So you have to think very carefully, before you use an impure pipe in your angular application.

This is the reason, **Angular team recommends not to use pipes to filter and sort data.** Angular recommends to move the filtering and sorting logic into the component itself.

"A pure pipe is only called when Angular detects a change in the value or the parameters passed to a pipe. An impure pipe is called for every change detection cycle no matter whether the value or parameter(s) changes."

Filter vs Pipe:

The concept of the filter is mainly used in the Angular 1.x version. From Angular 2 onwards, Google deprecated the Filter concept and introduced the new concept called Pipe. Now, as per the functionality, filter, and pipes, both are working like the same. But still, there are some differences as below –

Filters are acting just like helpers similar to function where we can pass input and other parameters and it will return us a formatted output value. In the case of a pipe, it works as an operator. It also accepts input value and modifies that value to return the desired output.

Filters cannot handle directly any async type operations. We need to set those values manually. But Pipe can handle Async operations on its own. For these types of operations, we need to use Async type pipes.

Built-in Angular Pipes:

The most commonly used built-in pipes are:

- Uppercase
- Lowercase
- Titlecase
- Slice
- Date
- Decimal
- Percent
- Currency
- JSON
- KeyValue
- I18nSelect
- I18nPlural
- Async

Syntax of Pipes:

The syntax of the Pipe in the HTML template begins with the input value and then followed the pipe symbol (|) and then need to provide the pipe name. The parameters of that pipe can be sent separately by a colon (:). In General, Pipe is working within the HTML only.

`Expression | pipeOperator[:pipeArguments]`

Where

Expression : is the expression, which you want to transform

| : is the Pipe Character

pipeOperator : name of the Pipe

pipeArguments: arguments to the Pipe

Using Pipes:

A pipe takes in data as input and transforms it to a desired output. In this example, we'll use pipes to transform a component's currentDate property into a human-friendly date.

`app.component.ts:`

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

@Component({
  selector: 'app-root',
  template: `
    <p><b>The Current Date in Unformatted Date: </b> {{currentDate}}</p>
    <p><b>The Current Date in Formatted Date: </b> {{currentDate | date}}</p>
  `
})
export class AppComponent {
  currentDate: Date = new Date();
}
```

In the above example, we are taking the current date and transforming it into the easily readable format using the date pipe. We have included the unformatted date format for comparison. The output is as shown below:



Using Pipes

The Current Date in Unformatted Date: Sun Dec 01 2019 01:50:46 GMT+0530 (India Standard Time)

The Current Date in Formatted Date: **Dec 1, 2019**

All pipes work this way.

Passing arguments (Parameterizing a pipe):

We can also pass optional arguments to the pipe. The arguments are added to the pipe using a **colon (:) sign** followed by the value of the argument. If there are multiple arguments separate each of them with the **colon (:) sign**. For example date pipe accepts one optional argument called "format". The "shortDate" is one of the valid value of the format argument, which displays the date in "MM/d/yy" format. The example code is as shown below.

`{{currentDate | date:'shortDate'}}`

The parameter '**shortDate**' displays the date as **12/1/19**



Chaining Pipes:

Pipes can be chained together to make use of multiple pipes in one expression. For example in the following code, the currentDate is passed to the **date Pipe**. The output of Date pipe is then passed to the **uppercase pipe**.

```
{{currentDate | date | uppercase}}
```

Output: DEC 1, 2019

This example - which displays SUNDAY, DECEMBER 1, 2019 - chains the same pipes as above, but passes in a parameter to date as well.

```
{{currentDate | date:'fullDate' | uppercase}}
```

Output: SUNDAY, DECEMBER 1, 2019

uppercase Pipe:

UpperCasePipe is an angular **Pipe API** that belongs to **CommonModule**. As the name suggests, this pipe transforms the text (string) to all uppercase.

We can use uppercase pipe as shown below:

expression | uppercase

- expression is any string value
- uppercase represent UpperCasePipe

lowercase Pipe:

LowerCasePipe is an angular **Pipe API** that belongs to **CommonModule**. As the name suggests, this pipe transforms the text (string) to all lowercase.

We can use lowercase pipe as shown below:

expression | lowercase

- expression is any string value
- lowercase represent LowerCasePipe

titlecase Pipe:

TitleCasePipe is an angular **Pipe API** that belongs to **CommonModule**. It transforms text (string) to title case. Capitalizes the first letter of each word, and transforms the rest of the word to lower case. Words are delimited by any whitespace character, such as a space, tab, or line-feed character.

We can use titlecase pipe as shown below:

expression | titlecase

- expression is any string value
- titlecase represent TitleCasePipe

Examples:

Upper Case Pipe

```
{{'Welcome to Angular Pipes' | uppercase}}
```

WELCOME TO ANGULAR PIPES

Lower Case Pipe

```
{{'Welcome to Angular Pipes' | lowercase}}
```

welcome to angular pipes

Title Case Pipe

```
{{'welcome to angular pipes' | titlecase}}
```

Welcome To Angular Pipes



Component Class File (app.component.ts):

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title = "Welcome to angular Pipes";
}
```

Component html File (app.component.html):

```
<h3>Upper Case: {{title | uppercase}}</h3>
<h3>Lower Case: {{title | lowercase}}</h3>
<h3>Title Case: {{title | titlecase}}</h3>
```

Output:

```
Upper Case: WELCOME TO ANGULAR PIPES
Lower Case: welcome to angular pipes
Title Case: Welcome To Angular Pipes
```

slice Pipe:

SlicePipe is an angular **Pipe** API that belongs to **CommonModule**. It Creates a new List (Array) or String containing a subset (slice) of the string or array. **SlicePipe** uses JavaScript API `Array.prototype.slice()` and `String.prototype.slice()` to perform its task. Slice pipe uses **slice** keyword with pipe operator.

Syntax:

```
array_or_string_expression | slice:start[:end]
```

Where

array_or_string_expression: Expression that will result into an array or a string. This result will be input for slice pipe.

slice: **SlicePipe** API uses **slice** keyword with pipe operator so it is the name of pipe.

start: Starting position/index to slice given array or string to return as subset.

1. If **start** index is **positive**, slice pipe will return the elements at **start** index from start and the elements after in array or string expression.

2. If **start** index is **negative**, slice pipe will return the elements at **start** index from end and the elements after in array or string expression.

3. If **start** index is **positive** and **greater than the size** of string or array expression then slice pipe will return **empty**.

4. If **start** index is **negative** and **greater than the size** of string or array expression then slice pipe will return **complete array or string**.



end: Ending index to slice given array or string to return as subset.

1. If **end** index has **not** been provided then slice pipe will return elements till end.
2. If **end** index is **positive** then slice pipe will return all elements before **end** index from the start of the array or string expression.
3. If **end** index is **negative** then slice pipe will return all elements before **end** index from the end of the array or string expression.

Examples:

SlicePipe using String Expression:

Suppose expression is a string as given below.

```
myStr: string = "abcdefghijklk";
```

We have given a number per character to our input string to understand start and end index. Index starts from 0.

0	1	2	3	4	5	6	7	8	9	10
a	b	c	d	e	f	g	h	i	j	k
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

1. Find the slice pipe with positive **start** and positive **end** index.

```
{{myStr | slice:3:7}}
```

Output:

defg

In our example we have following indexes.

start = 3

end = 7

Slice pipe will return substring starting from index **3** i.e. character **d** and will include all characters before index **7** i.e. up to **g**. The character at end index will not be included in the output substring.

2. Find the slice pipe with positive **start** and negative **end** index.

```
{{myStr | slice:3:-2}}
```

Output:

defghi

In our example we have following indexes.

start = 3

end = -2

Slice pipe will return substring starting from index **3** i.e. character **d** and will include all characters before index **-2** i.e. up to **i**.

3. Slice pipe with positive **start** index only.

```
{{myStr | slice:6}}
```

Output:

ghijk

In our example we have index as follows.

start = 6

In the output substring there are all the characters starting from index 6 i.e. g up to end.

4. Slice pipe with negative **start** index only.

```
{{myStr | slice:-6}}
```

Output:

fghijk

In our example we have index as follows.

start = -6

In the output substring there are all the characters starting from index -6 i.e. f up to end.

Now find the component used in our example for slice pipe using string expression.

Component Class (app.component.ts):

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

@Component({
  selector: 'app-root',
  template: `
    <h3>String Example</h3>
    {{myStr | slice:3:7}} <br/>
    {{myStr | slice:3:-2}} <br/>
    {{myStr | slice:6}} <br/>
    {{myStr | slice:-6}} <br/>
  `
})

export class AppComponent {
  myStr: string = "abcdefghijkl";
}
```

Output:

String Example

defg

defghi

ghijk

fghijk



SlicePipe using Array Expression

Now we will discuss slice pipe with array expression. In our example we have an array as follows.

myArray: number[] = [11, 22, 33, 44, 55, 66, 77, 88];

Find the index detail:

0	1	2	3	4	5	6	7
11	22	33	44	55	66	77	88
-8	-7	-6	-5	-4	-3	-2	-1

The element at **start** index is included and the element at **end** index is excluded in the output.

1.

{{myArray | slice:3:6}}

Output

44,55,66

2.

{{myArray | slice:2:-4}}

Output

33,44

3.

{{myArray | slice:5}}

Output

66,77,88

4.

{{myArray | slice:-5}}

Output

44,55,66,77,88

Now find the component used in our example for array expression.

Component Class (app.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h3>Array Example</h3>
    {{myArray | slice:3:6}} <br/>
    {{myArray | slice:2:-4}} <br/>
    {{myArray | slice:5}} <br/>
    {{myArray | slice:-5}} <br/>
    <ul>
      <li *ngFor="let num of myArray | slice:2:5">
        {{num}}
      </li>
    </ul>
  `
})
export class AppComponent {
  myArray: number[] = [11, 22, 33, 44, 55, 66, 77, 88];
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Output:



Array Example

```
44,55,66
33,44
66,77,88
44,55,66,77,88
```

- 33
- 44
- 55

Look into the below code:

```
<li *ngFor="let num of myArray | slice:2:5">
    {{num}}
</li>
```

Here **ngFor** will work on the elements of the subset obtained from

myArray | slice:2:5

That is [33,44,55]

date Pipe:

The date pipe formats the date according to locale rules. **DatePipe** provides different date formats that can be predefined date formats as well as custom date formats. **DatePipe** relates to **CommonModule**. The date input can be given as date object, milliseconds or an ISO (International Organization for Standardization) date string. **DatePipe** is a **Pipe** API that works using pipe operator (|). On the left side of the pipe, we place date expression and on the right side we place required date formats.

The syntax of the date pipe is as shown below:

date_expression | date[:format]

Where

A. **date_expression**: The values of date expression are the followings.

1. Date object
2. Date in milliseconds as Number
3. An ISO String

B. **date**: It is the name of the pipe

C. **:format** : Date format can be predefined values as well as custom values.



Predefined Date Format Options:

- 'short' : equivalent to 'M/d/yy, h:mm a' (e.g. 6/15/15, 9:03 AM)
- 'medium' : equivalent to 'MMM d, y, h:mm:ss a' (e.g. Jun 15, 2015, 9:03:01 AM)
- 'long' : equivalent to 'MMMM d, y, h:mm:ss a z' (e.g. June 15, 2015 at 9:03:01 AM GMT+1)
- 'full' : equivalent to 'EEEE, MMMM d, y, h:mm:ss a zzzz' (e.g. Monday, June 15, 2015 at 9:03:01 AM GMT+01:00)
- 'shortDate' : equivalent to 'M/d/yy' (e.g. 6/15/15)
- 'mediumDate' : equivalent to 'MMM d, y' (e.g. Jun 15, 2015)
- 'longDate' : equivalent to 'MMMM d, y' (e.g. June 15, 2015)
- 'fullDate' : equivalent to 'EEEE, MMMM d, y' (e.g. Monday, June 15, 2015)
- 'shortTime' : equivalent to 'h:mm a' (e.g. 9:03 AM)
- 'mediumTime' : equivalent to 'h:mm:ss a' (e.g. 9:03:01 AM)
- 'longTime' : equivalent to 'h:mm:ss a z' (e.g. 9:03:01 AM GMT+1)
- 'fullTime' : equivalent to 'h:mm:ss a zzzz' (e.g. 9:03:01 AM GMT+01:00)

Custom Date Format Options:

You can construct a format string using symbols to specify the components of a date-time value, as described in the following table. Format details depend on the locale. Fields marked with (*) are only available in the extra data set for the given locale.

Field Type	Format	Description	Example Value
Year	y	Numeric: minimum digits	2, 20, 201, 2017, 20173
	yy	Numeric: 2 digits + zero padded	02, 20, 01, 17, 73
	yyy	Numeric: 3 digits + zero padded	002, 020, 201, 2017, 20173
	yyyy	Numeric: 4 digits or more + zero padded	0002, 0020, 0201, 2017, 20173
Month	M	Numeric: 1 digit	9, 12
	MM	Numeric: 2 digits + zero padded	09, 12
	MMM	Abbreviated	Sep
	MMMM	Wide	September
	MMMMM	Narrow	S
Month standalone	L	Numeric: 1 digit	9, 12
	LL	Numeric: 2 digits + zero padded	09, 12
	LLL	Abbreviated	Sep
	LLLL	Wide	September
	LLLLL	Narrow	S

Week of year	w	Numeric: minimum digits	1... 53
	ww	Numeric: 2 digits + zero padded	01... 53
Week of month	W	Numeric: 1 digit	1... 5
Day of month	d	Numeric: minimum digits	1
	dd	Numeric: 2 digits + zero padded	01
Week day	E, EE & EEE	Abbreviated	Tue
	EEEE	Wide	Tuesday
	EEEEEE	Narrow	T
	EEEEEEE	Short	Tu
Period	a, aa & aaa	Abbreviated	am/pm or AM/PM
	aaaa	Wide (fallback to a when missing)	ante meridiem/post meridiem
	aaaaa	Narrow	a/p
Period*	B, BB & BBB	Abbreviated	mid.
	BBBB	Wide	am, pm, midnight, noon, morning, afternoon, evening, night
	BBBBB	Narrow	md
Period standalone*	b, bb & bbb	Abbreviated	mid.
	bbbb	Wide	am, pm, midnight, noon, morning, afternoon, evening, night
	bbbbb	Narrow	md
Hour 1-12	h	Numeric: minimum digits	1, 12
	hh	Numeric: 2 digits + zero padded	01, 12
Hour 0-23	H	Numeric: minimum digits	0, 23
	HH	Numeric: 2 digits + zero padded	00, 23
Minute	m	Numeric: minimum digits	8, 59
	mm	Numeric: 2 digits + zero padded	08, 59

Second	s	Numeric: minimum digits	0... 59
	ss	Numeric: 2 digits + zero padded	00... 59
Fractional seconds	S	Numeric: 1 digit	0... 9
	SS	Numeric: 2 digits + zero padded	00... 99
	SSS	Numeric: 3 digits + zero padded (= milliseconds)	000... 999
Zone	z, zz & zzz	Short specific non location format (fallback to O)	GMT-8
	zzzz	Long specific non location format (fallback to OOOO)	GMT-08:00
	Z, ZZ & ZZZ	ISO8601 basic format	-0800
	ZZZZ	Long localized GMT format	GMT-8:00
	ZZZZZ	ISO8601 extended format + Z indicator for offset 0 (= XXXXX)	-08:00
	O, OO & OOO	Short localized GMT format	GMT-8
	OOOO	Long localized GMT format	GMT-08:00

Date Expression:

Date expression can be date object declared as below.

```
currentDate = Date.now();
```

We use **DatePipe** as follows with interpolation:

```
{{currentDate | date : 'short'}}
```

Output: 12/1/2019, 6:00 AM

Date expression can be in milliseconds.

```
numDate = 1575160445377
```

and we use it as below:

```
{{numDate | date : 'short'}}
```

Output: 12/1/2019, 6:04 AM

Date expression can also be an ISO string.

```
strDate = 'Sun Dec 01 2019 06:10:15 GMT+0530';
```

and we use it as follows.

```
{{strDate | date : 'short'}}
```

Output: 12/1/2019, 6:10 AM

We can also use milliseconds and date string as date expression directly as follows.

```
{{1575160445377 | date : 'short'}}
```

```
{{'Sun Dec 01 2019 06:10:15 GMT+0530' | date : 'short'}}
```

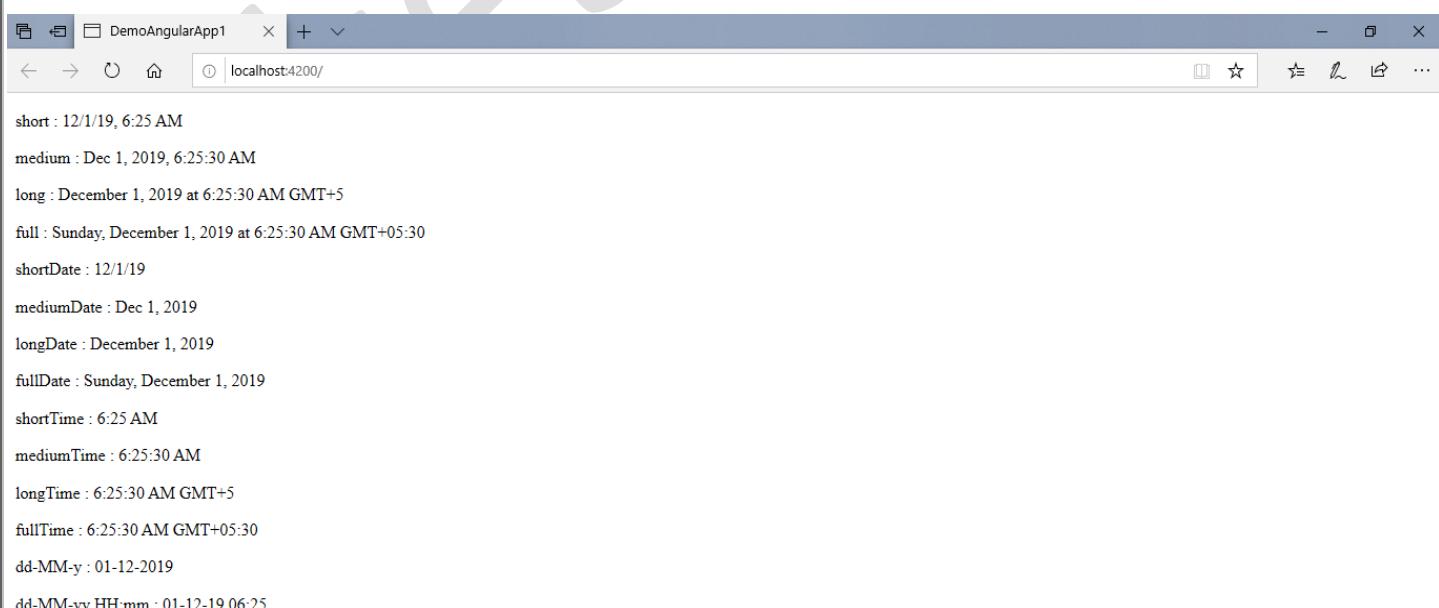
Example:

Component Class (app.component.ts):

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

@Component({
  selector: 'app-root',
  template: `{{title}}
    <p>short : {{currentDate | date:'short'}} </p>
    <p>medium : {{currentDate | date:'medium'}} </p>
    <p>long : {{currentDate | date:'long'}} </p>
    <p>full : {{currentDate | date:'full'}} </p>
    <p>shortDate : {{currentDate | date:'shortDate'}} </p>
    <p>mediumDate : {{currentDate | date:'mediumDate'}} </p>
    <p>longDate : {{currentDate | date:'longDate'}} </p>
    <p>fullDate : {{currentDate | date:'fullDate'}} </p>
    <p>shortTime : {{currentDate | date:'shortTime'}} </p>
    <p>mediumTime : {{currentDate | date:'mediumTime'}} </p>
    <p>longTime : {{currentDate | date:'longTime'}} </p>
    <p>fullTime : {{currentDate | date:'fullTime'}} </p>
    <p>dd-MM-y : {{currentDate | date:'dd-MM-y'}} </p>
    <p>dd-MM-yy HH:mm : {{currentDate | date:'dd-MM-yy HH:mm'}} </p>
  `
})
export class AppComponent
{
  title: string = "Angular's Date Pipe Example" ;
  currentDate: Date = new Date();
}
```

Output:



```
short : 12/1/19, 6:25 AM
medium : Dec 1, 2019, 6:25:30 AM
long : December 1, 2019 at 6:25:30 AM GMT+5
full : Sunday, December 1, 2019 at 6:25:30 AM GMT+05:30
shortDate : 12/1/19
mediumDate : Dec 1, 2019
longDate : December 1, 2019
fullDate : Sunday, December 1, 2019
shortTime : 6:25 AM
mediumTime : 6:25:30 AM
longTime : 6:25:30 AM GMT+5
fullTime : 6:25:30 AM GMT+05:30
dd-MM-y : 01-12-2019
dd-MM-yy HH:mm : 01-12-19 06:25
```

decimal Pipe:

DecimalPipe is an angular **Pipe** API and belongs to **CommonModule**. The Decimal Pipe is used to Format a number as decimal number into text. This pipe will format the number according to locale rules. It is used to set the minimum integer digits and minimum-maximum fractional digits. It also formats numbers into comma separated groups. It uses **number** keyword with pipe operator.

Syntax:

```
number_expression | number[:digitInfo]
```

Where

number_expression: An angular expression that will give output a number.

number : A pipe keyword that is used with pipe operator so it is the name of the pipe.

digitInfo : It defines number format.

Now we will understand how to use **digitInfo**. The syntax for **digitInfo** is as follows.

```
{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}
```

Where

minIntegerDigits : Minimum number of integer digits. Default is 1.

minFractionDigits : Minimum number of fraction digits. Default is 0.

maxFractionDigits : Maximum number of fraction digits. Default is 3.

Some sample examples:

1. Using default format:

```
minIntegerDigits = 1
```

```
minFractionDigits = 0
```

```
maxFractionDigits = 3
```

Now find a number that will be formatted.

```
num1: number = 12.638467846;
```

Now use Decimal Pipe.

```
{{num1 | number}}
```

Find the output.

12.638

We will observe that fraction digit has been truncated to count 3, because maximum fraction digit is only 3.

2. Use format '**3.2-5**':

```
minIntegerDigits = 3
```

```
minFractionDigits = 2
```

```
maxFractionDigits = 5
```

Now find the number that will be formatted.

```
num1: number = 12.638467846;
```

Now use Decimal Pipe.

```
{{num1 | number: '3.2-5'}}
```

Find the output.

012.63847

In our number, integer part is 12 that is of 2 digits, so adding 0 as prefix, it has been of 3 digits that is 012. This is because minimum integer digit required is 3. We will observe that fraction digit has been truncated to count 5, because maximum fraction digit is given to 5.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

3. Format '3.2-5'

minIntegerDigits = 3

minFractionDigits = 2

maxFractionDigits = 5

Now find the number that will be formatted.

num2 = 0.5;

Now use Decimal Pipe.

{ { num2 | number:'3.2-5' } }

Find the output.

000.50

In our number, integer part is 0 that is of 1 digit, so adding two times 0 as prefix, it has been of 3 digits that is 000. This is because minimum integer digit required is 3. We will observe that fraction digit has been increased to count 2, because minimum fraction digit is given to 2, so it's added with 0 as postfix.

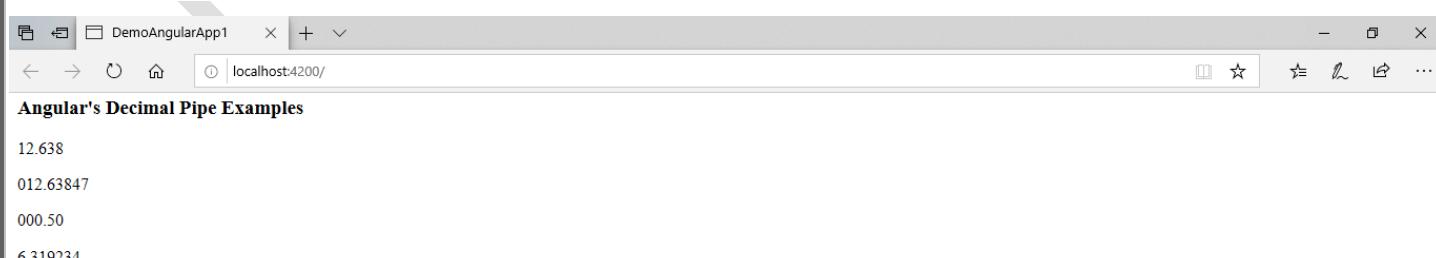
Now find the component used in our example.

Component Class (app.component.ts):

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

@Component({
  selector: 'app-root',
  template: `
    <h3>{{title}}</h3>
    <div>
      <p> {{num1 | number}} </p>
      <p> {{num1 | number:'3.2-5'}} </p>
      <p> {{num2 | number:'3.2-5'}} </p>
      <p> {{num1 * num2 | number:'1.3-6'}} </p>
    </div>
  `
})
export class AppComponent {
  title: string = "Angular's Decimal Pipe Examples";
  num1: number = 12.638467846;
  num2: number = 0.5;
}
```

Output:



Angular's Decimal Pipe Examples

12.638

012.63847

000.50

6.319234

percent Pipe:

Angular **PercentPipe** is an angular **Pipe** API that formats a number as a percentage according to locale rules. It belongs to **CommonModule**.

Syntax:

number_expression | percent[:digitInfo]

Where

number_expression: An angular expression that will give output a number.

percent : A pipe keyword that is used with pipe operator and it converts number into percent so it is name of the pipe.

digitInfo: It defines a percentage format. We have described the use of **digitInfo** in **DecimalPipe** section. It is used with following syntax.

{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}

Now find some sample examples.

1. Using default format:

minIntegerDigits = 1

minFractionDigits = 0

maxFractionDigits = 3

Now find a number that will be changed into percentage.

num1 = 2.5;

Now use Percent Pipe

{ { num1 | percent } }

Output:

250%

2. Use format '2.2-5'

minIntegerDigits = 2

minFractionDigits = 2

maxFractionDigits = 5

Now find the number that will be changed into percentage.

num1 = 2.5;

Now use Percent Pipe

{ { num1 | percent:'2.2-5' } }

Output:

250.00%

We will observe that there is two digits in fraction part. This is because minimum fraction digits required is 2.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Now find the component used in our example.

Component Class (app.component.ts):

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

@Component({
  selector: 'app-root',
  template: `
    <h3>{{title}}</h3>
    <div>
      <p> {{num1 | percent}} </p>
      <p> {{num1 | percent:'2.2-5'}} </p>
      <p> {{num2 | percent:'1.2-5'}} </p>
      <p> {{num1 * num2 | percent:'1.2-3'}} </p>
      <p> {{num3 | percent}} </p>
      <p> {{num3 | percent:'3.1-5'}} </p>
      <p> {{num3 | percent:'1.5-5'}} </p>
    </div>
  `
})

export class AppComponent {
  title: string = "Angular's Percent Pipe Example"
  num1: number = 2.5;
  num2: number = 0.5;
  num3: number = 0.25645;
}
```

Output:



Angular's Percent Pipe Example

250%
250.00%
50.00%
125.00%
26%
025.645%
25.64500%

currency Pipe:

CurrencyPipe is an angular **Pipe** API that formats a number as currency using locale rules. It belongs to **CommonModule**. **CurrencyPipe** uses **currency** keyword with pipe operator to format a number into currency format.

Syntax:

```
number_expression | currency[:currencyCode[:symbolDisplay[:digitInfo]]]
```

Where

number_expression : An angular expression that will give output a number.

currency: A pipe keyword that is used with pipe operator. It formats a number into currency format so it is name of the pipe..

currencyCode: This is the currency code such as **INR** for Indian rupee, **USD** for US dollar. Default is **USD**.

symbolDisplay: The format for the currency indicator.

```
symbolDisplay: 'code' | 'symbol' | 'symbol-narrow' | string | boolean = 'symbol'
```

One of the following:

- **code**: Show the code (such as USD).
- **symbol(default)**: Show the symbol (such as \$).
- **symbol-narrow**: Use the narrow symbol for locales that have two symbols for their currency.

For example, the Canadian dollar CAD has the symbol CA\$ and the symbol-narrow \$. If the locale has no narrow symbol, uses the standard symbol for the locale.

- **String**: Use the given string value instead of a **code** or a **symbol**.
- **Boolean** (marked deprecated in v5): true for **symbol** and false for **code**.

digitInfo: It defines a currency format. We have described the use of **digitInfo** in **DecimalPipe** section. It is used with following syntax.

{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}

Find some sample examples:

1. Using default format:

currencyCode = USD

symbolDisplay = false

minIntegerDigits = 1

minFractionDigits = 0

maxFractionDigits = 3

Now find a number that will be changed into currency.

cur1 = 0.25;

Now use Currency Pipe

{ { cur1 | currency } }

Output:

\$0.25

2. Use format '2.2-4'

currencyCode = USD

`symbolDisplay = true`

minIntegerDigits = 2

`minFractionDigits = 2`

maxFractionDigits = 4

Now find a number that will be changed into currency.

```
cur2 = 10.263782;
```

Now use Currency Pipe.

{{cur2}}

Output:

\$10.**26**38

Now find the component used in our ex

Component Class (app.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h3> {{title}} </h3>
    <div>
      <p> {{cur1 | currency}} </p>
      <p> {{cur1 | currency:'INR':false}} </p>
      <p> {{cur2 | currency:'INR':false:'1.2-4'}} </p>
      <p> {{cur2 | currency:'USD':true:'2.2-4'}} </p>
      <p> {{cur2 | currency:'EUR':true:'2.2-5'}} </p>
      <p> {{cur3 | currency:'JPY':true:'4.2-3'}} </p>
      <p> {{cur3 | currency:'INR':true:'4.2-3'}} </p>
      <p> {{cur3 | currency:'INR':true:'4.0-0'}} </p>
      <p> {{cur3 | currency:'CAD'}} </p>
      <p> {{cur3 | currency:'CAD':'code'}} </p>
      <p> {{cur3 | currency:'CAD':'symbol':'4.2-3'}} </p>
      <p> {{cur3 | currency:'CAD':'symbol-narrow':'4.2-4'}} </p>
      <p> {{cur3 | currency:'INR':'Indian Rupee':'4.1-1'}} </p>
      <p> {{1200 | currency:'INR':'symbol':'4.0'}} </p>
      <p> {{cur2 | currency:'CLP'}} </p>
    </div>
  `
```

```
})
export class AppComponent {
  title: string = "Angular's Currency Pipe Example";
  cur1: number = 0.25;
  cur2: number = 10.263782;
  cur3: number = 5500.56789;
}
```

Copyright © 2019 <https://www.facebook.com/groups/BakeshSoftNetAngular/>. All Rights Reserved

Output:

```
Angular's Currency Pipe Example

$0.25
INR0.25
INR10.2638
$10.2638
€10.26378
₹5,500.568
₹5,500.568
₹5,501
CA$5,500.57
CAD5,500.57
CA$5,500.568
$5,500.5679
Indian Rupee5,500.6
₹1,200
CLP10
```

Warning: The currency pipe has been changed in Angular v5. The `symbolDisplay` option (third parameter) is now a string instead of a boolean. The accepted values are "code", "symbol" or "symbol-narrow".

Angular Currency Pipe No Decimal:

Angular Currency Pipe by default displays two decimal points irrespective of currency type. If currency value is 100.

```
<p> {{value | currency:'INR':'symbol'}} </p>
```

Output:

₹100.00

To remove decimal points from the Angular currency pipe, we need to pass `digitInfo` parameter `fractions` as zero.

```
<p> {{value | currency:'INR':'symbol':'3.0'}} </p>
```

Output:

₹100

Note: Few country currencies does not support cent values in that case decimal points are truncated. For example Chilean peso Currency CLP does not support cents. So if the currency value is 100.23. It is displayed as 100.

```
<p> {{value | currency:'CLP':'symbol'}} </p>
```

Output:

CLP100

Angular Currency Pipe example with locale:

We can pass local as parameter to Angular currency Pipe as shown below.

```
<p> {{value | currency:'CAD':'symbol':'4.2-2':'fr'}} </p>
```

But the above code returns the error in console saying Missing locale data for the locale "fr".

We are passing French locale information as "fr". But in our application we don't have locale information for French.

We need to register the locale information.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

```
import { registerLocaleData } from '@angular/common';
import localeFr from '@angular/common/locales/fr';
registerLocaleData(localeFr, 'fr');
```

Follow the below steps to use Angular Currency Pipe with locale.

1. Import the registerLocaleData from @angular/common
2. Import locale Information from @angular/common/locales/fr.
3. And Finally register the information using registerLocaleData method.

```
<p> {{100.23 | currency:'CAD':'symbol':'4.2-2':'fr'}} </p>
```

Output:

0 100,23 \$CA

```
<p> {{100.23 | currency:'CAD':'symbol':'3.2-2':'fr'}} </p>
```

Output:

100,23 \$CA

```
<p> {{100.23 | currency:'EUR':'symbol':'3.2-2':'fr'}} </p>
```

Output:

100,23 €

Display Currency symbol to the right using Angular Currency Pipe:

A lot of European countries use the currency symbol at the right side of currency value (France, Germany, Spain, Italy countries).

If you pass the locale information the symbol will be automatically displayed at the right side of value as shown in above French locale.

But the remaining currencies without locale still displays currency symbol to the left only.

json Pipe:

JsonPipe is an angular **Pipe** API and belongs to **CommonModule**. JsonPipe is used to converts a value into its JSON-format representation. It is useful for debugging.

JsonPipe uses **JSON.stringify** to convert into JSON string. In **JSON.stringify**, **JSON** is a subset of JavaScript. **JsonPipe** uses **json** keyword to convert value into JSON string using pipe operator as follows.

Syntax:

value_expression | json

Where

value_expression: It is any object component. It means, a value of any type to convert into a JSON-format string.

json: It represent JsonPipe so it is name of the pipe.

For the example find the object of **Address** class in our component.

```
address1 = new Address('S.R. Nagar', 'Hyderabad', 'India');
```

Use **json** keyword with pipe operator (**|**) to convert the given object into JSON string.

```
<pre> {{address1 | json}} </pre>
```

<pre> tag helps to get JSON string as pretty print.

Output:

```
{
  "Area": "S.R. Nagar",
  "City": "Hyderabad",
  "Country": "India"
}
```

JsonPipe with Array:

Here we will create an array of objects of `Address` class and then we will display it into JSON format using `JsonPipe`.
Find the array created in our component.

```
address1 = new Address('Andheri', 'Mumbai', 'India');
address2 = new Address('Ameerpet', 'Hyderabad', 'India');
addresses: Address[] = [this.address1, this.address2];
```

Use `json` keyword with pipe operator (`|`).

```
<pre> {{addresses | json}} </pre>
```

Using HTML elements such as `<div>`, we will use `JsonPipe` as given below:

```
<pre> <div [innerHTML]="addresses | json"></div> </pre>
```

<pre> tag helps to get JSON string as pretty print.

Output:

```
[
  {
    "Area": "Andheri",
    "City": "Mumbai",
    "Country": "India"
  },
  {
    "Area": "Ameerpet",
    "City": "Hyderabad",
    "Country": "India"
  }
]
```

JsonPipe with Object of Objects:

In our example we have a `Person` class. This class will be initialized by passing objects of `Name` and `Address` class in our component.

```
addressNew1 = new Address('Andheri', 'Mumbai', 'India');
name = new Name('Manish', 'Kumar');
dob = new Date(1994, 11, 15);
person = new Person(101, this.name, this.dob, this.addressNew1);
```

Now if we want to convert `person` object's any property into JSON string using `JsonPipe`, we can do as follows.

```
<pre> {{person.pname | json}} </pre>
```

<pre> tag helps to get JSON string as pretty print.



Output:

```
{
  "fname": "Manish",
  "lname": "Kumar"
}
```

Now use **JsonPipe** with **person** object.

```
<pre> {{person | json}} </pre>
```

Output:

```
{
  "id": 101,
  "pname": {
    "fname": "Manish",
    "lname": "Kumar"
  },
  "dob": "1994-12-14T18:30:00.000Z",
  "address": {
    "Area": "Andheri",
    "City": "Mumbai",
    "Country": "India"
  }
}
```

If we use **<pre>** tag, we will get **pretty print** output.

Complete Example: Component Class (app.component.ts)

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

@Component({
  selector: 'app-root',
  template: `
    <h3> {{title}} </h3>
    <pre> {{data | json}} </pre>
    <pre> {{addresses | json}} </pre>
    <pre> {{person.pname | json}} </pre>
    <pre> {{person | json}} </pre>
  `
})
export class AppComponent {
  title: string = "Angular's Json Pipe Example";

  data = {
    'id': 101,
    'name': {
      'firstname': 'Angular',
      'lastname': 'Pipes'
    }
  };
}
```

```

address1 = new Address('Andheri', 'Mumbai', 'India');
address2 = new Address('Ameerpet', 'Hyderabad', 'India');
addresses: Address[] = [this.address1, this.address2];

addressNew1 = new Address('S.R. Nagar', 'Hyderabad', 'India');
name = new Name('Manish', 'Kumar');
dob = new Date(1994, 11, 15);
person = new Person(101, this.name, this.dob, this.addressNew1);
}

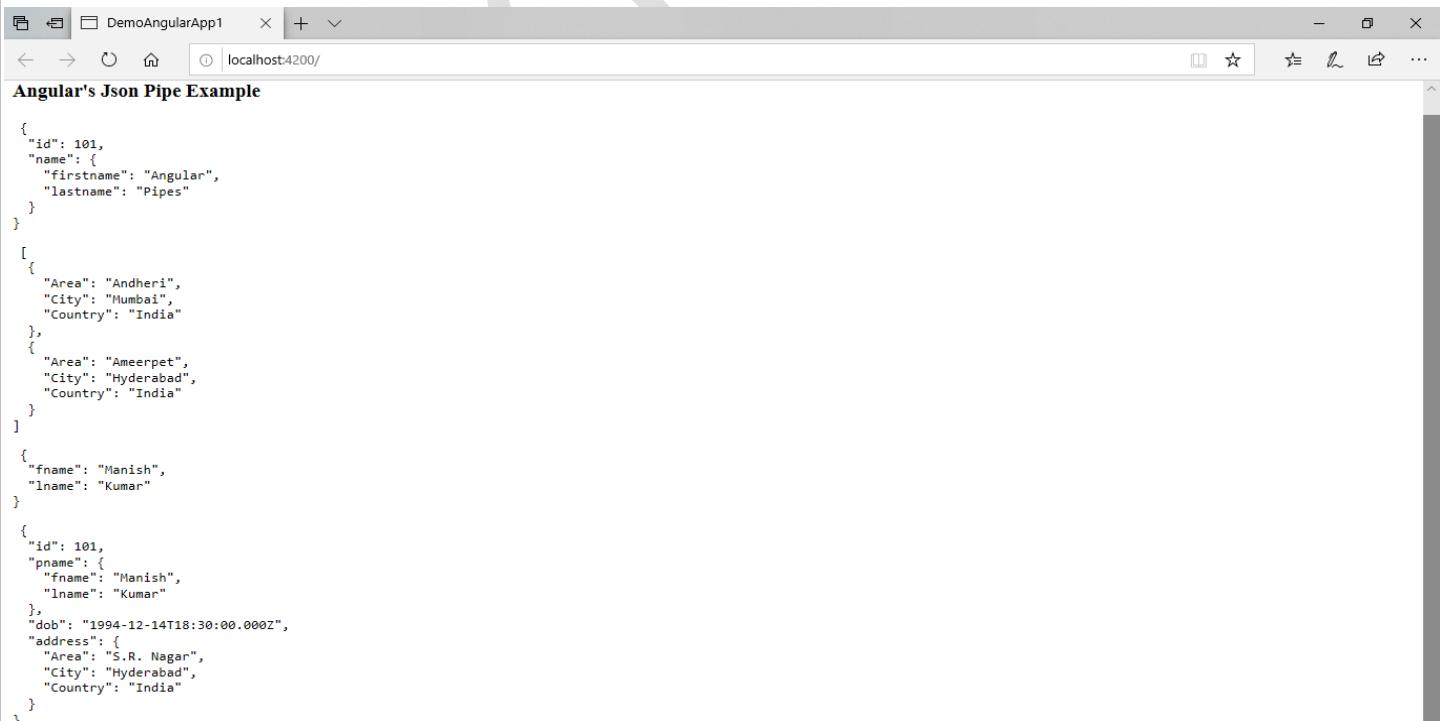
class Address {
    constructor(public Area:string, public City:string, public Country:string) {
    }
}

class Name {
    constructor(public fname:string, public lname:string) {
    }
}

class Person {
    constructor(public id:number, public pname:Name,
                public dob:Date, public address:Address) {
    }
}

```

Output:



The screenshot shows a browser window titled "DemoAngularApp1" with the URL "localhost:4200/". The page displays a JSON object representing a person's information, including their name, date of birth, and address.

```

{
  "id": 101,
  "name": {
    "firstname": "Angular",
    "lastname": "Pipes"
  },
  [
    {
      "Area": "Andheri",
      "City": "Mumbai",
      "Country": "India"
    },
    {
      "Area": "Ameerpet",
      "City": "Hyderabad",
      "Country": "India"
    }
  ],
  {
    "fname": "Manish",
    "lname": "Kumar"
  },
  {
    "id": 101,
    "pname": {
      "firstname": "Manish",
      "lastname": "Kumar"
    },
    "dob": "1994-12-14T18:30:00.000Z",
    "address": {
      "Area": "S.R. Nagar",
      "City": "Hyderabad",
      "Country": "India"
    }
  }
}

```

KeyValue Pipe:

It Transforms Object or Map into an array of key value pairs. It was introduced as a new pipe in Angular 6.1.

Syntax:

```
input_expression | keyvalue [ : compareFn ]
```

Where

Input: any

compareFn: (a: KeyValue, b: KeyValue) => number - Optional. Default is defaultComparator.

KeyValue pipe help you to iterate through objects, maps, and arrays. Today, the **ngFor** directive doesn't support iterations over objects or Maps. To fix this issue, Angular 6.1 introduces a new **KeyValue** pipe. The **KeyValue** pipe converts an **Object** or **Map** into an array of key-value pairs to use with **ngFor** directive.

As mentioned earlier, the **KeyValue** pipe converts the object or map into an array of key-value pairs. It also sorts the array based on the keys in the following order:

- ❖ First in alphabetical order if keys are strings
- ❖ Then by their numeric value if numbers
- ❖ Then by their boolean value if boolean.

If the keys are of complex types you can also define a custom compare function to manipulate the order based on your requirements.

The following angular code defines an **object**, a **map**, and an **array** in the app component. It also has a **descOrder** function for custom ordering. This function will be used for custom ordering of the **testArray** array.

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

@Component({
  selector: 'app-root',
  template: `
    <h3> {{title}} </h3>
    <p>Object & KeyValue Pipe (By Default Ordered By Key in Asc Order)</p>
    <div *ngFor="let item of myObject | keyvalue">
      {{ item | json }}
      Key: <b>{{item.key}}</b> and Value: <b>{{item.value}}</b>
    </div>

    <p>Object & KeyValue Pipe (By Default Ordered By Key in Asc Order)</p>
    <div *ngFor="let item of testObject | keyvalue">
      {{ item | json }}
      Key: <b>{{item.key}}</b> and Value: <b>{{item.value}}</b>
    </div>

    <p>Maps & KeyValue Pipe (By Default Ordered By Key in Asc Order)</p>
    <div *ngFor="let item of testMap | keyvalue">
      {{ item | json }}
      Key: <b>{{item.key}}</b> and Value: <b>{{item.value}}</b>
    </div>
```

```

<p>Arrays & KeyValue Pipe (Ordered By Key in Desc Order via Custom Compare Function)</p>
<div *ngFor="let item of testArray | keyvalue:descOrder">
  {{ item | json }}
  Key: <b>{{item.key}}</b> and Value: <b>{{item.value}}</b>
</div>
<br>

})
```

```

export class AppComponent {
  title: string = "Angular's KeyValue Pipe Example";

  myObject = {
    "Peter": 21,
    "Fleming": 22,
    "Smith": 23,
  }

  testObject: { [key: number]: string } =
    {
      1: 'Object Value 1',
      2: 'Object Value 2',
      3: 'Object Value 3'
    };

  testMap = new Map([
    [2, 'Map Value 2'],
    [3, 'Map Value 3'],
    [null, 'Map Value 4'],
    [1, 'Map Value 1']
  ]);

  testArray = [
    "Array Item 1",
    "Array Item 2",
    "Array Item 3"
  ];

  descOrder = (a,b) => {
    return a.key > b.key ? -1 : (b.key > a.key ? 1 : 0);
  }
}

```

The pipe 'keyvalue' could not be found error:
As the keyvalue pipe introduced in Angular 6.1 release. If you try to use `keyvalue` pipe in older versions of Angular you will get [The pipe 'keyvalue' could not be found](#) error.
You need to update `@angular/core` by running below angular cli ng command
`ng update @angular/core`

As you can see above, the syntax makes it quite easy to iterate objects, maps, arrays (with custom ordering) etc. using `*ngFor` and `keyvalue` pipe.

Note:

- As **Arrays** only have values. There are no keys present. The pipe will add the keys starting automatically from **0**.
- Objects/maps/arrays** will arrange the **order** by default using keys in ascending order.
- If the **keys** are string, keys will be arranged in alphabetical order.
- If the keys are mixed i.e., one key is string and other key is number then both are converted to strings, first sorted by numbers in ascending order and then sorted by strings in alphabetical order.
- The **Map** has an element with **null** key and entries are in a random order. The **null** or **undefined** keys will be displayed at the end.

You should see the following output when you run the app.

Output:



Angular's KeyValue Pipe Example

Object & KeyValue Pipe (By Default Ordered By Key in Asc Order)

```
{ "key": "Fleming", "value": 22 } Key: Fleming and Value: 22
{ "key": "Peter", "value": 21 } Key: Peter and Value: 21
{ "key": "Smith", "value": 23 } Key: Smith and Value: 23
```

Object & KeyValue Pipe (By Default Ordered By Key in Asc Order)

```
{ "key": "1", "value": "Object Value 1" } Key: 1 and Value: Object Value 1
{ "key": "2", "value": "Object Value 2" } Key: 2 and Value: Object Value 2
{ "key": "3", "value": "Object Value 3" } Key: 3 and Value: Object Value 3
```

Maps & KeyValue Pipe (By Default Ordered By Key in Asc Order)

```
{ "key": 1, "value": "Map Value 1" } Key: 1 and Value: Map Value 1
{ "key": 2, "value": "Map Value 2" } Key: 2 and Value: Map Value 2
{ "key": 3, "value": "Map Value 3" } Key: 3 and Value: Map Value 3
{ "key": null, "value": "Map Value 4" } Key: and Value: Map Value 4
```

Arrays & KeyValue Pipe (Ordered By Key in Desc Order via Custom Compare Function)

```
{ "key": "2", "value": "Array Item 3" } Key: 2 and Value: Array Item 3
{ "key": "1", "value": "Array Item 2" } Key: 1 and Value: Array Item 2
{ "key": "0", "value": "Array Item 1" } Key: 0 and Value: Array Item 1
```

As you can see, the key's order for array items is in a descending manner due to the custom ordering applied to the array's `keyvalue` pipe.

Preserve original order of objects, maps, and arrays while iterating using `*ngFor` and `keyvalue` pipe:

| In Component Class: | In Template: |
|--------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>myObject = { "Peter": 21, "Fleming": 22, "Smith": 23, }</pre> | <pre><div *ngFor="let item of myObject keyvalue: 0"> {{ item json }} Key: {{item.key}} and Value: {{item.value}} </div></pre> |

Output:

```
{ "key": "Peter", "value": 21 } Key: Peter and Value: 21
{ "key": "Fleming", "value": 22 } Key: Fleming and Value: 22
{ "key": "Smith", "value": 23 } Key: Smith and Value: 23
```

I18nSelect Pipe:

I18nSelectPipe is the generic selector that displays the string that matches the current value.

For example, if we want to replace the text like 'happy' with the text-emoji like ':)', or 'sad' with ':-(', then we can do this using the I18nSelectPipe.

Syntax to use I18nSelectPipe is:

```
{{ value_expression | i18nSelect : mapping }}
```

Where

value_expression: string - a string to be internationalized.

i18nSelect – It represent I18nSelectPipe.

mapping: object - an object that indicates the text that should be displayed for different values of the provided value.

Note: If none of the keys of the mapping match the value, then the content of the other key is returned when present, otherwise an empty string is returned.

Example1:

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

@Component({
  selector: 'app-root',
  template: `
    <h3> {{title}} </h3>
    <p>I am {{'happy' | i18nSelect : emojiMap}}</p>
    <p>I am {{'sad' | i18nSelect : emojiMap}}</p>
    <p>I am {{'none' | i18nSelect : emojiMap}}</p>
  `
})
export class AppComponent {
  title: string = "Angular's I18nSelect Pipe Example";

  emojiMap = {'happy':':-)', 'sad':':-(', 'other':':-|'};
}
```

Output:



Angular's I18nSelect Pipe Example

I am :)

I am :(

I am :|

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Example2:

```

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

@Component({
  selector: 'app-root',
  template: `
    <h3> {{title}} </h3>
    <b>Select Gender: </b>
    <select [(ngModel)]=gender>
      <option value="male">Male</option>
      <option value="female">Female</option>
      <option value="other">Other</option>
    </select>
    <br /><br />
    <div>{{gender | i18nSelect: inviteMap}} </div>
  `
})

export class AppComponent {
  title: string = "Angular's I18nSelect Pipe Example";
  gender: string = 'male';
  inviteMap: any = {'male': 'Invite him.', 'female': 'Invite her.', 'other': 'Invite them.'};
}

```

Output:



Angular's I18nSelect Pipe Example

Select Gender: Male

Invite him.

Here, by default male is selected in a dropdown list so I18nSelectPipe displays the string that matches the current value i.e. 'Invite him' for male and if you change selection like female or other in a dropdown list so based on that, I18nSelectPipe displays other string that matches the current value such as 'Invite her' or 'Invite them'.

I18nPlural Pipe:

It maps a value to a string that pluralizes the value according to locale rules. It belongs to **CommonModule**.

The i18nPlural pipe has a simple usage, where we just evaluate a numeric value against an object mapping different string values to be returned depending on the result of the evaluation. This way, we can render different strings on our template depending if the numeric value is zero, one, two, more than N, and so on.

Syntax:

```
{{ value_expression | i18nPlural : pluralMap [ : locale ] }}
```

Where

Value_expression: number - the number to be formatted

pluralMap: object - an object that mimics the ICU format.

For more details see <http://userguide.icu-project.org/formatparse/messages>.

locale: string - a string defining the locale to use (uses the current LOCALE_ID by default).

Optional. Default is undefined.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Example1:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h3> {{title}} </h3>
    <div>{{ messages.length | i18nPlural: messageMapping }}</div>
  `
})
export class AppComponent {
  title: string = "Angular's I18nPlural Pipe Example";
  messages: any[] = ['Message 1'];
  messageMapping: {[k: string]: string} = {'=0': 'No messages.', '=1': 'One message.', 'other': '# messages.'};
}
```

Output:



Angular's I18nPlural Pipe Example

One message.

Example2:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h3> {{title}} </h3>
    <b>Enter Value: </b>
    <input type="text" [(ngModel)]="value" />
    <br /><br />
    <div>{{ value | i18nPlural: messageMapping }}</div>
  `
})
export class AppComponent {
  title: string = "Angular's I18nPlural Pipe Example";
  value: number = 1;
  messageMapping: {[k: string]: string} = {'=0': 'No messages.', '=1': 'One message.', 'other': '# messages.'};
}
```

Output:



Angular's I18nPlural Pipe Example

Enter Value:

One message.

Async Pipe:

Angular provides a special pipe known as **Async**, which allows us to bind our templates directly to values that arrive asynchronously. This ability is great for working with **promises** and **observables**.

Generally, If we are getting the result using observable or promise, We need to use following steps,

1. Subscribe the observable or promise.
2. Wait for a callback.
3. Once a result is received, store the result of the callback in a variable.
4. The last step is to bind that variable on the template.

The **Async** pipe in angular will subscribe internally to an **Observable** or **Promise** and return the latest value it has emitted. Whenever a new value is emitted from an **Observable** or **Promise**, the **Async** pipe marks the component to be checked for changes. When the component gets destroyed, the **Async** pipe unsubscribes automatically to avoid potential memory leaks. We can use the **Async** pipe in Angular application by including the **CommonModule** which exports all the basic Angular directives and pipes.

"Unwraps a value from an asynchronous primitive."

Syntax of AsyncPipe

The below syntax shows how we can use the Async pipe with the object expression.

`{{obj_expression | async}}`

Where

obj_expression: Observable | Promise

async: It is the name of the pipe.

Example1: Using AsyncPipe with Promise

Async pipe for promises automatically adds a then callback and renders the response. Now let's see an example where we are binding a Promise to the view. Clicking the Resolve button resolves the promise.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <div>
      <p>{{title}}</p>
      <button (click)="clicked()">{{ arrived ? 'Reset' : 'Resolve' }}</button>
      &nbsp;<span>Wait for it... {{ greeting | async }}</span>
    </div>
  `
})
export class AppComponent {
  title: string = "Angular's Async Pipe with Promise Example";
  greeting: Promise<string>|null = null;
  arrived: boolean = false;
  private resolve: Function|null = null;
  constructor() { this.reset(); }
  reset() {
    this.arrived = false;
    this.greeting = new Promise<string>((resolve, reject) => { this.resolve = resolve; });
  }
  clicked() {
    if (this.arrived) {
      this.reset();
    } else {
      this.resolve !('hi there!');
      this.arrived = true;
    }
  }
}
```

It is the non-null assertion operator.
Note that the type for this.resolve is Function|null so it can possibly be null.

Angular's Async Pipe with Promise Example

Wait for it...

After Clicking Resolve Button

Angular's Async Pipe with Promise Example

Wait for it... hi there!

Example2: Using AsyncPipe with Observable

AsyncPipe for Observables automatically subscribes to the observable, renders the output, and then also unsubscribes when the component is destroyed. So, we don't need to handle the clean-up logic ourselves. There is no need to unsubscribe manually in the component. Angular handles subscriptions of async pipes for us automatically using `ngOnDestroy`. AsyncPipe uses the `OnPush` change detection out of the box. We need to make sure that all our business logic is immutable and always returns new objects. We can say that the `OnPush` change detection strategy is great for performance so we should be using an async pipe as much as possible. Now let's see the example below that binds the time Observable to the view. The Observable continuously updates the view with the current time.

```
import { Component } from '@angular/core';
import { Observable, Observer } from 'rxjs';

@Component({
  selector: 'app-root',
  template: `
    <div>
      <p> {{title}} </p>
      <p> <b>Time:</b> {{ time | async }} </p>
    </div>
  `
})

export class AppComponent {
  title: string = "Angular's Async Pipe with Observable Example"

  time = new Observable<string>((observer: Observer<string>) => {
    setInterval(() => observer.next(new Date().toString()), 1000);
  });
}
```

Output:



Angular's Async Pipe with Observable Example

Time: Wed Dec 04 2019 18:02:12 GMT+0530 (India Standard Time)

In the above example, when we pipe our observable directly to the async pipe, the async pipe performs a subscription for us and then returns whatever gets passed to it.

Advantages of using the async pipe are:

1. We don't need to call `subscribe` on our observable and store the intermediate data on our component.
2. We don't need to remember to unsubscribe from the observable when the component is destroyed.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Custom Pipe:

The Angular comes with many built-in pipes like Date pipe, Currency pipe, Decimal pipe etc... But these pipes might not be sufficient for our needs. That is where we need to build a Custom pipe in Angular to fulfil our needs.

We can write our own custom pipe and that will be used in the same way as angular built-in pipes. To create custom pipe, angular provides **Pipe** and **PipeTransform** interfaces. Every pipe is decorated with **@Pipe** where we define the name of our custom pipe.

Every pipe will implement **PipeTransform** interface. This interface provides **transform()** method and we have to override it in our custom pipe class. **transform()** method will decide the input types, number of arguments and its types and output type of our custom pipe. We perform the following steps to create a custom pipe.

Step 1: Create a typescript class.

Step 2: Decorate the class using **@Pipe**.

Step 3: Implement **PipeTransform** interface.

Step 4: Override **transform()** method.

Step 5: Configure the class in application module with **@NgModule**.

Step 6: Ready to use our custom pipe anywhere in application.

On the basis of change detection, angular provides two types of pipes.

Pure pipe: This will run only for pure changes in component properties.

Impure pipe: This will run for any type of changes in component properties.

Pure and impure pipe are declared using **@Pipe** decorator. Here we will create custom pipes with and without arguments. We will also create custom pipe using angular built-in pipes. We will also provide example of pure and impure pipe.

@Pipe Decorator and PipeTransform Interface:

To create custom pipe we need to understand angular **Pipe** and **PipeTransform** API.

@Pipe Decorator

@Pipe decorator is a **Pipe** interface. A typescript class is decorated by **@Pipe** to create an angular custom pipe.

Find the **Pipe** interface from the angular Pipe API document.

```
interface Pipe {
  name : string
  pure : boolean
}
```

name: Assign custom pipe name.

pure: Assign true or false. Default is true. If true then pipe will be a pure pipe otherwise impure pipe. So by default all pipes are pure pipe.

For example we will use **Pipe** interface as **@Pipe** decorator as follows.

```
@Pipe({
  name: 'welcome'
})
```

Here custom pipe name is **welcome** and as we know that by default **pure** metadata value is **true**, so **welcome** pipe will be a pure pipe.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

PipeTransform Interface:

PipeTransform is an angular interface. To create a custom pipe our typescript class has to implement **PipeTransform** interface. It has been defined as follows in angular **PipeTransform** API document.

```
interface PipeTransform {
  transform(value: any, ...args: any[]): any
}
```

In the above interface a method `transform()` has been defined. It accepts minimum one argument and maximum any number of arguments. The parameter type is `any`. It means we can pass any type of object. `transform()` method returns value of type `any`. To create custom pipe our typescript class will implement **PipeTransform** and override `transform()` method. The use of parameters is as follows.

1. First parameter (value: any): This is the value in left side of our pipe operator (`|`).
2. Optional parameters: These are arguments used with pipe in right side of pipe operator (`|`).
3. Value returned by method: This is the value that is the output of our custom pipe.

Create Simple Custom Pipe:

Now we will start creating our custom pipe. Here we will create a simple custom pipe named as welcome. The syntax is as follows to use in Component View (Template):

`string_expression | welcome`

We will pass a person name for `string_expression` as string and output will be a welcome message. Now follow the steps to create our **welcome** pipe.

Step 1: Create a typescript class named as `WelcomePipe`.

Step 2: Import `Pipe` and `PipeTransform` interface from angular core module.

Step 3: Decorate `WelcomePipe` with `@Pipe`. Its `name` metadata will define custom pipe name.

Step 4: `WelcomePipe` will implement `PipeTransform` interface.

Step 5: Override `transform()` method of `PipeTransform` interface. The parameter of `transform()` can be of any type. In our example we are using string data type and return type is also a string. Here we will perform task which needs to be done by our custom pipe and return the result. This is the result which will be returned by custom pipe.

Step 6: To make custom pipe available at application level, declare `WelcomePipe` in `@NgModule` decorator. In our example module file is `app.module.ts`.

Find the `WelcomePipe` class.

Pipe Class (`welcome.pipe.ts`):

```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
  name: 'welcome'
})
export class WelcomePipe implements PipeTransform {
  transform(value: string): string {
    let message = "Welcome to " + value;
    return message;
  }
}
```

The `name` metadata of `@Pipe` decorator has the value `welcome` and that will be the name of our custom pipe. In our `app.component.ts` file we are using **welcome** pipe as given below.

`{{name | welcome}}`

Output

Welcome to Rakesh

Pass Arguments in Custom Pipe:

Now we will discuss some custom pipes that will accept arguments. To facilitate custom pipe to accept arguments we have to use optional parameters in `transform()` method while creating custom pipe API. If we want one argument with custom pipe then use one optional parameter in `transform()` method. If we want two arguments with custom pipe then use two optional parameters in `transform()` method and so on. Now find our custom pipes that are using arguments.

1. Create `strformat` Custom Pipe

Syntax to use in Component View (Template):

`string_expression | strformat[:separator]`

The `strformat` will format the given string expression. The spaces between words will be replaced by given separator. Now find the typescript class `StrFormatPipe` that will create `strformat` pipe.

Pipe Class (`strformat.pipe.ts`):

```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
  name: 'strformat'
})
export class StrFormatPipe implements PipeTransform {
  transform(value: string, seperator: string): string {
    let result = value.split(' ').join(seperator);
    return result;
  }
}
```

The `name` metadata of `@Pipe` decorator has the value `strformat` and that will be the name of our custom pipe. We will observe that in `transform()` method first parameter is a string and this will be used by `string_expression`. Second parameter is also a string and this will be used by `strformat` pipe to pass arguments. In our example argument is string value given by `separator`. Now `transform()` method has return type as string, so the output of our pipe `strformat` will be a string.

In our `app.component.ts` file we are using `strformat` pipe as given below.

```
{ {message | strformat:'+'} }
```

Output

My+name+is+Rakesh

2. Create `division` Custom Pipe

Using `division` custom pipe we will divide a number by a given number.

Syntax to use in Component View (Template):

`number_expression | division[:num_divisor]`

Description:

`number_expression` : An expression that will return a number and that will be dividend.

`num_divisor`: This number is the argument of our pipe that will be used as divisor.

Now find the typescript class `DivisionPipe` that will create `division` pipe.

Pipe Class (division.pipe.ts):

```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
  name: 'division'
})
export class DivisionPipe implements PipeTransform {
  transform(dividend: number, divisor: number): number {
    let num = dividend / divisor;
    return num;
  }
}
```

Custom pipe name is assigned to **name** metadata of **@Pipe** decorator. Here custom pipe name is **division**. In **transform()** method, there are two parameters of number type. First parameter is for expression used with pipe operator and second parameter is used for pipe argument. Return type of method is also a number, so **division** pipe operator will give output as number.

In our **app.component.ts** file we are using **division** pipe as given below.

```
<div>Dividend: <input [(ngModel)]="dividend"></div>
<div>Divisor: <input [(ngModel)]="divisor"></div>
<p>
  Division Result: {{dividend | division: divisor}}
</p>
```

Here dividend and divisor values are input by user. Pure pipe and impure pipe both detect changes immediately for component string property value changes. So whenever there will be change in value of component property **dividend** and **divisor**, our custom pipe **division** will run every time.

Output:

Division Result: 3.2857142857142856

3. Create **repeat** Custom Pipe

We are creating a **repeat** pipe. This pipe will repeat a given word. This will be repeated as many times as given frequency.

Syntax to use in Component View (Template):

string_expression | repeat[:frequency]

Find the typescript class **RepeatPipe** that will create **repeat** pipe.

Pipe Class (repeat.pipe.ts):

```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
  name: 'repeat'
})
export class RepeatPipe implements PipeTransform {
  transform(word: string, frequency: number): string {
    let count = 1;
    let strResult= word;
    while (count < frequency) {
      strResult = strResult + ' ' + word;
      count = count + 1;
    }
    return strResult;
  }
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



`@Pipe` decorator has defined the pipe name as `repeat`. The method `transform()` has two parameters. One is of string type and second is of number type. First parameter is for `string_expression` and second parameter is for `frequency`. In our `app.component.ts` file we are using `repeat` pipe as given below.

```
{{name | repeat:5}}
```

Output

Rakesh Rakesh Rakesh Rakesh Rakesh

4. Create myjson Custom Pipe

Here we are creating `myjson` custom pipe. This pipe will convert an expression into JSON format. It will accept two arguments.

Syntax to use in Component View (Template):

`expression | myjson[:prettyprint[:fields]]`

Description:

expression: Expression that will be converted into JSON format. It can be primitive data type or any object.

prettyprint: If the value is 0, then no pretty print and if the value is 1 or greater than 1, then we will get pretty print JSON format.

fields: If no fields are provided then all the fields of object will take part in JSON format, if specified then only those fields will take part in JSON format.

If no argument is passed then `myjson` will convert the given object into JSON format with all fields and without pretty print. Now find the typescript class `MyJSONPipe` that will create `myjson` pipe.

Pipe Class (`myjson.pipe.ts`):

```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
  name: 'myjson'
})
export class MyJSONPipe implements PipeTransform {
  transform(value: any, prettyprint: number, fields: string): string {
    let array = (fields == null? null : fields.split(','));
    let pp = (prettyprint == null ? 0 : prettyprint);
    let result = JSON.stringify(value, array, pp);
    return result;
  }
}
```

The name of custom pipe `myjson` is provided in `@Pipe` decorator. The method `transform()` has three parameters. First parameter is of `any` type. It means any type of object can be passed to convert into JSON format. The rest two parameters are pipe arguments. Pretty print of JSON format also requires `<pre>` tag while using `myjson` pipe.

In our `app.component.ts` file we are using `myjson` pipe as given below.

```
<pre>{{person | myjson}}</pre>
<pre>{{person | myjson:0:'id,age'}}</pre>
<pre>{{person | myjson:1:'id,name'}}</pre>
```

```
person: Person = new Person(1,'David',25);
class Person{
  constructor(public id:number,public name:string,public age: number){}
}
```

Output:

{"id":1,"name":"David","age":25}

{"id":1,"age":25}

```
{
  "id": 1,
  "name": "David"
}
```

Use Built-in Pipe in Custom Pipe:

Custom pipe can also use angular built-in pipe such as **DatePipe**, **UpperCasePipe**, **LowerCasePipe**, **CurrencyPipe**, and **PercentPipe**. In our example we will create **myuppercaseone** and **myuppercasetwo** pipe. Both pipes will use **UpperCasePipe** built-in pipe. There are two ways to use **UpperCasePipe** in our custom pipes.

1. Extending **UpperCasePipe** built-in pipe class:

UpperCasePipe is an angular built-in pipe. It implements **PipeTransform** interface and override **transform()** method as we do in our custom pipes. Here we will create our custom pipe class named as **MyUppercaseOnePipe**. This time our class **MyUppercaseOnePipe** will not implement **PipeTransform** because it will extend **UpperCasePipe** built-in pipe and that is already implementing **PipeTransform** interface. Now find the **MyUppercaseOnePipe** class.

Pipe Class (**myuppercaseone.pipe.ts**):

```
import {Pipe} from '@angular/core';
import {UpperCasePipe} from '@angular/common';

@Pipe({
  name: 'myuppercaseone'
})

export class MyUppercaseOnePipe extends UpperCasePipe{
  transform(value: string): string {
    let result = super.transform(value);
    result = result.split(' ').join('-');
    return result;
  }
}
```

UpperCasePipe is the API of **CommonModule**, so we need to import **UpperCasePipe** class from angular **common** package. Here we are overriding **transform()** method from **UpperCasePipe**, so we cannot change number of parameters and types in **transform()** method. Now find the line of code.

```
let result = super.transform(value);
```

Using **super** keyword we call the parent class method. The value passed to **transform()** method of **MyUppercaseOnePipe** class is first processed by **transform()** method of **UpperCasePipe** class and then we perform our changes and then we return the final result.

In our **app.component.ts** file we are using **myuppercaseone** pipe as given below.

```
{{message | myuppercaseone}}
```

Output:

MY-NAME-IS-RAKESH

2. Using object of **UpperCasePipe** built-in pipe class:

We can also use built-in pipe in our custom pipe by creating object of built-in pipe API. Here we will create our custom pipe as usual by implementing **PipeTransform** and within **transform()** method we will use parameters and their types according to our requirements. To process the data by **UpperCasePipe** pipe, we will create the object of it and pass the data to its **transform()** method. Now find the **MyUppercaseTwoPipe** class.

Pipe Class (**myuppercasetwo.pipe.ts**):

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

```
import {Pipe, PipeTransform} from '@angular/core';
import {UpperCasePipe} from '@angular/common';
@Pipe({
  name: 'myuppercaseTwo'
})
export class MyUppercaseTwoPipe implements PipeTransform{
  transform(value: string, separator: string): string {
    let uppercase = new UpperCasePipe();
    let result = uppercase.transform(value);
    result = result.split(' ').join(separator);
    return result;
  }
}
```

In our `app.component.ts` file we are using **myuppercaseTwo** pipe as given below.

```
 {{message | myuppercaseTwo: '+'}}
```

Output

MY+NAME+IS+RAKESH

Chaining of Custom Pipes:

Chaining of custom pipes is using more than one pipe together. The output of first pipe will be the input for next pipe and so on.

Example:

```
 {{name | repeat:5 | myuppercaseOne}}
```

In the above code snippet we are using two custom pipes **repeat** and **myuppercaseOne**. First **repeat** pipe will run and then its output will be the input for **myuppercaseOne** pipe.

Output:

RAKESH-RAKESH-RAKESH-RAKESH-RAKESH

Pure and Impure Custom Pipe and Change Detection

Angular provides pure and impure pipes on the basis of change detection. Here we will discuss pure and impure pipes with examples.

1. Change Detection

Angular uses a change detection process for changes in data-bound values. Change detection runs after every keystroke, mouse move, timer tick, and server response. In case of pipe, angular picks a simpler, faster change detection algorithm.

2. Pure Pipes

By default all pipes are pure pipe. Pure pipes are those pipes that have `pure: true` in `@Pipe` decorator while creating pipe class. If we have not used `pure` metadata then its default value will be `true` in `@Pipe` decorator. Pure pipes executes only for pure changes in its input values.

Find the pure changes.

- Change to a primitive input values such as String, Number, Boolean.
- Change to object reference of Date, Array, Function, Object.

Above changes are pure changes. If the input values used with pure pipe, comes under the pure changes then pipe will run again to give output accordingly. Pure pipe is created as follows:

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

```
@Pipe({
  name: 'pipename'
})
```

By default `@Pipe` decorator is using `pure: true`.

3. Impure Pipes

Impure pipes will run for every component change detection cycle. So it is obvious that impure pipes will also run for pure changes. Impure pipe will run for every keystroke or mouse move. So the conclusion is that impure pipe will run a lot and hence we should take care while using impure pipe because it may reduce performance of the application and can destroy user experience. Impure pipe is created as follows.

```
@Pipe({
  name: 'pipename',
  pure: false
})
```

Make sure that metadata `pure` has been assigned with `false` boolean value.

4. Example of Pure and Impure Pipes:

Now we will discuss an example of pure and impure pipe. We will see that pure pipe will run only for pure changes whereas impure pipe will run for every type of changes in component properties. We will start by creating pure and impure pipes whose `transform()` method will use `Company` class as parameter type and `string` as return type.

Find the pure pipe named as `companyone`

Pipe Class (`companyone.pipe.ts`):

```
import {Pipe, PipeTransform} from '@angular/core';
import {Company} from './company';
@Pipe({
  name: 'companyone'
})
export class CompanyOnePipe implements PipeTransform {
  transform(obj: Company): string {
    let output = obj.cname+': '+obj.location;
    return output;
  }
}
```

Now find the impure pipe named as `companytwo`

Pipe Class (`companytwo.pipe.ts`):

```
import {Pipe, PipeTransform} from '@angular/core';
import {Company} from './company';
@Pipe({
  name: 'companytwo',
  pure: false
})
export class CompanyTwoPipe implements PipeTransform {
  transform(obj: Company): string {
    let output = obj.cname+': '+obj.location;
    return output;
  }
}
```

We will observe that **companyone** and **companytwo** both pipes are doing same task. The only difference is that **companyone** is a pure pipe and **companytwo** is an impure pipe.

Now we will code scenarios for pure and impure changes in **Company** object.

Suppose we have created **Company** object as follows:

```
compName:string = "RakeshSoftNet Technologies";
compLocation:string = "Hyderabad";
company = new Company(this.compName, this.compLocation);
```

a. Pure change in company object by changing its reference: To change the reference we will create a new object of **Company** class and assign it to component property **company** as follows:

```
createCompany() {
    this.company = new Company(this.compName, this.compLocation);
}
```

When we call the above method then pure and impure both pipes will run. It means **companyone** and **companytwo** both pipe will run again.

b. Impure change in company object by updating its property values: To generate the scenario of impure change, we will update the property value of our **company** object as follows:

```
updateCompany() {
    this.company.cname = this.compName;
    this.company.location = this.compLocation;
}
```

When we call the above method then only impure pipe will run. It means **companyone** pipe will not run again but **companytwo** pipe will run again for the changes in **company** object.

Now find the below code from **app.component.ts** file:

```
Company Name: <input [(ngModel)]="compName"/> {{compName}}
<br/><br/>
Location: <input [(ngModel)]="compLocation"/> {{compLocation}}
<br/><br/>
<button (click)="updateCompany()">Update Existing</button>
<button (click)="createCompany()">Create New</button>
<br/><br/>
<b>a. Using Pure Pipe : companyone</b><br/><br/>
{{company | companyone}}
<br/><br/>
<b>b. Using Impure Pipe : companytwo</b><br/><br/>
{{company | companytwo}}
```

Output

Initially both text box will be populated with following values.

Company Name : RakeshSoftNet Technologies

Location: Hyderabad

Company Name: RakeshSoftNet Technologies

Location: Hyderabad

a. Using Pure Pipe : companyone

RakeshSoftNet Technologies : Hyderabad

b. Using Impure Pipe : companytwo

RakeshSoftNet Technologies : Hyderabad

Case 1: Change the text box values as follows:

Company Name : RakeshSoftNet

Location: Pune

Now click on **Update Existing** button. Output of **companyone** pipe will be as follows.

RakeshSoftNet Technologies : Hyderabad

And output of **companytwo** pipe will be as follows.

RakeshSoftNet: Pune

Let us understand what is happening now. When we click on **Update Existing** button then `updateCompany()` will execute and performs impure change in `company` object. So only impure pipe **companytwo** will run again and change its output. There will be no change in pure pipe **companyone** output because it didn't not run again due to impure change in `company` object.

Case 2: Change the text box values as follows:

Company Name : RakeshSoftNet

Location: Pune

Now click on **Create New** button. Output of **companyone** pipe will be as follows.

RakeshSoftNet : Pune

And output of **companytwo** pipe will be as follows.

RakeshSoftNet : Pune

Let us understand what is happening. When we click on **Create New** button then `createCompany()` will execute and performs pure change in `company` object. So this time pure and impure pipe both will run again. It means **companyone** and **companytwo** both pipe will run again and change its output.

Complete Program:

Person Class (person.ts):

```
export class Person{
    constructor(public id:number,public name:string,public age:number){
    }
}
```

Company Class (company.ts):

```
export class Company {
    constructor(public cname:string, public location:string) {
    }
}
```

Welcome Pipe (welcome.pipe.ts):

```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
  name: 'welcome'
})
export class WelcomePipe implements PipeTransform {
  transform(value: string): string {
    let message = "Welcome to " + value;
    return message;
  }
}
```

StrFormat Pipe (strformat.pipe.ts):

```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
  name: 'strformat'
})
export class StrFormatPipe implements PipeTransform {
  transform(value: string, seperator: string): string {
    let result = value.split(' ').join(seperator);
    return result;
  }
}
```

Division Pipe (division.pipe.ts):

```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
  name: 'division'
})
export class DivisionPipe implements PipeTransform {
  transform(dividend: number, divisor: number): number {
    let num = dividend / divisor;
    return num;
  }
}
```

Repeat Pipe (repeat.pipe.ts):

```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
  name: 'repeat'
})
export class RepeatPipe implements PipeTransform {
  transform(word: string, frequency: number): string {
    let cnt = 1;
    let strResult= word;
    while (cnt < frequency) {
      strResult = strResult + ' ' + word;
      cnt = cnt + 1;
    }
    return strResult;
  }
}
```



MyJSON Pipe (myjson.pipe.ts):

```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
  name: 'myjson'
})
export class MyJSONPipe implements PipeTransform {
  transform(value: any, prettyprint: number, fields: string): string {
    let array = (fields == null? null : fields.split(','));
    let pp = (prettyprint == null ? 0 : prettyprint);
    let result: string = JSON.stringify(value, array, pp);
    return result;
  }
}
```

MyUppercaseOne Pipe (myuppercaseone.pipe.ts):

```
import {Pipe} from '@angular/core';
import {UpperCasePipe} from '@angular/common';
@Pipe({
  name: 'myuppercaseone'
})
export class MyUppercaseOnePipe extends UpperCasePipe{
  transform(value: string): string {
    let result = super.transform(value);
    result = result.split(' ').join('-');
    return result;
  }
}
```

MyUppercaseTwo Pipe (myuppercasetwo.pipe.ts):

```
import {Pipe, PipeTransform} from '@angular/core';
import {UpperCasePipe} from '@angular/common';
@Pipe({
  name: 'myuppercasetwo'
})
export class MyUppercaseTwoPipe implements PipeTransform{
  transform(value: string, seperator: string): string {
    let uppercase = new UpperCasePipe();
    let result = uppercase.transform(value);
    result = result.split(' ').join(seperator);
    return result;
  }
}
```

CompanyOne Pipe (companyone.pipe.ts):

```
import {Pipe, PipeTransform} from '@angular/core';
import {Company} from './company';
@Pipe({
  name: 'companyone'
})
export class CompanyOnePipe implements PipeTransform {
  transform(obj: Company): string {
    let output = obj.cname+ ' : ' + obj.location;
    return output;
  }
}
```

CompanyTwo Pipe (companytwo.pipe.ts):

```
import {Pipe, PipeTransform} from '@angular/core';
import {Company} from './company';
@Pipe({
  name: 'companytwo',
  pure: false
})
export class CompanyTwoPipe implements PipeTransform {
  transform(obj: Company): string {
    let output = obj.cname+ ' : ' + obj.location;
    return output;
  }
}
```

Root Module (app.module.ts): Import & Declare Pipes

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule, NO_ERRORS_SCHEMA, CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import {WelcomePipe} from './welcome.pipe';
import {StrFormatPipe} from './strformat.pipe';
import {DivisionPipe} from './division.pipe';
import {RepeatPipe} from './repeat.pipe';
import {MyJSONPipe } from './myjson.pipe';
import {MyUppercaseOnePipe} from './myuppercaseone.pipe';
import {MyUppercaseTwoPipe} from './myuppercasetwo.pipe';
import {CompanyOnePipe} from './companyone.pipe';
import {CompanyTwoPipe} from './companytwo.pipe';

@NgModule({
  declarations: [
    AppComponent,
    WelcomePipe, StrFormatPipe,
    DivisionPipe, RepeatPipe,
    MyJSONPipe, MyUppercaseOnePipe,
    MyUppercaseTwoPipe, CompanyOnePipe, CompanyTwoPipe
  ],
  imports: [
    BrowserModule, FormsModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent],
  schemas: [CUSTOM_ELEMENTS_SCHEMA, NO_ERRORS_SCHEMA]
})
export class AppModule { }
```

Root Component (app.component.ts):

```

import { Component } from '@angular/core';
import { Person } from './person';
import { Company } from './company';
@Component({
  selector: 'app-root',
  template: `
    <h3> {{title}} </h3>
    <p>{{name | welcome}}</p>
    <p>{{name | strformat:'+'}}</p>
    <div>Dividend: <input [(ngModel)]="dividend"></div>
    <br />
    <div>Divisor: <input [(ngModel)]="divisor"></div>
    <p>
      Division Result: {{dividend | division: divisor}}
    </p>
    <p>{{name | repeat:5}}</p>
    <pre>{{person | myjson}}</pre>
    <pre>{{person | myjson:0:'id,age'}}</pre>
    <pre>{{person | myjson:1:'id,name'}}</pre>
    <p>{{message | myuppercaseone}}</p>
    <p>{{message | myuppercasetwo:'+'}}</p>
    <p>{{name | repeat:5 | myuppercaseone}}</p>
    Company Name: <input [(ngModel)]="compName"/> {{compName}}
    <br/><br/>
    Location: <input [(ngModel)]="compLocation"/> {{compLocation}}
    <br/><br/>
    <button (click)="updateCompany()">Update Existing</button>
    <button (click)="createCompany()">Create New</button>
    <br/><br/>
    <b>a. Using Pure Pipe : companyone</b><br/><br/>
    {{company | companyone}}
    <br/><br/><b>b. Using Impure Pipe : companytwo</b><br/><br/>
    {{company | companytwo}}
  `

})
export class AppComponent {
  title: string = "Angular's Custom Pipe Example";
  name: string = "Rakesh";
  dividend: number = 23;
  divisor: number = 7;
  person:Person = new Person(1,'David',25);
  message: string = "My Name is Rakesh";
  compName:string = "RakeshSoftNet Technologies";
  compLocation:string = "Hyderabad";
  company = new Company(this.compName, this.compLocation);

  createCompany() {
    this.company = new Company(this.compName, this.compLocation);
  }
  updateCompany() {
    this.company.cname = this.compName;
    this.company.location = this.compLocation;
  }
}

```

Output:

Welcome to Rakesh

Rakesh

Dividend:

Divisor:

Division Result: 3.2857142857142856

Rakesh Rakesh Rakesh Rakesh Rakesh

```
{"id":1,"name":"David","age":25}
{"id":1,"age":25}
{
  "id": 1,
  "name": "David"
}
```

MY-NAME-IS-RAKESH

MY+NAME+IS+RAKESH

RAKESH-RAKESH-RAKESH-RAKESH-RAKESH

Company Name: RakeshSoftNet Technologies

Location: Hyderabad

a. Using Pure Pipe : companyone

RakeshSoftNet Technologies : Hyderabad

b. Using Impure Pipe : companytwo

RakeshSoftNet Technologies : Hyderabad

Few More Real Time Examples:

Example1: Creating a custom pipe that convert **Fahrenheit** to **Celsius** and **Celsius** to **Fahrenheit**.

Pipe Class (tempconvertor.pipe.ts)

```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
  name: 'tempConverter'
})
export class TempConverterPipe implements PipeTransform {
  transform(value: number, unit: string): string {
    if(value && !isNaN(value)) {
      if (unit === 'C') {
        let temperature = (value - 32) / 1.8;
        return temperature.toFixed(2) + " C";
      } else if (unit === 'F'){
        let temperature = (value * 1.8 ) + 32
        return temperature.toFixed(2) + " F";
      }
    }
    return;
  }
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

The above class (**TempConverterPipe**) is what implements the logic of the pipe. This class implements **PipeTransform interface**. Every pipe we built must implement the **PipeTransform interface**.

The PipeTransform interface has only one method known as the **transform**. This interface takes the value being piped as the first argument. It takes the variable number of optional arguments of any type. It returns the final transformed data.

We have implemented the transform method, which takes two arguments. The first is **value** and the second is the **unit**. The unit expects either C (Convert to Celsius) or F (convert to Fahrenheit). It converts the value received to either to Celsius or to Fahrenheit based on the Unit.

Finally, we decorate the class with a @pipe decorator. @pipe decorator is what tells Angular that the class is a Pipe. @Pipe decorator also provides the **metadata to pipe class**. In the above example, we have provided name metadata, which is the name of the pipe, which we will use in our template.

Importing the Custom Pipe:

Before using our Angular custom pipe, we need to tell our component, where to find it. This done by first by importing it and then including it in declarations array of the AppModule.

```
import { TempConverterPipe } from './tempconvertor.pipe';
@NgModule({
  declarations: [AppComponent, TempConverterPipe],
  imports: [BrowserModule, FormsModule, AppRoutingModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Using the Custom Pipe:

The custom pipes are used in the same as the Angular built-in pipes are used. Add the following code to your app.component.ts file:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <div>
      <h3>Fahrenheit to Celsius </h3>
      <p>Fahrenheit: <input type="text" [(ngModel)]="Fahrenheit"/>
      Celsius: {{Fahrenheit | tempConverter:'C'}}</p>
    </div>

    <div>
      <h3>Celsius to Fahrenheit </h3>
      <p>Celsius: <input type="text" [(ngModel)]="Celsius"/>
      Fahrenheit: {{Celsius | tempConverter:'F'}}</p>
    </div>
  `
})
export class AppComponent {
  Celsius: number;
  Fahrenheit: number;
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Output:



Fahrenheit to Celsius:

Fahrenheit: Celsius: 37.00 C

Celsius to Fahrenheit:

Celsius: Fahrenheit: 78.80 F

Example2: Creating a custom pipe that generates the result of given base and exponent or power value.

Exponents, or **powers**, are a way of indicating that a quantity is to be multiplied by itself some number of times. In the expression 2^5 , 2 is called the **base** and 5 is called the **exponent**, or **power**. 2^5 is shorthand for "multiply five twos together": $2^5 = 2 \times 2 \times 2 \times 2 \times 2 = 32$. 2^5 is read "two raised to the fifth power" or simply "two to the fifth."

In general,

$$x^n = \underbrace{x \cdot x \cdot x \cdot \dots \cdot x}_{n \text{ factors}}$$

where there are n x 's to be multiplied.

Pipe Class (power.pipe.ts):

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe ({
  name : 'power'
})

export class PowerPipe implements PipeTransform {
  transform(base: number, exponent: number): number {
    return Math.pow(base, exponent);
  }
}
```

Importing the Power Pipe and including in declaration array of the AppModule:

```
import { PowerPipe } from './power.pipe';

@NgModule({
  declarations: [AppComponent, PowerPipe],
  .....
})
```

```
export class AppModule {}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Using the Power Pipe:

Add the following code to your app.component.ts file:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h3>{{title}}</h3>
    <div>
      <b>Enter the Base Value:</b><br />
      <input #txtBase type='text' [(ngModel)]="baseValue" />
      <br /><br />
      <b>Enter the Power Value:</b><br />
      <input #txtPower type='text' [(ngModel)]="powerValue" />
      <br />
      <p *ngIf="txtBase.value!='' && txtPower.value!='' ? true : false">
        Result: {{baseValue | power:powerValue}}
      </p>
    </div>
  `
})
export class AppComponent {
  title: string = "Angular's Power Pipe Example";
  baseValue: number;
  powerValue: number;
}
```

Run the application and test it.

Output:



Angular's Power Pipe Example

Enter the Base Value:
2

Enter the Power Value:
5

Result: 32

Example3: Proper Case

Since in the earlier examples, we have seen Angular framework provides us to **uppercase**, **lowercase** and **titlecase** pipes against any string type value as in-build pipes. But, it does not provide any proper case type pipes. So, let's define a pipe that will transfer any string value as proper case and return the result.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Pipe Class (propercase.pipe.ts):

```
import { Pipe, PipeTransform } from '@angular/core'
@Pipe({
  name: 'propercase'
})
export class ProperCasePipe implements PipeTransform {
  transform(value: string, reverse: boolean): string {
    if (typeof (value) == 'string') {
      let intermediate = reverse == false ? value.toUpperCase() : value.toLowerCase();

      return (reverse == false ? intermediate[0].toLowerCase() :
        intermediate[0].toUpperCase()) + intermediate.substr(1);
    }
    else {
      return value;
    }
  }
}
```

Importing the ProperCase Pipe and including in declaration array of the AppModule:

```
import { ProperCasePipe } from './propercase.pipe';
@NgModule({
  declarations: [AppComponent, ProperCasePipe],
  .....
})
export class AppModule { }
```

Using the ProperCase Pipe:

Add the following code to your app.component.ts file:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h3>{{title}}</h3>
    <div>
      <b>Enter Text: </b>
      <input type='text' #txtMessage placeholder="Enter Text Here" [(ngModel)]="message" />
    </div>
    <br />
    <div *ngIf="txtMessage.value!='' ? true : false">
      <span><b>Result in Proper Case:</b></span>
      <div>
        <span>{{message | propercase}}</span>
      </div>
    </div>
  `
})
export class AppComponent {
  title: string = "Angular's Custom Pipe Example - Proper Case";
  message: string = "ProperCase Example";
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Run the application and test it.

Output:



Example4: Create a custom search filter pipe

A very common use case of this is to have an input box where a user enters a search text and the results are filtered appropriately.

Create the Filter Pipe:

Pipe Class (filter.pipe.ts):

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'filter'
})
export class FilterPipe implements PipeTransform {
  transform(items: any[], searchText: string): any[] {
    if(!items) return [];
    if(!searchText) return items;
    searchText = searchText.toLowerCase();
    //Must match searchText with starting in item of list items
    return items.filter( a => a.toLowerCase().startsWith(searchText));
    //OR
    //Must match searchText containing anywhere in item of list items
    return items.filter( a => a.toLowerCase().includes(searchText));
  }
}
```

This code will return a subset of an array of items if any item starts with the searchText string.

Importing the Filter Pipe and including in declaration array of the AppModule:

```
import { FilterPipe } from './filter.pipe';
```

```
@NgModule({
  declarations: [AppComponent, FilterPipe],
  .....
})
```

```
export class AppModule { }
```

Using the Filter Pipe:

Now you can use the filter pipe in your App Component. Add the following code to your app.component.ts file:

Component Class (app.component.ts):

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

@Component({
  selector: 'app-root',
  template: `
    <h3>{{title}}</h3>
    <div>
      <b>Enter Search Text Here: </b>
      <input [(ngModel)]="searchText" placeholder="Enter Search Text Here">
      <br />
      <ul>
        <li *ngFor="let item of items | filter : searchText">
          {{item}}
        </li>
      </ul>
      <div *ngIf="(items | filter: searchText).length === 0">
        <span style="color: red; font-weight: bold">No Matches Found !!!</span>
      </div>
    </div>
  `
})

export class AppComponent {
  title: string = "Angular's Custom Pipe Example - Search Text Filter Pipe";

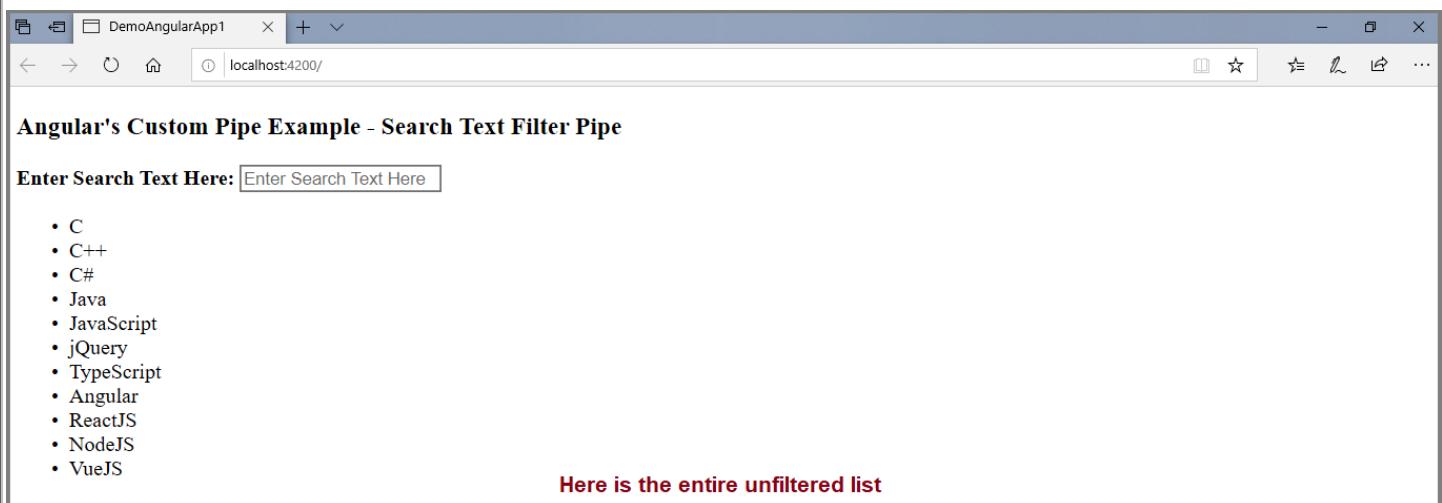
  items: string[] = [
    "C",
    "C++",
    "C#",
    "Java",
    "JavaScript",
    "jQuery",
    "TypeScript",
    "Angular",
    "ReactJS",
    "NodeJS",
    "VueJS"
  ];

  searchText: string;
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Run the application and test it.

Output:

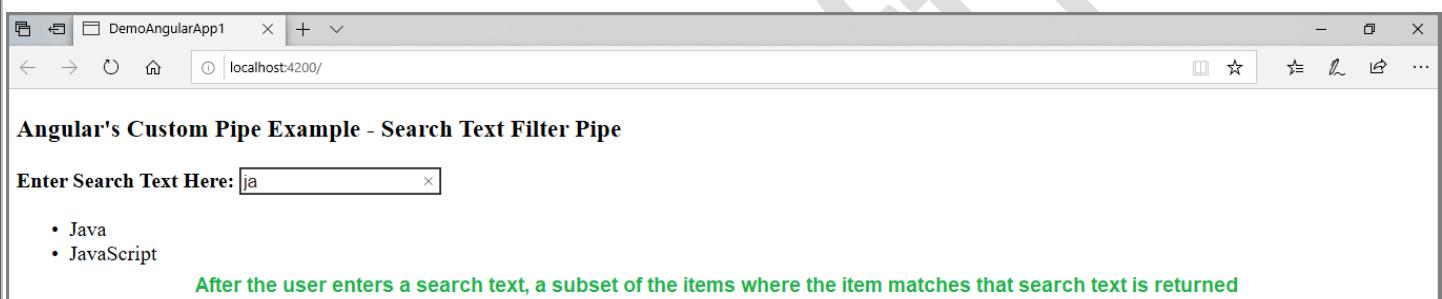


Angular's Custom Pipe Example - Search Text Filter Pipe

Enter Search Text Here:

- C
- C++
- C#
- Java
- JavaScript
- jQuery
- TypeScript
- Angular
- ReactJS
- NodeJS
- VueJS

Here is the entire unfiltered list

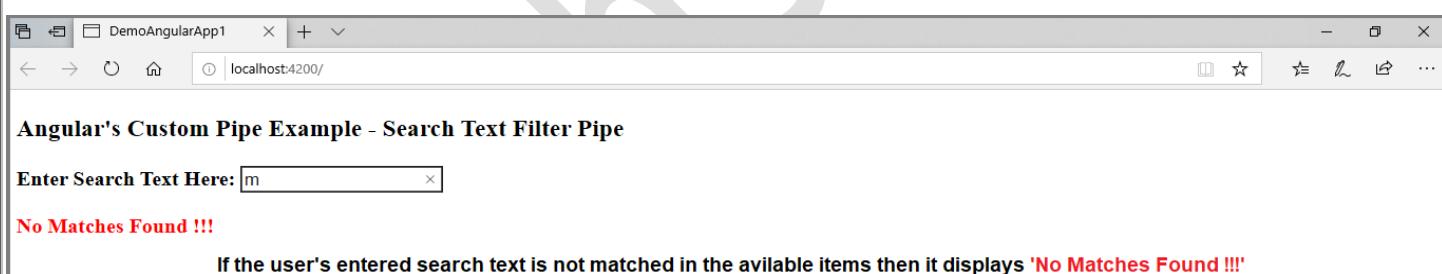


Angular's Custom Pipe Example - Search Text Filter Pipe

Enter Search Text Here: ja

- Java
- JavaScript

After the user enters a search text, a subset of the items where the item matches that search text is returned



Angular's Custom Pipe Example - Search Text Filter Pipe

Enter Search Text Here: m

No Matches Found !!!

If the user's entered search text is not matched in the available items then it displays 'No Matches Found !!!'

The above template code can be also done to display 'No Matched Found !!!' as following:

template:

```

<h3>{{title}}</h3>
<div>
  <b>Enter Search Text Here: </b>
  <input [(ngModel)]="searchText" placeholder="Enter Search Text Here">
  <br />
  <ul>
    <li *ngFor="let item of items | filter : searchText">
      {{item}}
    </li>
    <li *ngIf="(items | filter: searchText).length === 0" style="list-style-type:none">
      <span style="color: red; font-weight: bold">No Matches Found !!!</span>
    </li>
  </ul>
</div>
  
```

The above creating pipe & template code both can be also modified to display '**No Matched Found !!!**' if no item matched with given searchText as following:

Pipe Class (filter.pipe.ts):

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'filter'
})
export class FilterPipe implements PipeTransform {
  transform(items: any[], searchText: string): any[] {
    if(!items) return [];
    if(!searchText) return items;
    searchText = searchText.toLowerCase();
    //Must match searchText with starting in item of list items
    let result: any[] = items.filter( a => a.toLowerCase().startsWith(searchText));
    //OR
    //Must match searchText containing anywhere in item of list items
    let result: any[] = items.filter( a => a.toLowerCase().includes(searchText));
    if (result.length == 0){
      result = [-1];
    }
    return result;
  }
}
```

Component Class's Template (app.component.ts):

```
template: `
<h3>{{title}}</h3>
<div>
  <b>Enter Search Text Here: </b>
  <input [(ngModel)]="searchText" placeholder="Enter Search Text Here">
  <br />
  <ul>
    <ng-container *ngFor="let item of items | filter : searchText">
      <li *ngIf="item === -1" style="list-style-type:none">
        <span style="color: red; font-weight: bold">No Matches Found !!!</span>
      </li>
      <li *ngIf="item !== -1">{{item}}</li>
    </ng-container>
  </ul>
</div>
`
```

NgNonBindable in Angular:

We use **ngNonBindable** when we want to tell Angular not to compile, or bind, a particular section of our page.

The most common example of this is if we wanted to write out some Angular code on the page, for example if we wanted to render out the text `{{ name }}` on our page, like so:

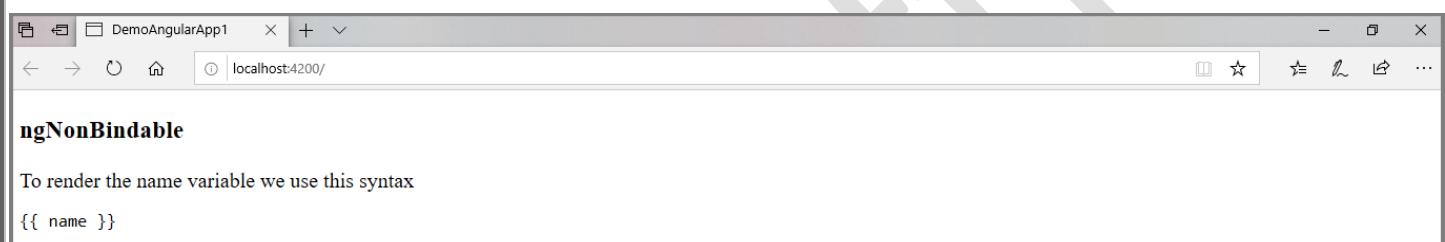
```
<div>
  To render the name variable we use this syntax <pre>{{ name }}</pre>
</div>
```

Normally Angular will try to find a variable called **name** on the component and print out the value of the **name** variable instead of just printing out `{{ name }}`.

To make angular ignore an element we simply add the **ngNonBindable** directive to the element, like so:

```
<div>
  To render the name variable we use this syntax <pre ngNonBindable>{{ name }}</pre>
</div>
```

If we run this in the browser we would see:



Summary:

We use **ngNonBindable** when we want to tell Angular not to perform any binding for an element.

Real Time Example:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h3>{{title}}</h3>
    <div>
      <span ngNonBindable><b>{{10 + 5}} = </b></span>
      <span>{{10 + 5}}</span>
    </div>
  `
})
export class AppComponent {
  title: string = "ngNonBindable";
}
```

Output:



Microsoft .Net Solution

Develop Your **Microsoft** Skills and Knowledge through great Training



Hyderabad's Best Institute for .NET

Angular



Rakesh Singh

RAKESHSOFTNET TECHNOLOGIES Hyderabad

🌐 <http://www.rakeshsoftnet.com>

FACEBOOK <https://www.facebook.com/RakeshSoftNet>

FACEBOOK <https://www.facebook.com/RakeshSoftNetTech>

TWITTER <https://twitter.com/RakeshSoftNet>

PHONE +91 40 4200 8807 MOBILE +91 89191 36822

Class Field Decorators in Angular:

Class field decorators add additional features to class members for Angular directives and components.

Angular offers following class field decorators:

- **@Input()**
- **@Output()**
- **@HostBinding()**
- **@HostListener()**
- **@ContentChild()**
- **@ContentChildren()**
- **@ViewChild()**
- **@ViewChildren()**

@Input():

Use the **@Input()** decorator in a child component or directive to let Angular know that a property in that component can receive its value from its parent component. So an **@Input()** allows data to be input into the child component from the parent component. This is one way communication from parent to child.

Example

In the child:

To use the **@Input()** decorator in a child component class, first import **Input** and then decorate the property with **@Input()**:

```
import { Component, Input } from '@angular/core'; // First, import Input
export class ItemDetailComponent {
  @Input() item: string; // decorate the property with @Input()
}
```

In the parent:

Use property binding to bind the property **item** in the child to the property **currentItem** of the parent:

```
<app-itemdetail [item]="currentItem"></app-itemdetail>
```

Aliasing with the **@Input()** decorator:

You can specify the alias for the property name by passing the alias name to the **@Input()** decorator. The internal name remains as usual.

```
// @Input(alias)
@Input('myItem') item: string;
```

@Output():

Use the **@Output()** decorator in the child component or directive to allow data to flow from the child out to the parent. This is one way communication from child to parent component. An **@Output()** property should normally be initialized to an Angular **EventEmitter** with values flowing out of the component as events.

Example:

In the child:

First, import **Output** and **EventEmitter** in the child component class. Then decorate a **EventEmitter** property with **@Output()** in the component class.

```
import { Component, Output, EventEmitter } from '@angular/core';
export class ItemOutputComponent {

  @Output() newItemEvent = new EventEmitter<string>();

  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
}
```

In the parent:

In the parent's template, bind the parent's method to the child's event:

```
<app-itemoutput (newItemEvent)="addItem($event)"></app-itemoutput>
```

Aliasing with the **@Output()** decorator

You can specify the alias for the property name by passing the alias name to the **@Output()**decorator. The internal name remains as usual.

```
// @Output(alias) propertyName = ...
@Output('myItemEvent') newItemEvent = new EventEmitter<string>();
```

@HostBinding():

In Angular, the Decorator (**@HostBinding**) feature allows you to set the host element's properties from the directive class. It is to interact with the host element on which the directive is applied. **@HostBinding** decorator provides the capabilities to modify properties on the host element.

The **@HostBinding** decorator allows us to programmatically set a property value on the directive's host element. It works similarly to a property binding defined in a template, except it specifically targets the host element. The binding is checked for every change detection cycle, so it can change dynamically if desired.

Let's say you want to change the style properties such as **height**, **width**, **color**, **margin**, **border**, etc. Or **other internal properties** of the host element in the directive class. Here you will need the **@HostBinding()** decorator function to access these properties on the host element and assign a value to it in the directive class.

The **@HostBinding()** decorator takes one parameter, the name of the host element property which value we want to assign in the directive.

Example:

In our example, our host element is an HTML div element. If you want to set border properties of the host element, you can do that using **@HostBinding()** decorator as shown below:

Create a Custom Directive

Let us first create a custom directive named **customborder** using Angular CLI using the below command.

```
ng generate directive customborder
```

Or

```
ng g d customborder
```

Directive Class (customborder.directive.ts):

```
import { Directive, HostBinding } from '@angular/core';

@Directive({
  selector: '[appCustomborder]'
})
export class CustomborderDirective {
  @HostBinding('style.border') border: string;
  constructor() {
    this.border = '10px double red';
  }
}
```

Using this code, the host element's border is set to a 10 pixel double red. Therefore, using the @ HostBinding decorator, you can set the host element's properties in the directive class.

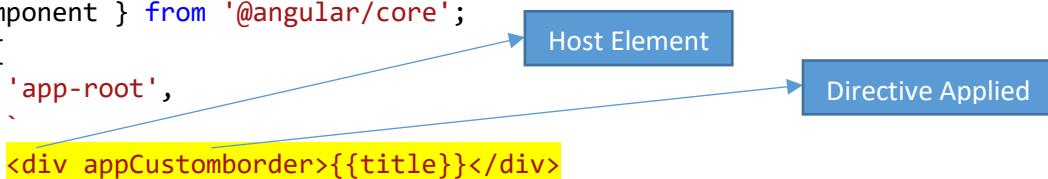
Import and Declare a Custom Directive in Root Module (app.module.ts):

```
import { CustomborderDirective } from './customborder.directive';
@NgModule({
  declarations: [
    AppComponent, CustomborderDirective],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Now use custom directive with default border to host element in our component as following:

Component Class (app.component.ts):

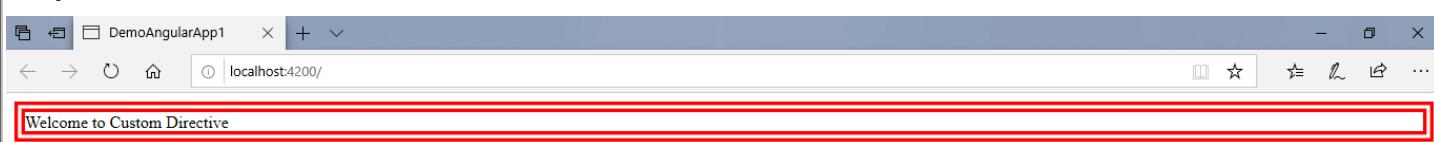
```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <div appCustomborder>{{title}}</div>
  `
})
export class AppComponent {
  title: string = "Welcome to Custom Directive";
}
```



The diagram illustrates the relationship between the Host Element and the Directive Applied. It shows two blue boxes: 'Host Element' and 'Directive Applied'. Two arrows point from the 'appCustomborder' selector in the component template to both boxes, indicating that the directive is applied to the host element.

That's it, we have completed the implementation of **HostBinding**. So let see the output in the browser and see the directive in action to apply the default border with '10px double red' to the host element of a directive.

Output:



@HostListener():

In Angular, the Decorator **@HostListener()** feature allows you to handle events from the host element of the directive class. It is used in angular to interact with the host element on which the directive is applied. The @HostListener decorative listens to the events which are raised on the actual host element.

Let's take the following requirements:

To understand @HostListener() in a better way, consider this simple scenario: on the click of the host element, you want to show an alert window. To do this in the directive class, add @HostListener() and pass the event 'click' to it. Also, associate a function to raise an alert as shown in the listing below:

Directive Class (elementclick.directive.ts):

```
import { Directive, HostListener } from '@angular/core';
@Directive({
  selector: '[appElementclick]'
})
export class ElementclickDirective {
  constructor() { }
  @HostListener('click') onClick() {
    window.alert('Host Element Clicked !!!');
  }
}
```

Import and Declare a Custom Directive in Root Module (app.module.ts):

```
import { ElementclickDirective } from './elementclick.directive';
@NgModule({
  declarations: [
    AppComponent, ElementclickDirective],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Component Class (app.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h1 appElementclick>{{title}}</h1>
  `
})
export class AppComponent {
  title: string = "Welcome to Custom Directive";
}
```

On the click of the host element, you will see the alert window.

Output:



Welcome to Custom Directive



Let's take the another following requirement: when you hover mouse over the host element, the color of the host element should change. In addition, when the mouse is out, the color of the host element should change to its default color. To do this, you need to handle events raised on the host element in the directive class. In Angular, you do this using `@HostListener()`.

In Angular, the `@HostListener()` function decorator makes it super easy to handle events raised in the host element inside the directive class. Let's go back to our requirement that says you must change the color to red when the mouse is hovering, and when it's gone, the color of the host element should change to black. To do this, you need to handle the `mouseover` and `mouseout` events of the host element in the directive class.

Directive Class (`changecolor.directive.ts`):

Now in order to access the host element on which `changecolor` directive will be applied we will need to inject `ElementRef` and `Renderer` into our directive.

```
import { Directive, ElementRef, Renderer, HostListener } from '@angular/core';
@Directive({
  selector: '[appChangecolor]'
})
export class ChangecolorDirective {
  constructor(private el: ElementRef, private renderer: Renderer) {
    this.ChangeColor('black');
  }
  @HostListener('mouseover') onMouseOver() {
    this.ChangeColor('red');
  }

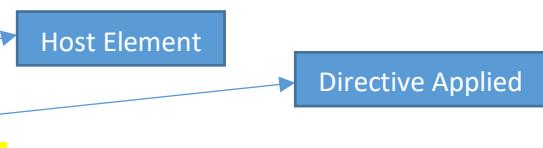
  @HostListener('mouseout') onMouseOut() {
    this.ChangeColor('black');
  }
  ChangeColor(color: string) {
    this.renderer.setStyle(this.el.nativeElement, 'color', color);
  }
}
```

Import and Declare a Custom Directive in Root Module (`app.module.ts`):

```
import { ChangecolorDirective } from './changecolor.directive';
@NgModule({
  declarations: [
    AppComponent, ChangecolorDirective],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Component Class (`app.component.ts`):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h1 appChangecolor>{{title}}</h1>
  `
})
export class AppComponent {
  title: string = "Welcome to Custom Directive";
}
```



The diagram illustrates the relationship between the Host Element and the Directive Applied. It shows two blue boxes: 'Host Element' and 'Directive Applied'. Arrows point from the 'app-root' selector in the component template to both boxes, indicating that the directive is applied to the host element.

Output:



Welcome to Custom Directive

When mouse over to the host element it changes the color to red and when the mouse is out set color back to black:



Welcome to Custom Directive

Working with the HostListener & HostBinding:

Note: We can also bind the properties like classes, attributes etc. using the **HostBinding**.

Use the code below. The **HostBinding** string variable **bgColor** can now update the background color and the **color** can update the text color and the **border** can update the border of the host element.

Directive Class (customstyle.directive.ts):

```
import { Directive, HostBinding, HostListener } from '@angular/core';
@Directive({
  selector: '[appCustomstyle]'
})
export class CustomstyleDirective {
  @HostBinding('style.background-color') bgColor : string;
  @HostBinding('style.color') color : string;
  @HostBinding('style.border') border : string;
  constructor() {
    this.ChangeStyle('aqua','black','1px solid red');
  }
  @HostListener('mouseover') onMouseOver() {
    this.ChangeStyle('yellow','red','5px solid blue');
  }
  @HostListener('mouseout') onMouseOut() {
    this.ChangeStyle('aqua','black','1px solid red');
  }
  ChangeStyle(bgColor: string, color: string, border: string) {
    this.bgColor = bgColor;
    this.color = color;
    this.border = border;
  }
}
```

The two methods **onMouseOver()** and **onMouseOut()** listens to the **mouseover** and **mouseout** events on the host on which the **appCustomstyle** directive will be applied

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



Import and Declare a Custom Directive in Root Module (app.module.ts):

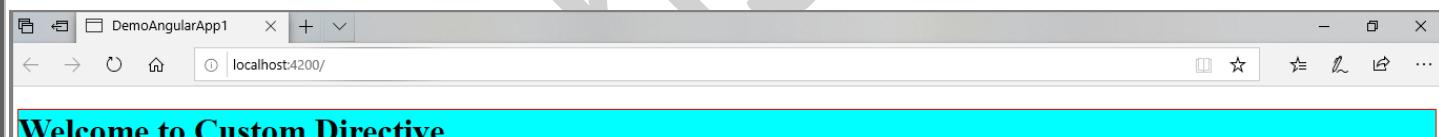
```
import { CustomstyleDirective } from './customstyle.directive';
@NgModule({
  declarations: [
    AppComponent, CustomstyleDirective],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Component Class (app.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h1 appCustomstyle>{{title}}</h1>
  `
})
export class AppComponent {
  title: string = "Welcome to Custom Directive";
}
```

That's it, we have completed the implementation of **HostListener** and **HostBinding**. Now let's run and see the directive in action.

Output:



When the mouse over to the host element it changes the styles like background color, text color and border with new values and when the mouse is out set all styles back to default because whenever mouseover and mouseout happened on the host element “onMouseOver” and “onMouseOut” get triggered since we are listening to the mouse over and out event through **HostListener**.



Let's take the another following requirement:

For example, let's say that we want to create a directive for buttons that dynamically adds a class when we press on it. That could look something like:

Directive Class (buttonpress.directive.ts):

```
import { Directive, HostBinding, HostListener } from '@angular/core';
@Directive({
  selector: '[appButtonPress]'
})
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

```
export class ButtonPressDirective {
    @HostBinding('class.pressed') isPressed: boolean;
    @HostListener('mousedown') hasPressed() {
        this.isPressed = true;
    }
    @HostListener('mouseup') hasReleased() {
        this.isPressed = false;
    }
}
```

Import and Declare a Custom Directive in Root Module (app.module.ts):

```
import { ButtonPressDirective } from './buttonpress.directive';
@NgModule({
    declarations: [
        AppComponent, ButtonPressDirective],
    imports: [],
    providers: [],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Component Style (app.component.css): To define the css class which needs to applied when the button is pressed.

```
.pressed {
    background-color:blue;
    color: white;
    font-weight: bold;
    font-size: 20px;
    border: 2px solid red;
}
```

Component Class (app.component.ts):

```
import { Component } from '@angular/core';
@Component({
    selector: 'app-root',
    template: `
        <h1>{{title}}</h1>
        <button appButtonPress>Press Me !!!</button>
    `,
    styleUrls: ['./app.component.css']
})
export class AppComponent {
    title: string = "Welcome to Custom Directive";
}
```

Output:

Welcome to Custom Directive

Press Me !!!

When we press the button, class property will be applied with the given class i.e. 'pressed'

DOM Structure (Elements Tab):

```
<!DOCTYPE html>
<html lang="en">
<head>...</head>
<body>
    <app-root ng-version="8.2.3" _nghost-xvr-c0="">
        <h1 _ngcontent-xvr-c0="">Welcome to Custom Directive</h1>
        <button class="pressed" appbuttonpress="" _ngcontent-xvr-c0="">Press Me !!!</button>
    </app-root>

```

Let's take the another following requirement:

For example, listen's for a keydown event on the host element and sets its text and border color to a random color from a set of a few available colours:

Directive Class (rainbow.directive.ts):

```
import { Directive, HostBinding, HostListener } from '@angular/core';
@Directive({
  selector: '[appRainbow]'
})
export class RainbowDirective {
  possibleColors: string[] = [
    'darksalmon', 'hotpink', 'lightskyblue', 'goldenrod', 'peachpuff',
    'mediumspringgreen', 'cornflowerblue', 'blanchedalmond', 'lightslategrey'
  ];
  @HostBinding('style.color') color: string;
  @HostBinding('style.border-color') borderColor: string;

  @HostListener('keydown') newColor() {
    const colorPick: number = Math.floor(Math.random() * this.possibleColors.length);
    this.color = this.borderColor = this.possibleColors[colorPick];
  }
}
```

Import and Declare a Custom Directive in Root Module (app.module.ts):

```
import { RainbowDirective } from './rainbow.directive';
@NgModule({
  declarations: [
    AppComponent, RainbowDirective],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})
```

```
export class AppModule { }
```

Component Class (app.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h1>{{title}}</h1>
    <input type="text" appRainbow />
  `
})
export class AppComponent {
  title: string = "Welcome to Custom Directive";
}
```

Output:



The screenshot shows a browser window titled "DemoAngularApp1" with the URL "localhost:4200". The page content is "Welcome to Custom Directive". Below the title, there is a text input field with the placeholder "A rainbow of colors!!". A tooltip appears over the input field with the text: "The @HostListner with the 'keydown' argument listens for the keydown event on the host. We define a function attached to this decorator that changes the value of color and borderColor, and our changes get reflected on the host automatically."

@ContentChild and @ContentChildren:

@ContentChild and **@ContentChildren** are used to fetch single child element or all child elements from content DOM. **@ContentChild** gives first element matching the selector from the content DOM. **@ContentChildren** gives all elements of content DOM as **QueryList**.

Contents queried by **@ContentChild** and **@ContentChildren** are set before **ngAfterContentInit()** is called. If we do any change in content DOM for the matching selector, that will be observed by **@ContentChild** and **@ContentChildren** and we will get updated value.

As a selector for **@ContentChild** and **@ContentChildren**, we can pass directive, component or local template variable. By default **@ContentChildren** only selects direct children of content DOM and not all descendants. **@ContentChildren** has a metadata **descendants** and setting its value true, we can fetch all descendant elements. Here we will discuss **@ContentChild** and **@ContentChildren** example using **directive**, **component** and **ElementRef**.

@ContentChild:

@ContentChild gives the first element or directive matching the selector from the content DOM. If new child element replaces the old one matching the selector in content DOM, then property will also be updated. **@ContentChild** has following metadata properties.

selector: Directive type or the name used for querying. Find the example when type is directive.

```
@ContentChild(BookDirective) book: BookDirective;
```

read: This is optional metadata. It reads a different token from the queried element.

static: True to resolve query results before change detection runs, false to resolve after change detection.

@ContentChildren:

@ContentChildren is used to get **QueryList** of elements or directives from the content DOM. When there is change in content DOM, data in **QueryList** will also change. If child elements are added, we will get those new elements in **QueryList**. If child elements are removed, then those elements will be removed from the **QueryList**. The metadata properties of **@ContentChildren** are as follows.

selector: Directive type or the name used for querying. Find the example when type is directive.

```
@ContentChildren(BookDirective) topBooks: QueryList<BookDirective>;
```

descendants: This is Boolean value. When it is true then direct children and other descendants will also be included. If the value is false then only direct children will be included. **descendants** is used as follows.

```
@ContentChildren(BookDirective, {descendants: true}) allBooks: QueryList<BookDirective>;
```

The default value of **descendants** is **false**.

read: This is optional metadata. It reads a different token from the queried element.

Using AfterContentInit:

AfterContentInit is a lifecycle hook that is called after directive content is fully initialized. It has a method **ngAfterContentInit()**. This method runs after angular loads external content into the component view. This method runs once after first **ngDoCheck()** method.

Contents queried by **@ContentChild** and **@ContentChildren** are set before **ngAfterContentInit()** is called.

AfterContentInit is used as given below:

```
import { Component, AfterContentInit, ElementRef } from '@angular/core';
@Component({
  selector: 'app-friend',
  template: ``
})
export class FriendComponent implements AfterContentInit {
  @ContentChild('name') nameRef: ElementRef;
  ngAfterContentInit() {
    console.log(this.nameRef.nativeElement.innerHTML);
  }
}
```

Example 1: @ContentChild and @ContentChildren using Directive:

First we will create a directive with selector as element name.

Directive Class (**book.directive.ts**):

```
import { Directive, Input } from '@angular/core';
@Directive({
  selector: 'book'
})
export class BookDirective {
  @Input() bookId: string;
  @Input() bookName: string;
}
```

In the above directive we have used two `@Input()` properties. `<book>` element can be used in any component. Now create a component to use `@ContentChild` decorator to query element of type `<book>`.

Component Class (**writer.component.ts**):

```
import { Component, ContentChild } from '@angular/core';
import { BookDirective } from './book.directive';
@Component({
  selector: 'app-writer',
  template: `
    Name: {{writerName}}
    <br/>
    Latest Book: {{book?.bookId}} - {{book?.bookName}}
  `
})
export class WriterComponent {
  @ContentChild(BookDirective, { static: false }) book: BookDirective;
  writerName: string = 'David Miller';
}
```

Code snippet of `app.component.html` to use `<app-writer>` and `<book>` elements:

```
<app-writer>
  <book bookId="1" bookName="Angular 8" *ngIf="latestBook"></book>
  <book bookId="2" bookName="TypeScript" *ngIf="!latestBook"></book>
</app-writer>
<br/>
<button (click)="onChangeBook()">Change Book</button>
```

Code snippet of `app.component.ts`:

```
latestBook: boolean = true;
onChangeBook() {
  this.latestBook = (this.latestBook === true)? false : true;
}
```

Now we will create the example of `@ContentChildren` using directive.

Component Class (`favouritebooks.component.ts`):

```
import { Component, ContentChildren, QueryList } from '@angular/core';
import { BookDirective } from './book.directive';
@Component({
  selector: 'app-favouritebooks',
  template: `
    <b>Top Favourite Books</b>
    <ng-template ngFor let-book [ngForOf]="topBooks">
      <br/>{{book.bookId}} - {{book.bookName}}
    </ng-template>

    <br/><b>All Favourite Books</b>
    <ng-template ngFor let-book [ngForOf]="allBooks">
      <br/>{{book.bookId}} - {{book.bookName}}
    </ng-template>
  `
})
export class FavouriteBooksComponent {
  @ContentChildren(BookDirective) topBooks: QueryList<BookDirective>
  @ContentChildren(BookDirective, {descendants: true}) allBooks: QueryList<BookDirective>
}
```

In the above component we are using `@ContentChildren` two times, one with default `descendants` and second with `descendants` with `true` value.

Code snippet of `app.component.html` to use `<app-favouritebooks>` and `<book>` elements.

```
<app-favouritebooks>
  <book bookId="1" bookName="ASP.NET MVC"></book>
  <book bookId="2" bookName="ASP.NET Web API"></book>
  <app-favouritebooks>
    <book bookId="3" bookName="JavaScript"></book>
  </app-favouritebooks>
  <app-favouritebooks *ngIf="showAllBook">
    <book bookId="4" bookName="jQuery"></book>
    <book bookId="5" bookName="Node JS"></book>
  </app-favouritebooks>
</app-favouritebooks>
<br/>
<button (click)="onShowAllBooks()">
  <label *ngIf="!showAllBook">Show More</label>
  <label *ngIf="showAllBook">Show Less</label>
</button>
```

Code snippet of `app.component.ts`:

```
showAllBook: boolean = false;
onShowAllBooks() {
  this.showAllBook = (this.showAllBook === true)? false : true;
}
```

Complete Code of AppComponent Class and Html Template:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  latestBook: boolean = true;
  onChangeBook() {
    this.latestBook = (this.latestBook === true)? false : true;
  }
  showAllBook: boolean = false;
  onShowAllBooks() {
    this.showAllBook = (this.showAllBook === true)? false : true;
  }
}
```

app.component.html:

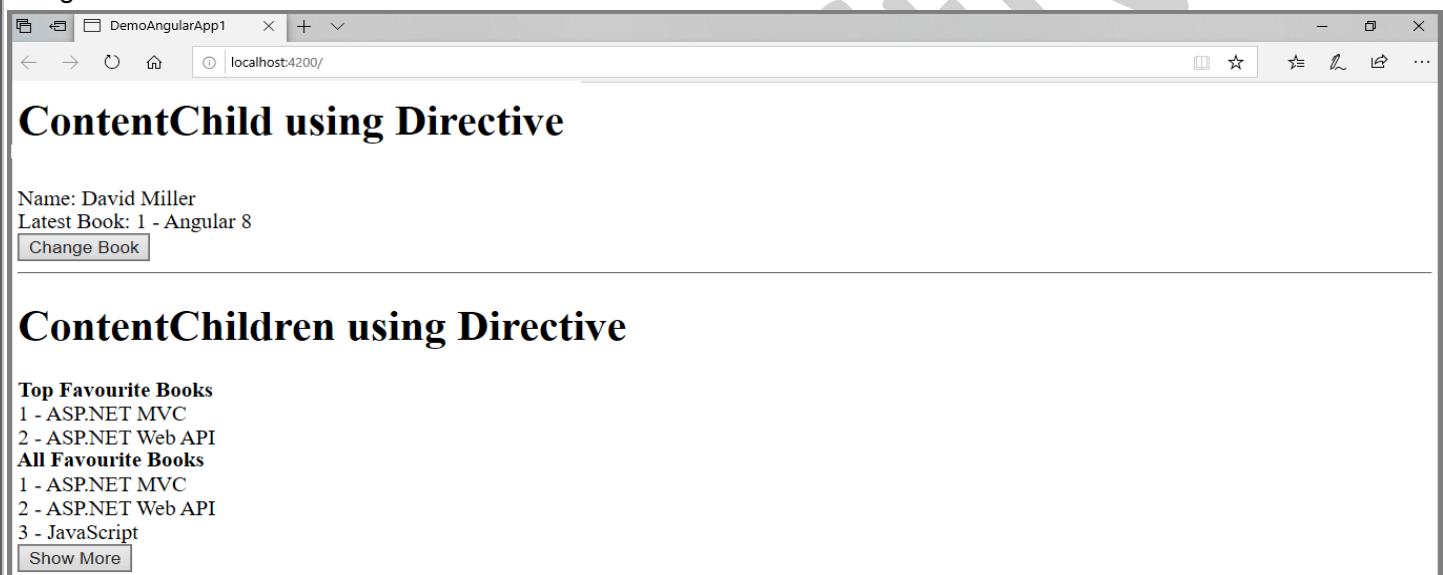
```
<h1>ContentChild using Directive</h1>
<app-writer>
  <book bookId="1" bookName="Angular 8" *ngIf="latestBook"></book>
  <book bookId="2" bookName="TypeScript" *ngIf="!latestBook"></book>
</app-writer>
<br />
<button (click)="onChangeBook()">Change Book</button>
<hr />
<h1>ContentChildren using Directive</h1>
<app-favouritebooks>
  <book bookId="1" bookName="ASP.NET MVC"></book>
  <book bookId="2" bookName="ASP.NET Web API"></book>
  <app-favouritebooks>
    <book bookId="3" bookName="JavaScript"></book>
  </app-favouritebooks>
  <app-favouritebooks *ngIf="showAllBook">
    <book bookId="4" bookName="jQuery"></book>
    <book bookId="5" bookName="Node JS"></book>
  </app-favouritebooks>
</app-favouritebooks>
<br />
<button (click)="onShowAllBooks()">
  <label *ngIf="!showAllBook">Show More</label>
  <label *ngIf="showAllBook">Show Less</label>
</button>
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Now Import and Declare a Directive and Components in Root Module (app.module.ts):

```
import { AppComponent } from './app.component';
import { BookDirective } from './book.directive';
import { WriterComponent } from './writer.component';
import { FavouriteBooksComponent } from './favouritebooks.component';
@NgModule({
  declarations: [
    AppComponent, BookDirective, WriterComponent, FavouriteBooksComponent
  ],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Now run the application and find the print screen of the output of **@ContentChild** and **@ContentChildren** decorators using directive.



The screenshot shows two examples of Angular directives:

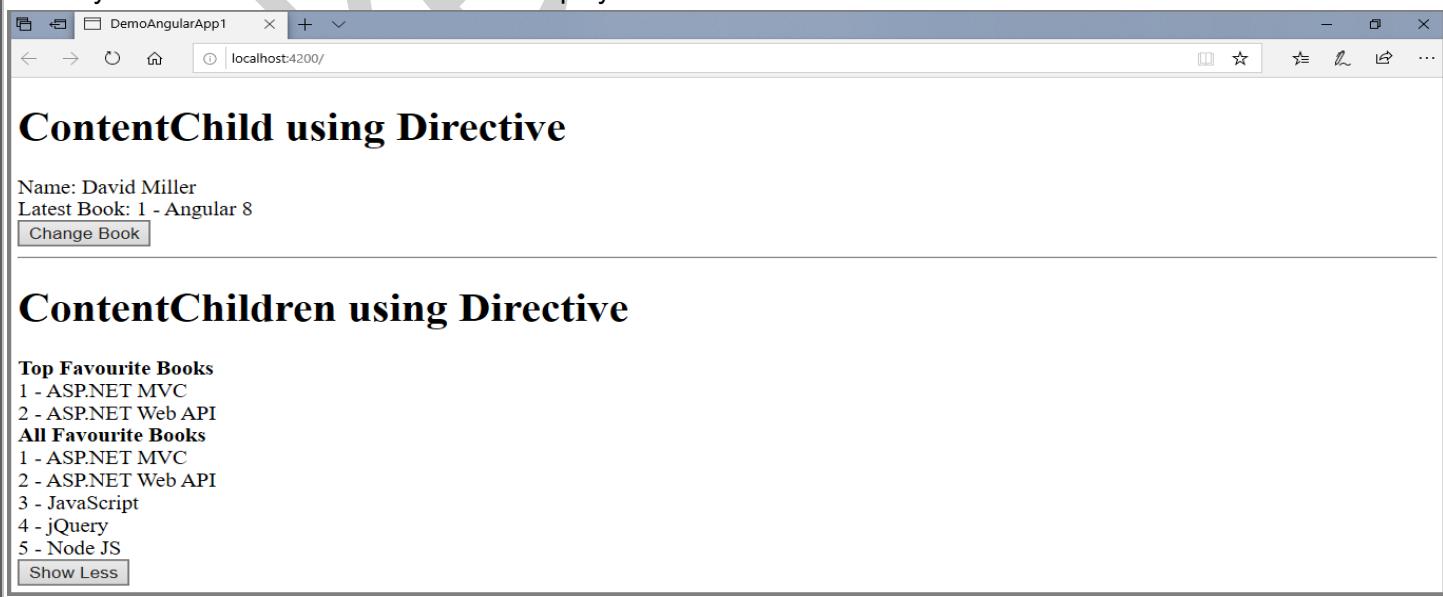
ContentChild using Directive

Name: David Miller
Latest Book: 1 - Angular 8
[Change Book](#)

ContentChildren using Directive

Top Favourite Books
1 - ASP.NET MVC
2 - ASP.NET Web API
All Favourite Books
1 - ASP.NET MVC
2 - ASP.NET Web API
3 - JavaScript
[Show More](#)

Now if you click 'Show More' button it will display all favourite books as shown below:



The screenshot shows the expanded list of favourite books using the **Show More** button:

ContentChild using Directive

Name: David Miller
Latest Book: 1 - Angular 8
[Change Book](#)

ContentChildren using Directive

Top Favourite Books
1 - ASP.NET MVC
2 - ASP.NET Web API
All Favourite Books
1 - ASP.NET MVC
2 - ASP.NET Web API
3 - JavaScript
4 - jQuery
5 - Node JS
[Show Less](#)

Example 2: @ContentChild and @ContentChildren using Component:

Example of `@ContentChild` and `@ContentChildren` decorators using component. Here for child element we will create a component instead of directive.

Component Class (`city.component.ts`):

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'app-city',
  template: ''
})
export class CityComponent {
  @Input() cityId: string;
  @Input() cityName: string;
}
```

In the above component, we have used two `@Input()` properties. Now find the component that will use `@ContentChild`.

Component Class (`address.component.ts`):

```
import { Component, ContentChild } from '@angular/core';
import { CityComponent } from './city.component';

@Component({
  selector: 'app-address',
  template: `
    <b>{{title}}</b>
    <br/>
    City: {{city?.cityId}} - {{city?.cityName}}
  `
})
export class AddressComponent {
  @ContentChild(CityComponent,{static:false}) city: CityComponent;
  title: string = 'Address';
}
```

Code snippet of `app.component.html` to use `<app-address>` and `<app-city>` element.

```
<h1>ContentChild using Component</h1>
<app-address>
  <app-city cityId="1" cityName="Varanasi" *ngIf="homeTown"></app-city>
  <app-city cityId="2" cityName="Hyderabad" *ngIf="!homeTown"></app-city>
</app-address>
<br/>
<button (click)="onChangeCity()">Change City</button>
```

Code snippet of `app.component.ts`.

```
homeTown = true;
onChangeCity() {
  this.homeTown = (this.homeTown === true)? false : true;
}
```

Example for `@ContentChildren` using component:

Component Class (favouritecities.component.ts):

```
import { Component, ContentChildren, QueryList } from '@angular/core';
import { CityComponent } from './city.component';
@Component({
  selector: 'app-favouritecities',
  template: `
    <b>Top Favourite Cities</b>
    <ng-template ngFor let-city [ngForOf]="topCities">
      <br/>{{city.cityId}} - {{city.cityName}}
    </ng-template>
    <br/>
    <b>All Favourite Cities</b>
    <ng-template ngFor let-city [ngForOf]="allCities">
      <br/>{{city.cityId}} - {{city.cityName}}
    </ng-template>
  `
})
export class FavouriteCitiesComponent {
  @ContentChildren(CityComponent) topCities: QueryList<CityComponent>
  @ContentChildren(CityComponent, {descendants: true}) allCities: QueryList<CityComponent>
}
```

Code snippet of `app.component.html` to use `<app-favouritecities>` and `<app-city>` element.

```
<h1>ContentChild using Component</h1>
<app-address>
  <app-city cityId="1" cityName="Varanasi" *ngIf="homeTown"></app-city>
  <app-city cityId="2" cityName="Hyderabad" *ngIf="!homeTown"></app-city>
</app-address>
<br/>
<button (click)="onChangeCity()">Change City</button>
<hr />
<h1>ContentChildren using Component</h1>
<app-favouritecities>
  <app-city cityId="1" cityName="Pune"></app-city>
  <app-city cityId="2" cityName="Mumbai"></app-city>
  <app-favouritecities>
    <app-city cityId="3" cityName="New Delhi"></app-city>
  </app-favouritecities>
  <app-favouritecities *ngIf="showAllCity">
    <app-city cityId="4" cityName="Bengaluru"></app-city>
    <app-city cityId="5" cityName="Chennai"></app-city>
  </app-favouritecities>
</app-favouritecities>
<br/>
<button (click)="onShowAllCities()" >
  <label *ngIf="!showAllCity">Show More</label>
  <label *ngIf="showAllCity">Show Less</label>
</button>
```

Code snippet of `app.component.ts`:

```
showAllCity = false;
onShowAllCities() {
  this.showAllCity = (this.showAllCity === true)? false : true;
}
```



Complete Code of AppComponent Class and Html Template:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  homeTown = true;
  onChangeCity() {
    this.homeTown = (this.homeTown === true)? false : true;
  }
  showAllCity = false;
  onShowAllCities() {
    this.showAllCity = (this.showAllCity === true)? false : true;
  }
}
```

app.component.html:

```
<h1>ContentChild using Component</h1>
<app-address>
  <app-city cityId="1" cityName="Varanasi" *ngIf="homeTown"></app-city>
  <app-city cityId="2" cityName="Hyderabad" *ngIf="!homeTown"></app-city>
</app-address>
<br/>
<button (click)="onChangeCity()">Change City</button>
<hr />
<h1>ContentChildren using Component</h1>
<app-favouritecities>
  <app-city cityId="1" cityName="Pune"></app-city>
  <app-city cityId="2" cityName="Mumbai"></app-city>
<app-favouritecities>
  <app-city cityId="3" cityName="New Delhi"></app-city>
</app-favouritecities>
<app-favouritecities *ngIf="showAllCity">
  <app-city cityId="4" cityName="Bengaluru"></app-city>
  <app-city cityId="5" cityName="Chennai"></app-city>
</app-favouritecities>
</app-favouritecities>
<br/>
<button (click)="onShowAllCities()" >
  <label *ngIf="!showAllCity">Show More</label>
  <label *ngIf="showAllCity">Show Less</label>
</button>
```

Now Import and Declare the Components in Root Module (app.module.ts):

```
import { AppComponent } from './app.component';
import { CityComponent } from './city.component';
import { AddressComponent } from './address.component';
import { FavouriteCitiesComponent } from './favouritecities.component';
@NgModule({
  declarations: [
    AppComponent, CityComponent, AddressComponent, FavouriteCitiesComponent
  ],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Now run the application and find the print screen of the output of **@ContentChild** and **@ContentChildren** decorators using component.

ContentChild using Component

Address
City: 1 - Varanasi
Change City

Top Favourite Cities
1 - Pune
2 - Mumbai

All Favourite Cities
1 - Pune
2 - Mumbai
3 - New Delhi
Show More

Now if you click 'Show More' button it will display all favourite cities as shown below:

ContentChild using Component

Address
City: 1 - Varanasi
Change City

Top Favourite Cities
1 - Pune
2 - Mumbai

All Favourite Cities
1 - Pune
2 - Mumbai
3 - New Delhi
4 - Bengaluru
5 - Chennai
Show Less

Example 3: @ContentChild and @ContentChildren using ElementRef

Example of `@ContentChild` and `@ContentChildren` decorators using `ElementRef`. First find the component that will use `@ContentChild` with `ElementRef`.

Component Class (friend.component.ts):

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

@Component({
  selector: 'app-friend',
  template: `
    Friend Name: {{friendName}}
  `
})

export class FriendComponent implements AfterContentInit {
  @ContentChild('name', {static:false}) nameRef: ElementRef;

  get friendName(): String {
    return this.nameRef.nativeElement.innerHTML;
  }

  ngAfterContentInit() {
    console.log(this.friendName);
  }
}
```

In the above component `name` inside `@ContentChild('name')` is the local template variable of a HTML element.

Code snippet of `app.component.html` to use `<app-friend>` with a `<div>` element.

```
<app-friend>
  <div #name *ngIf="bestFriend">Alina</div>
  <div #name *ngIf="!bestFriend">Martina</div>
</app-friend>
<br/>
<button (click)="onChangeFriend()">Change Friend</button>
```

Code snippet of `app.component.ts`:

```
bestFriend = true;
onChangeFriend() {
  this.bestFriend = (this.bestFriend === true)? false : true;
}
```

Example for `@ContentChildren` with `ElementRef`:

Component Class (`favouritefriends.component.ts`):

```
import { Component, ContentChildren, QueryList, ElementRef, AfterContentInit } from '@angular/core';

@Component({
  selector: 'app-favouritefriends',
  template: `
    <b>All Favourite Friends</b>
    <br />
    {{allFriends}}
  `
})
export class FavouriteFriendsComponent implements AfterContentInit {
  @ContentChildren('name') allFriendsRef: QueryList<ElementRef>;

  get allFriends(): string {
    return this.allFriendsRef ? this.allFriendsRef.map(f => f.nativeElement.innerHTML).join(', ') : '';
  }

  ngAfterContentInit() {
    console.log(this.allFriends);
  }
}
```

In the above component `name` inside `@ContentChildren('name')` is the local template variable of a HTML element.

Code snippet of `app.component.html` to use `<app-favouritefriends>` with `<div>` element.

```
<app-favouritefriends>
  <div #name>Mohan</div>
  <div #name>Amit</div>
  <div #name *ngIf="showAllFriend">Narendra</div>
  <div #name *ngIf="showAllFriend">Pawan</div>
</app-favouritefriends>
<br/>
<button (click)="onShowAllFriends()" >
  <label *ngIf="!showAllFriend">Show More</label>
  <label *ngIf="showAllFriend">Show Less</label>
</button>
```

Code snippet of `app.component.ts`:

```
showAllFriend = false;
onShowAllFriends() {
  this.showAllFriend = (this.showAllFriend === true)? false : true;
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



Complete Code of AppComponent Class and Html Template:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})

export class AppComponent {
  bestFriend = true;
  onChangeFriend() {
    this.bestFriend = (this.bestFriend === true)? false : true;
  }

  showAllFriend = false;
  onShowAllFriends() {
    this.showAllFriend = (this.showAllFriend === true)? false : true;
  }
}
```



app.component.html:

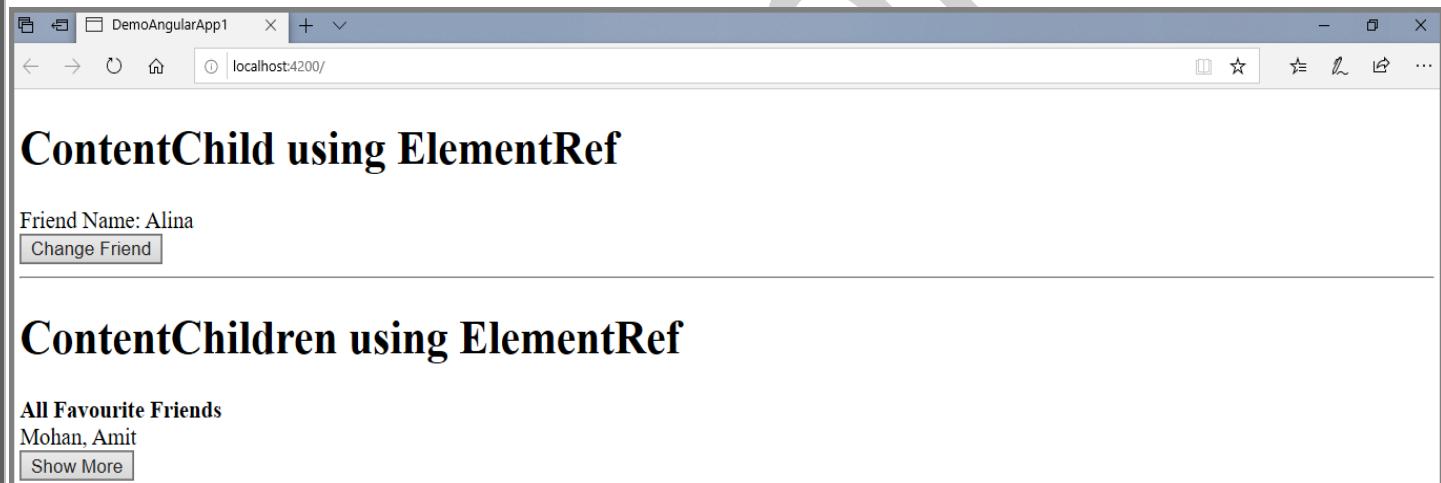
```
<h1>ContentChild using ElementRef</h1>
<app-friend>
  <div #name *ngIf="bestFriend">Alina</div>
  <div #name *ngIf="!bestFriend">Martina</div>
</app-friend>
<br/>
<button (click)="onChangeFriend()">Change Friend</button>
<hr/>
<h1>ContentChildren using ElementRef</h1>
<app-favouritefriends>
  <div #name>Mohan</div>
  <div #name>Amit</div>
  <div #name *ngIf="showAllFriend">Narendra</div>
  <div #name *ngIf="showAllFriend">Pawan</div>
</app-favouritefriends>
<br/>
<button (click)="onShowAllFriends()" >
  <label *ngIf="!showAllFriend">Show More</label>
  <label *ngIf="showAllFriend">Show Less</label>
</button>
```

Now Import and Declare the Components in Root Module (app.module.ts):

```
import { AppComponent } from './app.component';
import { FriendComponent } from './friend.component';
import { FavouriteFriendsComponent } from './favouritefriends.component';

@NgModule({
  declarations: [
    AppComponent, FriendComponent, FavouriteFriendsComponent
  ],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Now run the application and find the print screen of the output of **@ContentChild** and **@ContentChildren** decorators using **ElementRef**:



DemoAngularApp1 > localhost:4200/

ContentChild using ElementRef

Friend Name: Alina

Change Friend

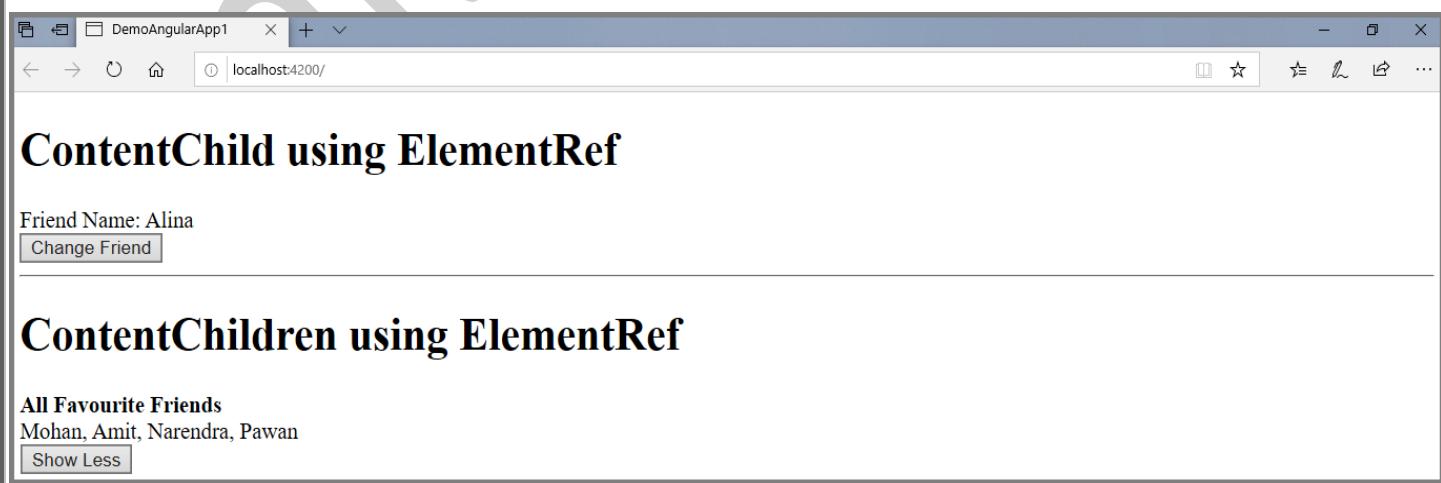
ContentChildren using ElementRef

All Favourite Friends

Mohan, Amit

Show More

Now if you click 'Show More' button it will display all favourite friends as shown below:



DemoAngularApp1 > localhost:4200/

ContentChild using ElementRef

Friend Name: Alina

Change Friend

ContentChildren using ElementRef

All Favourite Friends

Mohan, Amit, Narendra, Pawan

Show Less

@ViewChild():

`@ViewChild()` decorator configures a view query. `@ViewChild()` decorator can be used to get the first element or the directive matching the selector from the view DOM. `@ViewChild()` provides the instance of another component or directive in a parent component and then parent component can access the methods and properties of that component or directive. In this way using `@ViewChild()` components can communicate with a component or directive. In a parent component we can use `@ViewChild()` for components, directives and template reference variable with `ElementRef` or `TemplateRef`. To use `@ViewChild()` we need to pass child component name or directive name or template variable as an argument.

1. Suppose we have a component named as `StopwatchComponent` then within a parent component, we use it as follows:

```
@ViewChild(StopwatchComponent, {static: false}) stopwatchComponent: StopwatchComponent;
```

The selector of the `StopwatchComponent` will be used in parent component HTML template.

2. Now suppose we have a directive `CpColorDirective`, then we use it as follows:

```
@ViewChild(CpColorDirective, {static: false}) cpColorDirective: CpColorDirective;
```

The selector name of directive `CpColorDirective` will be used in host element of DOM layout in parent component HTML template.

3. Now if we have a template reference variable defined in parent component as given below:

```
<input type="text" #title>
```

Then we can use `title` template variable with `@ViewChild()` as given below:

```
@ViewChild('title', {static: false}) elTitle : ElementRef;
```

In the same way we can use `TemplateRef` with template reference variable. Here we will discuss `@ViewChild()` complete example with component, directive and template variable. We will also discuss example to use multiple `@ViewChild()` in parent component. Now let us discuss the example one by one.

@ViewChild() using Component:

`@ViewChild()` can be used for component communication. A component will get instance of another component inside it using `@ViewChild()`. In this way parent component will be able to access the properties and methods of child component. The child component selector will be used in parent component HTML template. Now let us discuss example.

1. Number Example

In this example we will increase and decrease the counter using two buttons. The counter will be from a child component. Increase and decrease button will be inside parent component. We will communicate parent and child components using `@ViewChild()` decorator.

First of all we will create a component where we will write methods to increase and decrease counter.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Component Class (number.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-number',
  template: '<b>{{message}}</b>'
})
export class NumberComponent {
  message:string = '';
  count:number = 0;
  increaseByOne() {
    this.count = this.count + 1;
    this.message = "Counter: " + this.count;
  }
  decreaseByOne() {
    this.count = this.count - 1;
    this.message = "Counter: " + this.count;
  }
}
```

Now we will create the instance of **NumberComponent** in our parent (root) component using **@ViewChild()**.

Component Class (app.component.ts):

```
import { Component, ViewChild } from '@angular/core';
import { NumberComponent } from './number.component';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  @ViewChild(NumberComponent, {static: false}) numberComponent: NumberComponent;
  increase() {
    this.numberComponent.increaseByOne();
  }
  decrease() {
    this.numberComponent.decreaseByOne();
  }
}
```

We will observe that we are able to access the methods of **NumberComponent** in **AppComponent**. We will use selector of **NumberComponent** in HTML template of **AppComponent**.

app.component.html

```
<h3>@ViewChild using Component</h3>
Number Example:
<button type="button" (click)="increase()">Increase</button>
<br/><br/>
<app-number></app-number>
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

As usual both components `NumberComponent` and `AppComponent` need to be configured in `@NgModule` in application module.

Application Root Module (`app.module.ts`):

```
import { AppComponent } from './app.component';
import { NumberComponent } from './number.component';
@NgModule({
  declarations: [
    AppComponent, NumberComponent
  ],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Now run the application and find the print screen of the output of `@ViewChild` decorators using **Component**:

Output:



2. Stopwatch Example:

In this example we will create a simple stopwatch. We will create two components, one of which will contain the functionality of stopwatch and second component will instantiate first component using `@ViewChild()` decorator. Find the component that will contain the functionality of simple stopwatch.

Component Class (`stopwatch.component.ts`):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-stopwatch',
  template: '<h2>{counter}</h2>'
})
export class StopwatchComponent {
  shouldRun:boolean = false;
  counter:number = 0;
  start() {
    this.shouldRun = true;
    let interval = setInterval(() =>
      {
        if(this.shouldRun === false){
          clearInterval(interval);
        }
        this.counter = this.counter + 1;
      }, 1000);
  }
  stop() {
    this.shouldRun = false;
  }
}
```

Now we will instantiate the above component using `@ViewChild()` in the following component.

Component Class (app.component.ts):

```
import { Component, ViewChild } from '@angular/core';
import { StopwatchComponent } from './stopwatch.component';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  @ViewChild(StopwatchComponent, {static: false}) stopwatchComponent: StopwatchComponent;
  startStopwatch() {
    this.stopwatchComponent.start();
  }
  stopStopwatch() {
    this.stopwatchComponent.stop();
  }
}
```

We observe that we are able to access the methods of `StopwatchComponent` in `AppComponent` (Parent Component).

Now use the selector of `StopwatchComponent` in the HTML template of `AppComponent`.

app.component.html:

```
<h3>@ViewChild using Component</h3>
Stopwatch Example:
<button type="button" (click)="startStopwatch()">Start</button>
<br/>
<app-stopwatch></app-stopwatch>
```

As usual both components `StopwatchComponent` and `AppComponent` need to be configured in `@NgModule` in application module.

Application Root Module (app.module.ts):

```
import { AppComponent } from './app.component';
import { StopwatchComponent } from './stopwatch.component';
@NgModule({
  declarations: [
    AppComponent, StopwatchComponent
  ],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Now run the application and find the print screen of the output of `@ViewChild` decorators using `Component`:

Output:



@ViewChild() using Directive:

`@ViewChild()` can instantiate a directive within a component and in this way the component will be able to access the directive methods. Here we will create a directive that will contain the methods to change the text color of the host element of DOM layout. Find the directive.

Directive Class (rsncolor.directive.ts):

```
import { Directive, ElementRef, AfterViewInit } from '@angular/core';
@Directive({
  selector: '[rsnColor]'
})
export class RsnColorDirective implements AfterViewInit{
  constructor(private elRef: ElementRef) {
  }
  ngAfterViewInit() {
    this.elRef.nativeElement.style.color = 'red';
  }
  change(changedColor: string) {
    this.elRef.nativeElement.style.color = changedColor;
  }
}
```

`AfterViewInit`: It is the lifecycle hook that is called after a component view has been fully initialized. To use `AfterViewInit`, our class will implement it and override its method `ngAfterViewInit()`.

Now create the parent (root) component, that will instantiate `RsnColorDirective` and access its methods.

Component Class (app.component.ts):

```
import { Component, ViewChild } from '@angular/core';
import { RsnColorDirective } from './rsncolor.directive';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  @ViewChild(RsnColorDirective, {static: false}) rsnColorDirective: RsnColorDirective;
  changeColor(color: string) {
    this.rsnColorDirective.change(color);
  }
}
```

In the HTML template of component a host element will use directive.

app.component.html:

```
<h3>@ViewChild using Directive</h3>
Color Example:
<p rsnColor>Change my Color</p>
<div>
  Change Color:
  <input type="radio" name="color" (click)="changeColor('green')"> Green
  <input type="radio" name="color" (click)="changeColor('cyan')"> Cyan
  <input type="radio" name="color" (click)="changeColor('blue')"> Blue
</div>
```

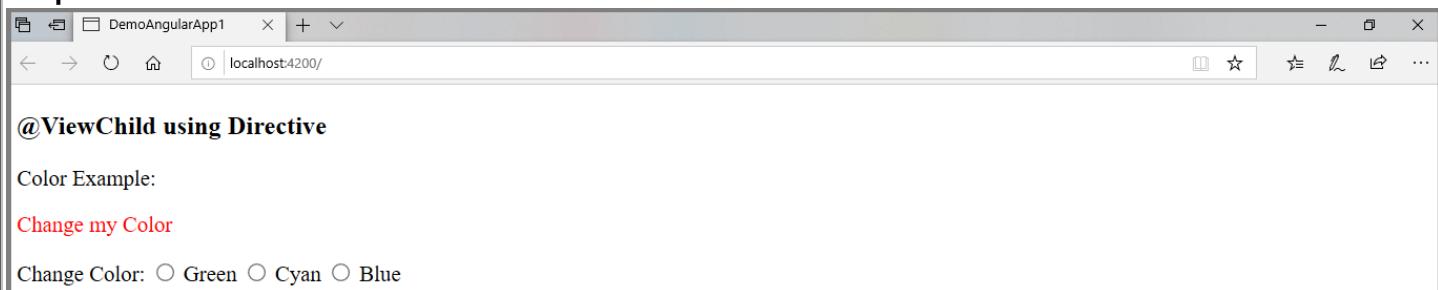
As usual both component and directive need to be configured in `@NgModule` in application module.

Application Root Module (app.module.ts):

```
import { AppComponent } from './app.component';
import { RsnColorDirective } from './rsn-color.directive';
@NgModule({
  declarations: [
    AppComponent, RsnColorDirective
  ],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Now run the application and find the print screen of the output of `@ViewChild` decorators using **Directive**:

Output:



@ViewChild using Directive

Color Example:

[Change my Color](#)

Change Color: Green Cyan Blue

@ViewChild() with Template Variable using ElementRef to access Native Element:

`@ViewChild()` can instantiate `ElementRef` corresponding to a given template reference variable. The template variable name will be passed in `@ViewChild()` as an argument. In this way component will be able to change the appearance and behavior of element of given template variable. Find the HTML template.

app.component.html:

```
<h3>@ViewChild with Template Variable using ElementRef to access Native Element</h3>
<div>
  Name: <br/> <input type="text" #name /> <br/> <br/>
  City: <br/> <input type="text" #city />
</div>
```

In the above HTML template, we have two input boxes and their template reference variables are `#name` and `#city`.

We will instantiate corresponding `ElementRef` using `@ViewChild()` as given below in the component.

Component Class (app.component.ts):

```
import { Component, ViewChild, ElementRef, AfterViewInit } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent implements AfterViewInit {
  @ViewChild('name', {static: false}) elName : ElementRef;
  @ViewChild('city', {static: false}) elCity : ElementRef;
  ngAfterViewInit() {
    this.elName.nativeElement.style.backgroundColor = 'cyan';
    this.elName.nativeElement.style.color = 'red';
    this.elCity.nativeElement.style.backgroundColor = 'cyan';
    this.elCity.nativeElement.style.color = 'red';
  }
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



Now run the application and see the output as given below:

Output:

The screenshot shows a browser window titled "DemoAngularApp1" with the URL "localhost:4200/". The page content is as follows:

```
@ViewChild with Template Variable using ElementRef to access Native Element

Name: 
City: 
```

Using Multiple @ViewChild()

A component can use multiple `@ViewChild()` decorator. We have already seen in the previous example of `@ViewChild()` with template variable where we are using multiple `@ViewChild()` decorator. Here we will combine all our above components and directive and template variable for multiple `@ViewChild()` decorator demo in a parent component.

Component Class (app.component.ts):

```
import { Component, ViewChild, ElementRef, AfterViewInit } from '@angular/core';
import { NumberComponent } from './number.component';
import { StopwatchComponent } from './stopwatch.component';
import { RsnColorDirective } from './rsncolor.directive';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent implements AfterViewInit {
  @ViewChild(NumberComponent, {static: false}) numberComponent: NumberComponent;
  @ViewChild(StopwatchComponent, {static: false}) stopwatchComponent: StopwatchComponent;
  @ViewChild(RsnColorDirective, {static: false}) rsnColorDirective: RsnColorDirective;
  @ViewChild('title', {static: false}) elTitle: ElementRef;
```

```
ngAfterViewInit() {
  this.elTitle.nativeElement.style.backgroundColor = 'cyan';
  this.elTitle.nativeElement.style.color = 'red';
}
increase() {
  this.numberComponent.increaseByOne();
}
decrease() {
  this.numberComponent.decreaseByOne();
}
startStopwatch() {
  this.stopwatchComponent.start();
}
stopStopwatch() {
  this.stopwatchComponent.stop();
}
changeColor(color: string) {
  this.rsnColorDirective.change(color);
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

HTML Template (app.component.html):

```

<h3> Using Multiple @ViewChild </h3>
<b>1. Number Example</b><br />
<button type="button" (click)="increase()">Increase</button>
&nbsp;
<button type="button" (click)="decrease()">Decrease</button>
<br />
<app-number></app-number>

<br /><br />
<b>2. Stopwatch Example</b><br />
<button type="button" (click)="startStopwatch()">Start</button>
&nbsp;
<button type="button" (click)="stopStopwatch()">Stop</button>
<br />
<app-stopwatch></app-stopwatch>

<b>3. Color Example </b><br />
<p rsnColor>Change my Color </p>
<div>
  Change Color:
  <input type="radio" name="color" (click)="changeColor('green')"> Green
  <input type="radio" name="color" (click)="changeColor('cyan')"> Cyan
  <input type="radio" name="color" (click)="changeColor('blue')"> Blue
</div>
<br />
<b>4. Title: <input type="text" #title>
```

Now as usual all components and directives need to be configured in `@NgModule` in application module.

Application Root Module (app.module.ts):

```

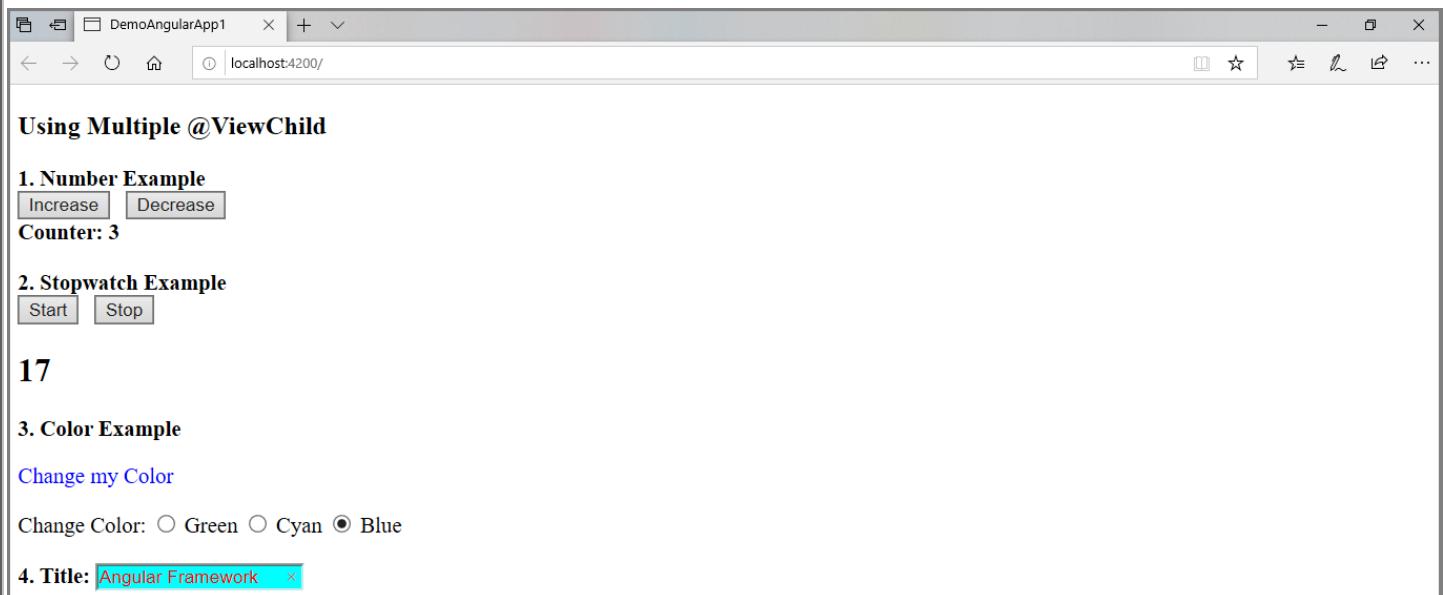
import { AppComponent } from './app.component';
import { NumberComponent } from './number.component';
import { StopwatchComponent } from './stopwatch.component';
import { RsnColorDirective } from './rsncolor.directive';

@NgModule({
  declarations: [
    AppComponent, NumberComponent, StopwatchComponent, RsnColorDirective
  ],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Now run the application and find the print screen of the output of multiple `@ViewChild` decorators using **Component**, **Directive** and **ElementRef**:

Output:



@ViewChildren:

Angular `@ViewChildren` Decorator is used to get the `QueryList` of elements or directives from the view DOM. When a new child element is added or removed, the `QueryList` will be updated and its `changes` function will emit new value. `@ViewChildren` sets the data before `ngAfterViewInit` callback. `@ViewChildren` has following metadata properties.

selector: Selector for querying.

read: It is used to read different token from the queried elements.

`@ViewChildren` supports following selector.

1. A Component class

```
@ViewChildren(WriterComponent) writers: QueryList<WriterComponent>;
```

2. A Directive Class

```
@ViewChildren(MessageDirective) msgList: QueryList<MessageDirective>;
```

3. Template reference variable

```
@ViewChildren('bkWriter') writers: QueryList<WriterComponent>;
@ViewChildren('pname') persons: QueryList<ElementRef>;
```

4. Using `read` metadata.

```
@ViewChildren(WriterComponent, { read: ElementRef }) writers: QueryList<ElementRef>;
@ViewChildren(MessageDirective, {read: ViewContainerRef}) msgList: QueryList<ViewContainerRef>;
```

Here we will discuss `@ViewChildren` complete example with **Component**, **Directive** and **Template Reference Variable**.

@ViewChildren with Component

To use **@ViewChildren** with Component, we need to pass Component name or template reference variable as selector to **@ViewChildren**.

Suppose we have a component **WriterComponent**. We can use **QueryList** of **WriterComponent** in any other component using **@ViewChildren**.

```
@ViewChildren(WriterComponent) writers1: QueryList<WriterComponent>;
```

Find the code to use **read** metadata. Here we will read **ElementRef** from the queried elements.

```
@ViewChildren(WriterComponent, {read: ElementRef}) writers2: QueryList<ElementRef>;
```

Here we will read **ViewContainerRef** from the queried elements.

```
@ViewChildren(WriterComponent,{read: ViewContainerRef}) writers3: QueryList<ViewContainerRef>;
```

We can access **QueryList** inside **ngAfterViewInit** method because **@ViewChildren** sets the data before **AfterViewInit**. Now find the complete code.

Component Class (writer.component.ts):

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'app-writer',
  template: `
    <p>{{writerName}} - {{bookName}}</p>
  `
})
export class WriterComponent {
  @Input('name') writerName: string;
  @Input('book') bookName: string;
}
```

Component Class (app.component.ts):

```
import { Component, ViewChildren, AfterViewInit, ViewContainerRef, QueryList, ElementRef } from '@angular/core';
import { WriterComponent } from './writer.component';
@Component({
  selector: 'app-root',
  template: `
    <h3>@ViewChildren with Component</h3>
    <div>
      <app-writer name="John" book="Angular Tutorials"></app-writer>
      <app-writer name="Smith" book="React Tutorials"></app-writer>
      <app-writer name="David" book="jQuery Tutorials"></app-writer>
      <app-writer name="Peter" book="JavaScript Tutorials" *ngIf="allWritersVisible"></app-writer>
      <app-writer name="Alina" book="TypeScript Tutorials" *ngIf="allWritersVisible"></app-writer>
    </div>
    <button (click)="onShowAllWriters()">
      <label *ngIf="!allWritersVisible">Show More</label>
      <label *ngIf="allWritersVisible">Show Less</label>
    </button>
  `
})
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

```

export class AppComponent implements AfterViewInit {
  @ViewChildren(WriterComponent) writers1: QueryList<WriterComponent>;
  @ViewChildren(WriterComponent, { read: ElementRef }) writers2: QueryList<ElementRef>;
  @ViewChildren(WriterComponent, { read: ViewContainerRef }) writers3: QueryList<ViewContainerRef>;
  allWritersVisible = false;

  ngAfterViewInit() {
    console.log('--- @ViewChildren with Component ---');
    this.writers1.forEach(writer => console.log(writer));
    console.log('Count:', this.writers1.length);

    this.writers1.changes.subscribe(list => {
      list.forEach(writer => console.log(writer.writerName + ' - ' + writer.bookName));
      console.log('Count:', this.writers1.length);
    });

    console.log("Result with ElementRef:");
    this.writers2.forEach(elRef => console.log(elRef));
    console.log('Count:', this.writers2.length);

    console.log("Result with ViewContainerRef:");
    this.writers3.forEach(vcRef => console.log(vcRef));
    console.log('Count:', this.writers3.length);
  }

  onShowAllWriters() {
    this.allWritersVisible = (this.allWritersVisible === true) ? false : true;
  }
}

```

Now `WriterComponent` and `AppComponent` need to be configured in `@NgModule` in application root module.

Application Root Module (`app.module.ts`):

```

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

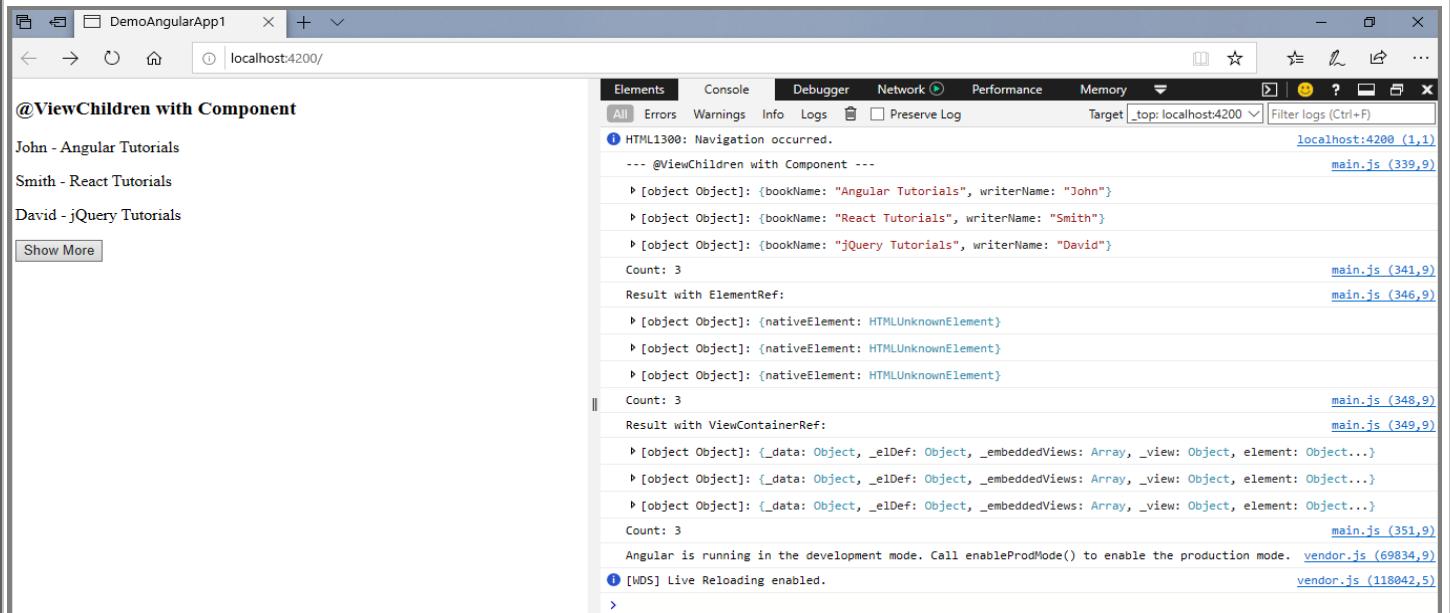
@NgModule({
  declarations: [
    AppComponent, WriterComponent
  ],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})

export class AppModule { }

```

Now run the application and find the print screen of the output of `@ViewChildren` decorators using Component:

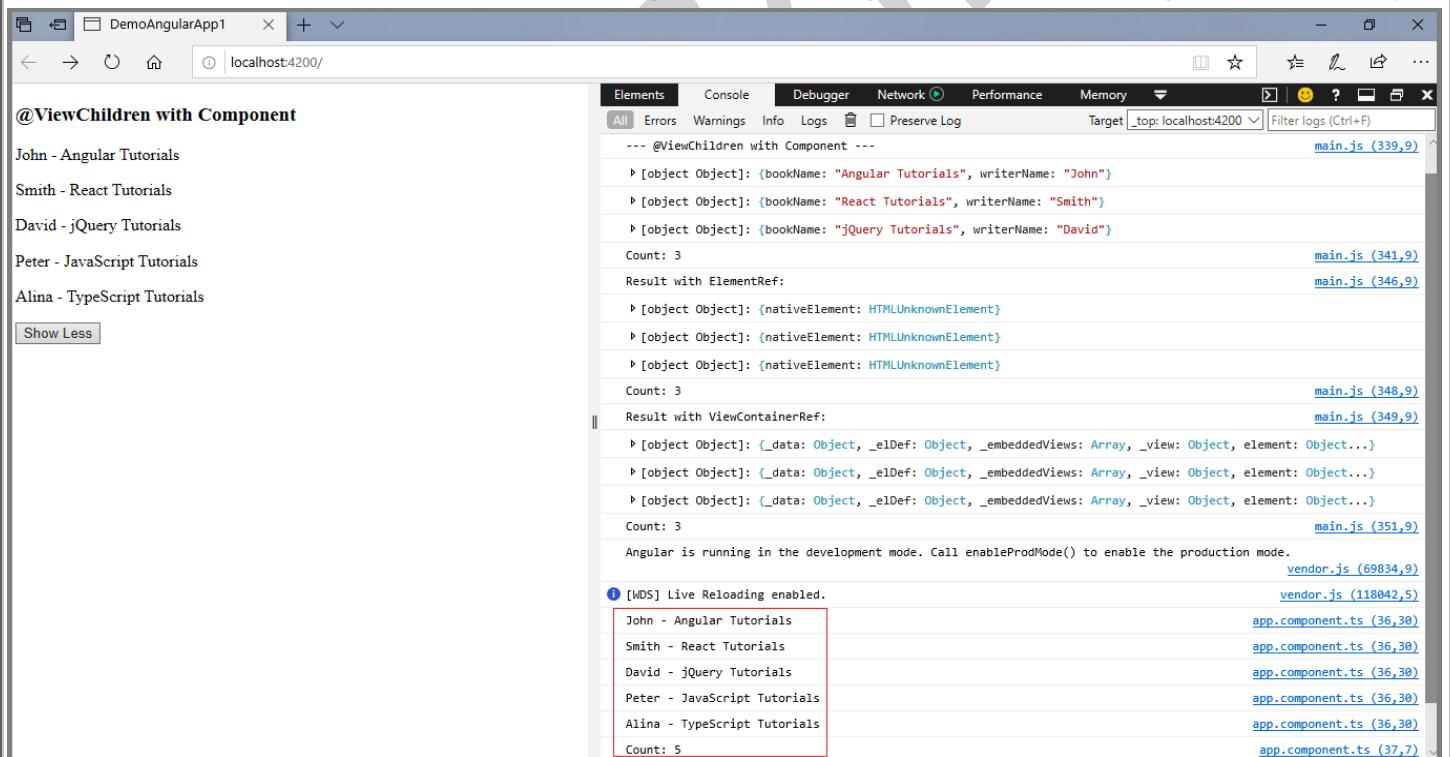
Output:



The screenshot shows a browser window titled "DemoAngularApp1" at "localhost:4200". The page content displays a list of tutorials: "John - Angular Tutorials", "Smith - React Tutorials", "David - jQuery Tutorials", and "Peter - JavaScript Tutorials". Below this is a "Show More" button. The browser's developer tools (Console tab) show the following log entries related to the component structure:

- HTML1300: Navigation occurred.
- @ViewChildren with Component ---
- Count: 3
- Result with ElementRef:
- Count: 3
- Result with ViewContainerRef:
- Count: 3
- Angular is running in the development mode. Call enableProdMode() to enable the production mode.
- [WDS] Live Reloading enabled.

On click of button, the size of `QueryList` will be changed and this change can be observed using `QueryList.changes`.



The screenshot shows the same browser setup. After clicking a button, the list of tutorials now includes "Alina - TypeScript Tutorials" and "Peter - JavaScript Tutorials". The "Show Less" button is visible. The developer tools log shows the following changes:

- @ViewChildren with Component ---
- Count: 3
- Result with ElementRef:
- Count: 3
- Result with ViewContainerRef:
- Count: 3
- Angular is running in the development mode. Call enableProdMode() to enable the production mode.
- [WDS] Live Reloading enabled.

A red box highlights the new entries in the list: "Alina - TypeScript Tutorials" and "Peter - JavaScript Tutorials".

We can also use template reference variable with `@ViewChildren` to obtain `QueryList` of Component. Suppose we have following HTML code.

```
<app-writer name="John" book="Angular Tutorials" #bkWriter></app-writer>
<app-writer name="Smith" book="React Tutorials" #bkWriter></app-writer>
<app-writer name="David" book="jQuery Tutorials" #bkWriter></app-writer>
```

In the above code we have used `bkWriter` as template reference variable for `WriterComponent`. Now we will query data as following:

```
@ViewChildren('bkWriter') writers1: QueryList<WriterComponent>;
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

@ViewChildren with Directive:

We will query Directive using `@ViewChildren` and embed a message using `createEmbeddedView` of `ViewContainerRef` class.

Directive Class (`message.directive.ts`):

```
import { Directive, ViewContainerRef } from '@angular/core';
@Directive({
  selector: '[rsnMsg]'
})
export class MessageDirective {
  constructor(public vcRef: ViewContainerRef) { }
}
```

Component Class (`app.component.ts`):

```
import { Component, ViewChild, ViewChildren, AfterViewInit,
         ViewContainerRef, TemplateRef, QueryList } from '@angular/core';
import { MessageDirective } from './message.directive';
@Component({
  selector: 'app-root',
  template: `
    <h3>@ViewChildren with Directive</h3>
    <div rsnMsg></div>
    <div rsnMsg></div>
    <div rsnMsg></div>

    <ng-template #msgTemp>
      Welcome to World of Angular!
    </ng-template>
  `
})
export class AppComponent implements AfterViewInit {
  @ViewChildren(MessageDirective) msgList: QueryList<MessageDirective>;
  @ViewChild('msgTemp',{static:false}) msgTempRef: TemplateRef<any>;
  ngAfterViewInit() {
    console.log('--- @ViewChildren with Directive ---');
    console.log("ViewChildren Length:", this.msgList.length);
    this.msgList.forEach(messageDirective =>
      messageDirective.vcRef.createEmbeddedView(this.msgTempRef));
  }
}
```

We can also use `read` metadata to read `ViewContainerRef` directly from the directive.

```
@ViewChildren(MessageDirective, {read: ViewContainerRef}) msgList: QueryList<ViewContainerRef>;
ngAfterViewInit() {
  this.msgList.forEach(vcRef => vcRef.createEmbeddedView(this.msgTempRef));
}
```

Now `MessageDirective` and `AppComponent` need to be configured in `@NgModule` in application root module.

Application Root Module (`app.module.ts`):

```
import { AppComponent } from './app.component';
import { MessageDirective } from './message.directive';
@NgModule({
  declarations: [
    AppComponent, MessageDirective
  ],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Now run the application and find the print screen of the output of `@ViewChildren` decorators using `Directive`:

Output:

The screenshot shows a browser window with the URL `localhost:4200`. The page displays the text "Welcome to World of Angular!" three times. In the browser's developer tools, the console tab is open, showing the following logs:

- HTML1300: Navigation occurred.
- @ViewChildren with Directive ---
- ViewChildren Length: 3
- Angular is running in the development mode. Call enableProdMode() to enable the production mode.
- [WDS] Live Reloading enabled.

@ViewChildren with ElementRef:

We will use `@ViewChildren` with `ElementRef` here.

Component Class (`app.component.ts`):

```
import { Component, ViewChildren, AfterViewInit, QueryList, ElementRef } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    

### @ViewChildren with ElementRef</h3> <div> <div #pname>David</div> <div #pname>Smith</div> <div #pname>Peter</div> </div> ` }) export class AppComponent implements AfterViewInit { @ViewChildren('pname') persons: QueryList<ElementRef>; ngAfterViewInit() { console.log('--- @ViewChildren with ElementRef ---'); console.log('ViewChildren Length:', this.persons.length); this.persons.forEach(elRef => console.log(elRef.nativeElement.innerText)); } }


```

In the above code we have some `<div>` and we have assigned template reference variable to it and we are querying `ElementRef`.

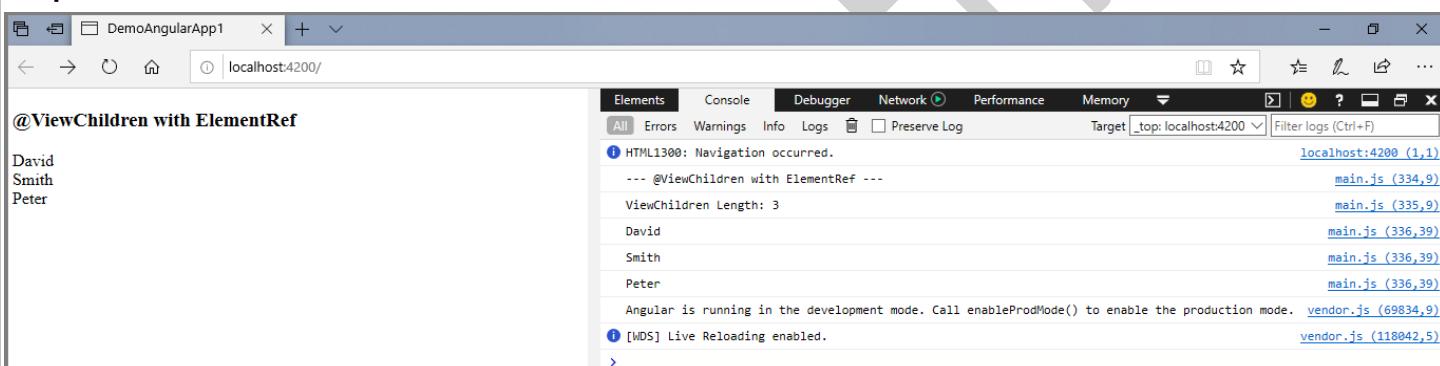
Now `AppComponent` need to be configured in `@NgModule` in application root module.

Application Root Module (app.module.ts):

```
import { AppComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Now run the application and find the print screen of the output of `@ViewChildren` decorators using `ElementRef`:

Output:



The screenshot shows a browser window with the URL `localhost:4200`. The page content is titled `@ViewChildren with ElementRef` and contains the names `David`, `Smith`, and `Peter` listed vertically. Below the browser window is its developer tools console. The console has tabs for Elements, Console, Debugger, Network, Performance, and Memory. The Console tab is active. It shows the following output:

```
HTML1300: Navigation occurred.
--- @ViewChildren with ElementRef ---
ViewChildren Length: 3
David
Smith
Peter
Angular is running in the development mode. Call enableProdMode() to enable the production mode.
[WD] Live Reloading enabled.
```

@ViewChild vs @ViewChildren vs @ContentChild vs @ContentChildren

`@ViewChild` and `@ViewChildren` query the elements from view DOM and `@ContentChild` and `@ContentChildren` query the elements from content DOM.

1 `@ViewChild` queries a single element or directive. It will be first element or the Directive matching the selector from the view DOM. It is used as following.

```
@ViewChild(WriterComponent, {static: false/true}) writer: WriterComponent;
@ViewChild(MessageDirective, {static: false/true}) message: MessageDirective;
```

2. `@ViewChildren` is used to get the `QueryList` of elements or directives from the view DOM. It is used as following.

```
@ViewChildren(WriterComponent) writers: QueryList<WriterComponent>;
@ViewChildren(MessageDirective) msgList: QueryList<MessageDirective>;
```

3. `@ContentChild` gives the first element or directive matching the selector from the content DOM. It is used as following.

```
@ContentChild(CityComponent, {static: false/true}) city: CityComponent;
@ContentChild(BookDirective, {static: false/true}) book: BookDirective;
```

4. `@ContentChildren` is used to get `QueryList` of elements or directives from the content DOM. It is used as following.

```
@ContentChildren(CityComponent) cities: QueryList<CityComponent>
@ContentChildren(BookDirective) books: QueryList<BookDirective>
```



<ng-template>, <ng-container>, <ng-content>, *ngTemplateOutlet in Angular: <ng-template> in Angular:

<ng-template> is an angular element which us used for rendering HTML templates. It is never displayed directly. It can be displayed using structural directive, **ViewContainerRef** etc. Suppose we have following code in our HTML template.

```
<ng-template>
  <p>Hello Angular !!!</p>
</ng-template>
```

When the code runs then the code written inside <ng-template> will not be displayed but there will be a comment.

<!-- -->

What is ng-template in Angular?

1. ng-template is a virtual element and its contents are displayed only when needed (based on conditions).
2. ng-template should be used along with structural directives like [ngIf], [ngFor], [NgSwitch] or custom structural directives. That is why in the above example the contents of ng-template are not displayed.
3. ng-template never meant to be used like other HTML elements. It's an internal implementation of Angular's structural directives.
4. When you use a structural directive in Angular we will add a prefix asterisk(*) before the directive name. This asterisk is short hand notation for *ng-template*.
5. Whenever Angular encounter with the asterisk(*) symbol, we are informing Angular saying that it is a structural directive and Angular will convert directive attribute to *ng-template* element.
6. ng-template is not exactly a true web element. When we compile our code, we will not see a ng-template tag in HTML DOM.
7. Angular will evaluate the ng-template element to convert it into a comment section in HTML DOM.

Before rendering HTML, angular replaces <ng-template> with a comment. <ng-template> can be used with structural directive, **ViewContainerRef** etc.

In our example we will discuss how to use <ng-template> with **ngFor**, **ngIf** and **ngSwitch**. We will also create custom structural directive that we will use with <ng-template>. Using **TemplateRef** and **ViewContainerRef** we can reuse the code of <ng-template> in our HTML template.

<ng-template> example with **ngFor** is as follows:

```
<ng-template ngFor let-person [ngForOf]="persons" let-i="index">
  <p>{{i + 1}}. {{person.name}} : {{person.age}} </p>
</ng-template>
```

1. <ng-template> with ngFor

Find the example to use <ng-template> with **ngFor**.

Component Class (app.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  persons = [
    {name: 'David', age: '25'},
    {name: 'Smith', age: '23'},
    {name: 'Peter', age: '23'},
    {name: 'Martin', age: '21'},
  ];
}
```

HTML Template (app.component.html):

```
<h3>ng-template with ngFor</h3>
<ng-template ngFor let-person [ngForOf]= "persons" let-i="index">
  <p>{{i + 1}}. {{person.name}} : {{person.age}} </p>
</ng-template>
```

ng-template with ngFor

1. David : 25
2. Smith : 23
3. Peter : 23
4. Martin : 21

`let` declares template input variable. The properties of `ngFor` such as `index`, `first`, `last`, `odd`, `even` can be assigned to a variable using `let` within the `<ng-template>`. Find the sample code.

```
<ng-template ngFor let-item [ngForOf]="items" let-i="index" let-o="odd" let-e="even"
    let-f="first" let-l="last" [ngForTrackBy]="trackByFn">
    ...
</ng-template>
```

You might be thinking that why we need to use asterisk(*) notation when we can use `ng-template` element directly. Yes we can use `ng-template` instead of short hand notation.

*ngIf is a simple directive without much complexity. This asterisk notation or microsyntax (in Angular terms) is very useful incase complex structural directives like *ngFor. Take a look at the below example:

```
<p *ngFor="let person of persons; let i=index">
    {{i + 1}}. {{person.name}} : {{person.age}}
</p>

<ng-template ngFor let-person [ngForOf]= "persons" let-i="index">
    <p>{{i + 1}}. {{person.name}} : {{person.age}} </p>
</ng-template>
```

With the asterisk notation or Angular microsyntax we can give instructions to the directive in simple string format. And Angular microsyntax parser convert that string into the attributes of `ng-template` as shown above.

We don't need to get into the implementation details of *ngFor. All we need to understand is asterisk (*) notation easy to write and understand. Until unless you have good reason, prefer asterisk(*) notation instead of `ng-template`.

2. `<ng-template>` with `nglf`

Here we will provide example to use `<ng-template>` with `nglf`, `nglf-else` and `nglf-then-else`.

Component Class (app.component.ts):

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  toggleFlag1= true;
  toggleFlag2= true;
  toggleFlag3= true;

  onToggle1() {
    this.toggleFlag1 = (this.toggleFlag1 === true)? false : true;
  }
  onToggle2() {
    this.toggleFlag2 = (this.toggleFlag2 === true)? false : true;
  }
  onToggle3() {
    this.toggleFlag3 = (this.toggleFlag3 === true)? false : true;
  }
}
```

HTML Template (app.component.html):

```
<h3>ng-template with ngIf</h3>
<button type="button" (click)="onToggle1()">Toggle</button>
<ng-template [ngIf]="toggleFlag1">
  <div>Hello World!</div>
```

Shorthand Notation
`<div *ngIf="toggleFlag1">`
 `<div>Hello World!</div>`
`</div>`

```
<ng-template [ngIf]="toggleFlag1"
[ngIfThen]="#thenTextBlock">
</ng-template>
```

```
<ng-template #thenTextBlock>
  <div>Hello World!</div>
</ng-template>
```

ng-template with ngIf-else

```
<button type="button" (click)="onToggle2()">Toggle</button>
<div *ngIf="toggleFlag2; else msgElseBlock">
  <div>Hello World!</div>
</div>
<ng-template #msgElseBlock>
  <div>Else Block: Hello World!</div>
</ng-template>
```

```
<ng-template [ngIf]="toggleFlag2"
[ngIfElse]="#msgElseBlock">
  <div>Hello World!</div>
</ng-template>
<ng-template #msgElseBlock>
  <div>Else Block: Hello World!</div>
</ng-template>
```

ng-template with ngIf-then-else

```
<button type="button" (click)="onToggle3()">Toggle</button>
<div *ngIf="toggleFlag3; then thenBlock else elseBlock">
</div>
<ng-template #thenBlock>
  <div>Then Block: Hello World!</div>
</ng-template>
<ng-template #elseBlock>
  <div>Else Block: Hello World!</div>
</ng-template>
```

```
<ng-template [ngIf]="toggleFlag3"
[ngIfThen]="#thenBlock"
[ngIfElse]="#elseBlock">
</ng-template>
<ng-template #thenBlock>
  <div>Then Block: Hello World!</div>
</ng-template>
<ng-template #elseBlock>
  <div>Else Block: Hello World!</div>
</ng-template>
```

In case of **then** and **else**, we need to assign a local template reference variable to our `<ng-template>` element and then conditionally `ngIf` will display it.

Output:

```
ng-template with ngIf
Toggle
Hello World!

ng-template with ngIf-else
Toggle
Hello World!

ng-template with ngIf-then-else
Toggle
Then Block: Hello World!
```

After Clicking Toggle Button:

```
ng-template with ngIf
Toggle

ng-template with ngIf-else
Toggle
Else Block: Hello World!

ng-template with ngIf-then-else
Toggle
Else Block: Hello World!
```

```
<div *ngIf="toggleFlag1 then thenTextBlock">
    <div>Hello World!</div>
</div>
<ng-template #thenTextBlock>
    <div>Hello World of Angular!</div>
</ng-template>
```

OR

```
<ng-template [ngIf]="toggleFlag1"
[ngIfThen]="#thenTextBlock">
    <div>Hello World!</div>
</ng-template>
<ng-template #thenTextBlock>
    <div>Hello World of Angular!</div>
</ng-template>
```

We can reuse the `#thenTextBlock` template in both places.

Now we will see what happens, if we use both `then` template & inline template in `*ngIf/[ngIf]` directive?

If you see the final output “Hello World of Angular!” will be visible in the webpage.

Inline template will be ignored if `*ngIf/[ngIf]` contains alternative `then` template.

We can change the `then` or `else` template references dynamically at run time by taking advantage of these `[ngIfThen]` and `[ngIfElse]`.

Dynamically Change Nglf Then, Else Templates At Runtime In Angular:

We can take full advantage of `nglfThen`, `nglfElse` template references in `*ngIf` directive to change `then` or `else` templates templates dynamically at runtime.

Component Class (`app.component.ts`):

```
import { Component, OnInit, ViewChild, TemplateRef } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent implements OnInit {
  show: boolean;

  constructor() {
    this.show = true;
  }

  thenTemplate: TemplateRef<any> | null = null;

  @ViewChild('primaryTemplate', { static: true }) primaryTemplate: TemplateRef<any> | null = null;
  @ViewChild('secondaryTemplate', { static: true }) secondaryTemplate: TemplateRef<any> | null = null;

  switchThenTemplate() {
    this.thenTemplate =
      (this.thenTemplate === this.primaryTemplate) ?
        this.secondaryTemplate :
        this.primaryTemplate;
  }

  ngOnInit() {
    this.thenTemplate = this.primaryTemplate;
  }
}
```

HTML Template (app.component.html):

```

<button (click)="show=!show">{{show ? 'hide' : 'show'}}</button>
<br />
<button (click)="switchThenTemplate()">Switch Then Template</button>
show = {{show}}
<div *ngIf="show; then thenTemplate; else elseTemplate">
  Inline template Ignored
</div>
<ng-template #primaryTemplate>
  Default Primary then Template
</ng-template>
<ng-template #secondaryTemplate>
  Secondary then Template
</ng-template>
<ng-template #elseTemplate>
  Else Template
</ng-template>

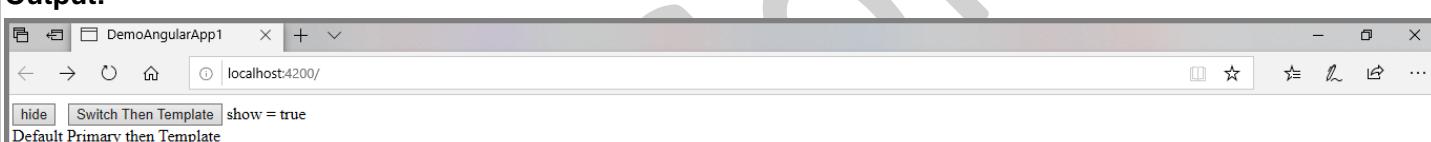
```

```

<ng-template [ngIf]="show"
[ngIfThen]="thenTemplate"
[ngIfElse]="elseTemplate">
<div>
  Inline template Ignored
</div>
</ng-template>

```

Output:



Whenever we click on "Switch Then Template" button then template being toggle between primary and secondary template which is assigned to [ngIfThen] template reference.



Using *ngIf with multiple conditions:

What if we want to use *ngIf with multiple conditions to display an element in the DOM?

And it's very common scenario.

In our traditional programming languages we can use logical operators like AND, OR and NOT inside an if condition.

AND condition in *ngIf

We can use multiple conditions in *ngIf with logical operator AND (&&) to decide the trustworthy of *ngIf expression. If all conditions are true, then element will be added to the DOM.

```

<div *ngIf="isCondition1 && isCondition2">
  <!-- ..Element.. -->
</div>

```

OR condition in *ngIf

Similary If you want to display the element with *ngIf only if one of the condition is true. We can use OR (||) operator.

```

<div *ngIf="isCondition1 || isCondition2">
  <!-- ..Element.. -->
</div>

```

NOT condition in *ngIf

We can use NOT operator(!) to invert the *ngIf condition as shown below

```
<div *ngIf="!isCondition1">
  <!-- ..Element.. -->
</div>
```

We can combine these logical operators with as many as conditions in *ngIf directive.

```
<div *ngIf="(condition1 && condition2) || condition3">
  <!-- ..Element.. -->
</div>
```

Checking null or undefined with *ngIf

We mostly deal with objects while displaying data in Angular.

So it's very common for us to forgot or initialize objects before displaying them in HTML.

In that case we will get "Cannot read property of undefined" errors in console.

Component Class (app.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  productItem: Product;
}
interface Product{
  productId: number;
  productName: string;
}
```

HTML Template (app.component.html):

```
<div>
  {{productItem.productId}}
  {{productItem.productName}}
</div>
```

We can use safe navigation operator ? to check for null or undefined.

```
<div>
  {{productItem?.productId}}
  {{productItem?.productName}}
</div>
```

The problem is for each property we display, we have to use safe navigation operator.

To handle such scenarios, we can check if the object is undefined or null using *ngIf before displaying the data as shown below:

```
<div *ngIf="productItem">
  {{productItem.productId}}
  {{productItem.productName}}
</div>
```



Compare Strings for equality using *ngIf

To compare string for equality we can use double equal(==) or triple equal(==>) operator inside *ngIf expression.

```
<div *ngIf="product.type === 'electronics'>
    <!-- Show electronics products related data -->
</div>
```

It's very common to misplace or forgot to use double equal or triple equal and people tend to use assignment operator (single equal) while compare strings for equality.

You will get "Parser Error: Bindings cannot contain assignments at column" error when you use an assignment operator inside *ngIf as shown below.

```
<div *ngIf="product.type = 'electronics'>
    <!-- Show electronics products related data -->
</div>
```

We should use quotes while comparing with static string inside an *ngIf.

To compare with other component string variables no need to use quotes.

```
<div *ngIf="product.type === typeOfProduct">
    <!-- Show products related data -->
</div>
```

Checking array length with *ngIf

In some cases we want to display array of elements only when array length greater than zero.

In that case to check array length and display else template pass conditional expression `array.length > 0` to *ngIf.

Component Class (app.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  productItems : Product[];
}
interface Product{
  productId: number;
  productName: string;
}
```

HTML Template (app.component.html):

```
<div *ngIf="productItems.length > 0">
    <!-- Show products data -->
</div>
```

If you execute above code you will get "Cannot read property 'length' of undefined" error, because the array is not initialized.

To avoid "Cannot read property 'length' of undefined" error, we have to use safe navigation operator(?) while checking length of array.

```
<div *ngIf="productItems?.length > 0">
    <!-- Show products data -->
</div>
```

Using *ngIf as variable:

In the above example we are using productItem object to display product details in component html file.

Instead of using productItem component variable, we can save it in a local variable (for instance product) using as syntax so that it can be used in component template.

```
<div *ngIf="productItem as product">
    {{product.productId}}
    {{product.productName}}
</div>
```

To use the *ngif as variable in alternative then template or else template, refer the variable prefixed with let keyword in <ng-template> element.

```
<div *ngIf="productItem as product; then productDetails">
</div>

<ng-template #productDetails let-product>
    {{product.id}}
    {{product.description}}
</ng-template>
```

ngIf NullInjectorError: No provider for TemplateRef! error

If you miss writing asterisk before ngIf you will get following error:

**ERROR Error: StaticInjectorError[TemplateRef]:
StaticInjectorError[TemplateRef]: NullInjectorError: No provider for
TemplateRef!**

Summary:

- *ngIf is a directive and can be applied on any HTML or angular component.
- *ngIf evaluates the given condition & then renders the “then template” or “else template”.
- The default templates of *ngIf are “then template” which is inline template of ngIf and “else template” is blank.
- We can write *ngIf else blocks by using <ng-template> element.
- We can use logical operators like AND (&&), OR (||), NOT (!) inside *ngIf to work with multiple conditions.
- We can avoid “Cannot read property of undefined” errors using *ngIf
- While working with observables, We can avoid multiple subscriptions and safe navigation operators by using *ngIf directive.

Difference Between NgIf And Hidden Or Display:none In Angular:

The main difference between angular ngIf directive & hidden or display:none is ngIf will add or remove the element from DOM based on condition or expression. hidden attribute in html5 and display none CSS will show or hide the HTML element.

ngIf in Angular:

Have a look at a simple ngIf example:

```
<div *ngIf="true">This will be added to DOM by ngIf</div>
<div *ngIf="false">This will be removed from DOM by ngIf</div>
```

```
<ng-template [ngIf]="true">
  <div>This will be added to DOM by ngIf</div>
</ng-template>
<ng-template [ngIf]="false">
  <div>This will be removed from DOM by ngIf</div>
</ng-template>
```

As explained in ngIf else angular will convert the `ngIf` element to `<ng-template>` element. See the generated HTML DOM.

```
<!--bindings={ "ng-reflect-ng-if": "true" }-->
<div>
  This will be added to DOM by ngIf
</div>
<!--bindings={ "ng-reflect-ng-if": "false" }-->
```

ngIf vs Hidden:

And the second div is not at all added the DOM. Angular will convert `<ng-template>` to a comment which gives us information about evaluated ngIf condition true or false.

hidden or display:none in Angular

Now we will see an example with hidden attribute in html5 and display none CSS.

```
<p [style.display]="'block'">
  This paragraph is visible.
</p>
<p [style.display]="'none'">
  This paragraph is hidden but still in the DOM.
</p>
<p hidden>
  This paragraph is hidden but still in the DOM.
</p>
```

And the generated HTML is:

```
<p style="display: block;">This paragraph is visible.</p>
<p style="display: none;">
  This paragraph is hidden but still in the DOM.
</p>
<p hidden="">
  This paragraph is hidden but still in the DOM.
</p>
```

ngIf hidden in Angular:

Irrespective of display hide or show the paragraph element will be added to the DOM.

Why ngIf directive remove the element rather than hide it?

Hiding and showing the element after rendering is fine if the paragraph or div element is very small and with simple interaction.

But with Angular, we can build rich applications some of the components may use too many resources. And even though the component is hidden, the component will be attached to its DOM element. It will keep on listening for events. Angular keep on checking for changes related to data bindings. The component behavior still exists even though it is hidden.

The component and its children components will be tie up resources. Memory burden might be high which results in poor performance, responsiveness can degrade and the user has no knowledge, why the application is slow.

So it's better to add or remove elements to the component element to the HTML DOM rather than hiding or showing them.

But if the component is simple it is better to hide or show it, because component reinitialization operation could be expensive.

ngIf vs hidden or display none in Angular:

ngIf	hidden or display none
Angular's structural directive	Normal HTML5 attribute
No DOM element is added if ngIf evaluates to false.	DOM element will be added to HTML
ngIf might be slow while adding removing elements to the DOM because of intializations	As the element is already added to DOM showing and hiding or very fast
Ideal for rich angular applications	Ideal if the component is simple no much interaction

3. <ng-template> with ngSwitch

ngSwitch is not a single directive it is actually used along with other directives *ngSwitchCase and *ngSwitchDefault

Find the example to use `<ng-template>` with `ngSwitch`.

Component Class (app.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  Course: string;
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

HTML Template (app.component.html):

```

<h3>ng-template with ngSwitch</h3>
<input id="rdb1" type="radio" name="course" (click)="Course='Angular'">
<label for="rdb1">Angular</label>&nbsp;
<input id="rdb2" type="radio" name="course" (click)="Course='React JS'">
<label for="rdb2">React JS</label>&nbsp;
<input id="rdb3" type="radio" name="course" (click)="Course='Node JS'">
<label for="rdb3">Node JS</label>&nbsp;
<input id="rdb4" type="radio" name="course" (click)="Course='Vue JS'">
<label for="rdb4">Vue JS</label>
<br/>
<div [ngSwitch]="Course">
  <p *ngSwitchCase="'Angular'">Angular Course</p>
  <p *ngSwitchCase="'React JS'">React JS Course</p>
  <p *ngSwitchCase="'Node JS'">Node JS Course</p>
  <p *ngSwitchCase="'Vue JS'">Vue JS Course</p>
  <p *ngSwitchDefault>No Course Selected</p>
</div>

```

OR Using <ng-template>

```

<div [ngSwitch]="Course">
  <ng-template [ngSwitchCase]="'Angular'"><p>Angular Course</p></ng-template>
  <ng-template [ngSwitchCase]="'React JS'"><p>React JS Course</p></ng-template>
  <ng-template [ngSwitchCase]="'Node JS'"><p>Node JS Course</p></ng-template>
  <ng-template [ngSwitchCase]="'Vue JS'"><p>Vue JS Course</p></ng-template>
  <ng-template ngSwitchDefault><p>No Course Selected</p></ng-template>
</div>

```

Output:



ng-template with ngSwitch

Angular React JS Node JS Vue JS

No Course Selected

If you select a radio button then particular course will display as following:



ng-template with ngSwitch

Angular React JS Node JS Vue JS

Angular Course

4. <ng-template> with TemplateRef and ViewContainerRef

Here we will use `<ng-template>` with `TemplateRef` and `ViewContainerRef`. First we will assign a template reference variable to `<ng-template>` and then we will access its reference as `TemplateRef` using `@ViewChild()`. Now we will fetch the view container reference using a directive. We will fetch all view container references provided by directive using `@ViewChildren()` and `QueryList`. Then we will embed template reference to all view container references obtained by directive. Find the example.

Directive Class (message.directive.ts):

```
import { Directive, ViewContainerRef } from '@angular/core';
@Directive({
  selector: '[rsnMsg]'
})
export class MessageDirective {
  constructor(public vcRef: ViewContainerRef) { }
}
```

Component Class (app.component.ts):

```
import { Component, ViewChild, ViewChildren, AfterViewInit, TemplateRef, ViewContainerRef, QueryList } from
'@angular/core';
import { MessageDirective } from './message.directive';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent implements AfterViewInit{
  @ViewChild('msg', {static:true}) msgTempRef : TemplateRef<any>

  @ViewChildren(MessageDirective) queryList: QueryList<MessageDirective>

  ngAfterViewInit() {
    this.queryList.map(messageDirective =>
      messageDirective.vcRef.createEmbeddedView(this.msgTempRef));
  }
}
```

HTML Template:

```
<h3>ng-template with TemplateRef and ViewContainerRef</h3>
<ng-template #msg>
  Welcome to Angular Class.<br/>
  Happy Learning!!!
</ng-template>

<h3>Message</h3>

<div rsnMsg></div>

<h3>Message</h3>

<div rsnMsg></div>
```

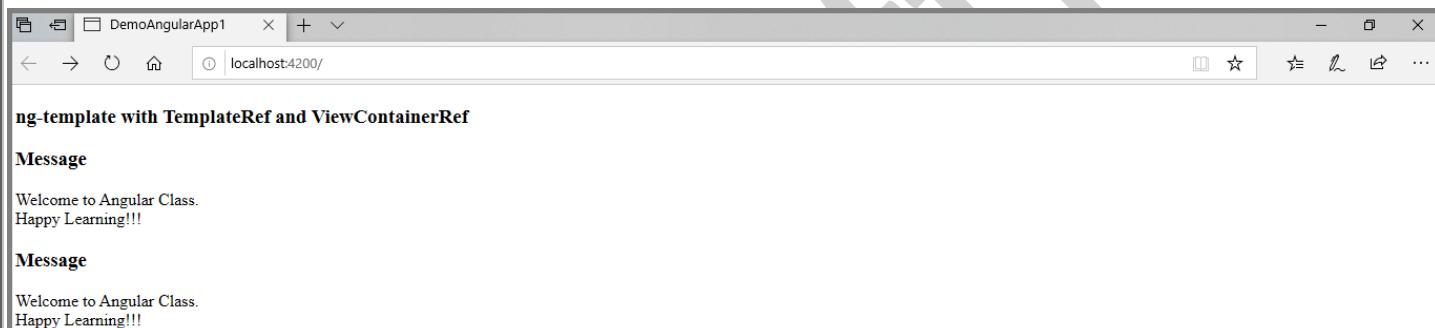
Now `AppComponent` and `MessageDirective` need to be configured in `@NgModule` in application root module.

Application Root Module (`app.module.ts`):

```
import { AppComponent } from './app.component';
import { MessageDirective } from './message.directive';

@NgModule({
  declarations: [
    AppComponent, MessageDirective
  ],
  imports: [...],
  providers: [...],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

When we run application, the HTML code enclosed by `<ng-template>` will be embedded wherever we have used our `rsnMsg` directive in our HTML template.



5. `<ng-template>` with Custom Structural Directive

Here we will provide example of `<ng-template>` with custom structural directive.

Directive Class (`rsnIf.directive.ts`):

```
import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';

@Directive({
  selector: '[rsnIf]'
})
export class RSNIfDirective {
  constructor( public templateRef: TemplateRef<any>, public viewContainer: ViewContainerRef ) {

  }
  @Input() set rsnIf(condition: boolean) {
    if (condition) {
      this.viewContainer.createEmbeddedView(this.templateRef);
    } else {
      this.viewContainer.clear();
    }
  }
}
```

Component Class (app.component.ts):

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  showRsnIf: string = '';
}
```

HTML Template (app.component.html):

```
<h3>ng-template with Custom Structural Directive</h3>
<div>
  Show Message:
  <input type="radio" name="rb" (click)="showRsnIf='yes'"> Yes
  <input type="radio" name="rb" (click)="showRsnIf='no'"> No
</div>
<br />
<div *rsnIf="showRsnIf === 'yes'">
  Hello rsnIf Directive.
</div>
<ng-template [rsnIf]="showRsnIf === 'no'">
  <div>
    Message not Available.
  </div>
</ng-template>
```

Now `AppComponent` and `MessageDirective` need to be configured in `@NgModule` in application root module.

Application Root Module (app.module.ts):

```
import { AppComponent } from './app.component';
import { RSNIfDirective } from './rsn-if.directive';
@NgModule({
  declarations: [
    AppComponent, RSNIfDirective
  ],
  imports: [...],
  providers: [...],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Output:



<ng-container> in Angular:

<ng-container> is an angular element using this element that can host structural directives. ng-container find it's usage when multiple structural directives are to be used together.

Let's consider an example where we are using multiple structural directives :-

Component Class (app.component.ts):

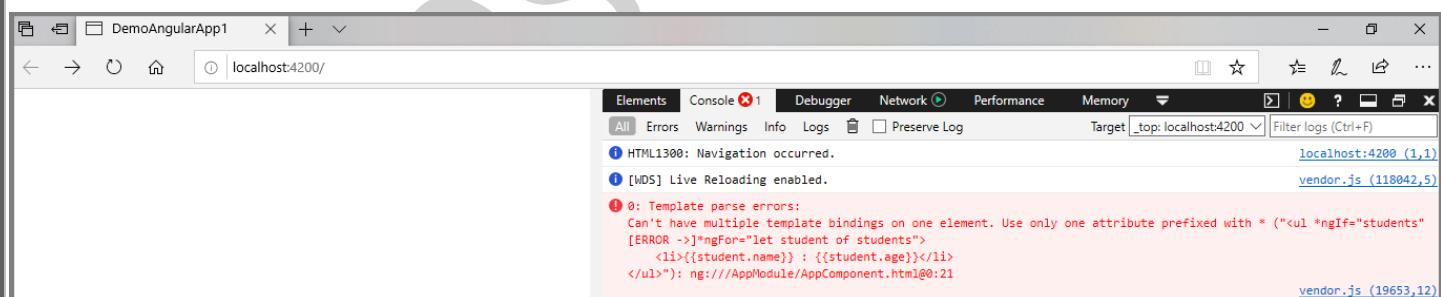
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  students = [
    { name: "David", age: 25 },
    { name: "Peter", age: 23 },
    { name: "Smith", age: 27 },
    { name: "John", age: 21 },
    { name: "Alina", age: 26 }
  ]
}
```

HTML Template (app.component.html):

```
<div *ngIf="students" *ngFor="let student of students">
  <p>{{student.name}} : {{student.age}}</p>
</div>
```

Here in the above code, we have used *ngIf and *ngFor on same element. So it will not work and give the following error in the browser console:-



This shows that two structural directives cannot be used together to the same element. It is saying, you can't have multiple template bindings on one element. Use only one attribute. So to make above code work following changes is to be made:-

HTML Template (app.component.html):

```
<div *ngIf="students">
  <div *ngFor="let student of students">
    <p>{{student.name}} : {{student.age}}</p>
  </div>
</div>
```

Now, check the output in the browser, it will work fine now.

So, what did we do to fix this problem? We have created an additional div with the `*ngIf` directive, and moved the div with `ngFor` inside it.

This solution will work, but we created another div element. There is a way to avoid creating extra element and get the result.

So, here is the solution: `ng-container` directive. Let's see how to use it.

The following behaves exactly the same, but without adding any extra element to the DOM at runtime:

HTML Template (app.component.html):

```
<ng-container *ngIf="students">
  <div *ngFor="let student of students">
    <p>{{student.name}} : {{student.age}}</p>
  </div>
</ng-container>
```

You can see, we have replaced `<div>` with `<ng-container>` directive.

As we can see, the `ng-container` directive provides us with an element that we can attach a structural directive to a section of the page, without having to create an extra element just for that.

Another Example:

Component Class (app.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  employees = [
    { empId: null, empName: null, city: null, experience: null },
    { empId: null, empName: null, city: null, experience: null },
    { empId: 1235, empName: "Michel", city: "London", experience: "7 Years" },
    { empId: 2351, empName: "Peter", city: "Dubai", experience: "8 Years" }
  ];
}
```

HTML Template (app.component.html):

```
<div *ngFor="let emp of employees">
  <div>{{emp.empId}} {{emp.empName}} {{emp.city}} {{emp.experience}}</div>
</div>
  ▲ <div>
    ▲ <div></div>
    </div>
  ▲ <div>
    ▲ <div></div>
    </div>
  ▲ <div>
    <div>1235 Michel London 7 Years</div>
  </div>
  ▲ <div>
    <div>2351 Peter Dubai 8 Years</div>
  </div>
```

Have you seen the above code? Employees array data having two null data rows. The above code will work perfectly but unnecessary first two div will be rendered in UI without having data.

Solution using <ng-container>:-

```
<ng-container *ngFor="let emp of employees">
  <div *ngIf="emp.empId">
    {{emp.empId}} {{emp.empName}} {{emp.city}} {{emp.experience}}
  </div>
</ng-container>

  <!--bindings={ "ng-reflect-ng-for-of": "[object Object],[object Object]" }-->
  <!-->
  <!--bindings={ "ng-reflect-ng-if": null }-->
  <!-->
  <!-->
  <!--bindings={ "ng-reflect-ng-if": null }-->
  <!-->
  <!-->
  <!--bindings={ "ng-reflect-ng-if": "1235" }-->
  <div>1235 Michel London 7 Years</div>
  <!-->
  <!-->
  <!--bindings={ "ng-reflect-ng-if": "2351" }-->
  <div>2351 Peter Dubai 8 Years</div>
  <!-->
```

Using <ng-container> avoiding blank data row display in the UI. Some other scenario described the below example.

```
<table border='1' style="border-collapse:collapse">
  <tr *ngFor="let emp of employees" *ngIf="emp.empId">
    <td>{{emp.empId}}</td>
    <td>{{emp.empName}}</td>
    <td>{{emp.city}}</td>
    <td>{{emp.experience}}</td>
  </tr>
</table>
```

We can not use both statements on the same element as shown in the above code. So write the code as given following below using <ng-container>:-

```
<table border='1' style="border-collapse:collapse">
  <ng-container *ngFor="let emp of employees">
    <tr *ngIf="emp.empId">
      <td>{{emp.empId}}</td>
      <td>{{emp.empName}}</td>
      <td>{{emp.city}}</td>
      <td>{{emp.experience}}</td>
    </tr>
  </ng-container>
</table>
```

Avoiding the use of unnecessary elements like that can help prevent having unwanted styles applied if some of your CSS rules would normally target them.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

<ng-content> in Angular (Content Projection in Angular):

They are used to create configurable components. This means the components can be configured depending on the needs of its user. This is well known as **Content Projection**.

Content Projection (also known as **Transclusion**) is a way to import HTML content from outside the component and insert that content into the component's template in a designated spot.

In Angular, content projection is used to project content in a component. Content projection allows you to insert a shadow DOM in your component. To put it simply, if you want to insert HTML elements or other components in a component, then you do that using the concept of content projection. In Angular, you achieve content projection using **<ng-content></ng-content>**. You can make reusable components and scalable applications by properly using content projection.

To understand content projection, let us consider **GreetComponent** as shown in the code listing below:

Component Class (greet.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-greet',
  template: `{{message}}`
})
export class GreetComponent {
  message: string = "Hello Angular !!!";
}
```

Now if you use **GreetComponent** in another component, and want to pass a greeting message from the parent component, then you should use the **@Input()** decorator. This way, a modified **GreetComponnet** will look like the listing below:

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'app-greet',
  template: `{{message}}`
})
export class GreetComponent {
  @Input() message: string = "Hello Angular !!!";
}
```

Using the **@Input()** decorator, you can pass a simple string to the GreetComponnet, but what if you need to pass different types of data to the **GreetComponent** such as:

1. Inner HTML
2. HTML Elements
3. Styled HTML
4. Another Component, etc.

To pass or project styled HTML or another component, content projection is used. Let us modify the **GreetComponent** to use content projection in this code:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-greet',
  template: `
    <div>
      <ng-content></ng-content>
    </div>
  `
})
export class GreetComponent { }
```

We are using this to project content in the **GreetComponent**. When you use the **GreetComponent** in another component like **AppComponent (Root Component)**, you'll pass content as shown below:

AppComponent (app.component.ts):

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

Html Template (app.component.html):

```
<app-greet>
  <h1>Hello Angular !!!</h1>
</app-greet>
```

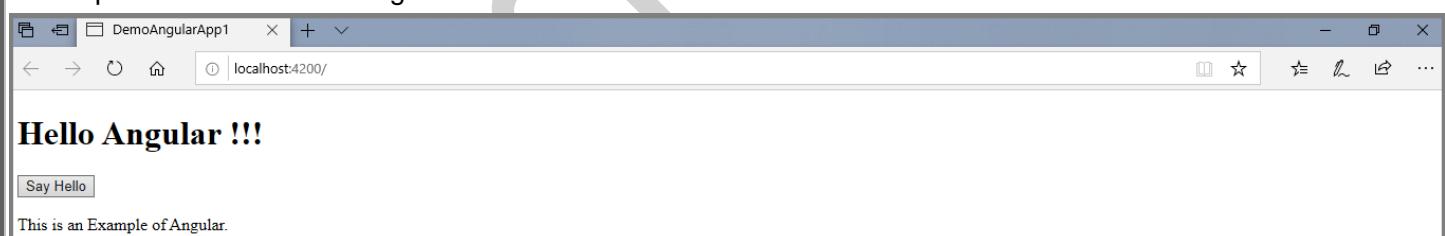
In the listing above, you are projecting styled HTML to the **GreetComponent** and you'll get the following output:



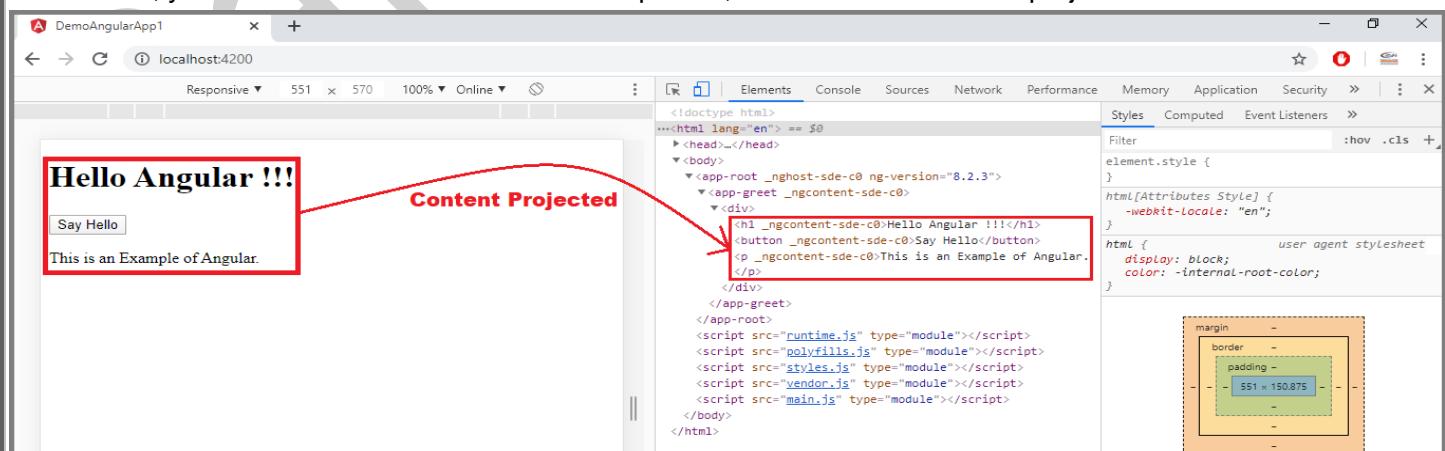
This is an example of **Single Slot Content Projection**. Whatever you pass to the **GreetComponent** will be projected. So, let us pass more than one HTML element to the **GreetComponent** as shown in the listing below:

```
<app-greet>
  <h1>Hello Angular !!!</h1>
  <button>Say Hello</button>
  <p>This is an Example of Angular.</p>
</app-greet>
```

Here we are passing three HTML elements to the **GreetComponent**, and all of them will be projected. You will get the output as shown in the image below:



In the DOM, you can see that inside the **GreetComponent**, all HTML elements are projected.



You may have a requirement to project elements in multiple slots of the component. In this next example, let's say you want to create a greeting card like this:

Angular Training

Welcome to our Training Institution. We are best in service.

[Click Here for More Details !!!](#)

This can be created using the component as shown below:

GreetComponent (greet.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-greet',
  template: `
    <div>
      <h1>Angular Training</h1>
      <p>{{greetText}}</p>
      <button>Click Here for More Details !!!</button>
    </div>
  `
})
export class GreetComponent {
  greetText: string = "Welcome to our Training Institution. We are best in service."
}
```

Let us say we have a requirement to pass the header element and a button element so the header and button can be dynamically passed to the **GreetComponent**. This time, we need two slots:

1. A slot for the header element
2. A slot for the button element

Let's modify the **GreetComponent** to cater to the above requirement as shown in the image below:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-greet',
  template: `
    <div>
      <ng-content></ng-content> ←
      <p>{{greetText}}</p>
      <ng-content></ng-content> ←
    </div>
  `
})
export class GreetComponent {
  greetText: string = "Welcome to our Training Institution. We are best in service."
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Here we're using ng-content two times. Now, the question is, do we select a particular ng-content to project the h1 element and another ng-content to project a button element?

You can select a particular slot for projection using the <ng-content> selector. There are four types of selectors:

1. Project using tag selector.
2. Project using class selector.
3. Project using id selector.
4. Project using attribute selector.

You can use the tag selector for multi-slot projection as shown in the listing below:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-greet',
  template: `
    <div>
      <ng-content select="h1"></ng-content>
      <p>{{greetText}}</p>
      <ng-content select="button"></ng-content>
    </div>
  `
})
export class GreetComponent {
  greetText: string = "Welcome to our Training Institution. We are best in service."
}
```

Next, you can project content to the **GreetComponent** in another component like **AppComponent (Root Component)** as shown below:

AppComponent (app.component.ts):

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

Html Template (app.component.html):

```
<app-greet>
  <h1>Angular Training</h1>
  <button>Click Here for More Details !!!</button>
</app-greet>
<app-greet>
  <h1>ASP.NET MVC Training</h1>
  <button>Click Here for More Details !!!</button>
</app-greet>
```

As you can see, we are using the **GreetComponent** twice and projecting different h1 and button elements. You'll get the output as shown in the image below:



DemoAngularApp1

localhost:4200/

Angular Training

Welcome to our Training Institution. We are best in service.

Click Here for More Details !!!

ASP.NET MVC Training

Welcome to our Training Institution. We are best in service.

Click Here for More Details !!!



The problem with using tag selectors is that all h1 elements will get projected to the **GreetComponent**. In many scenarios, you may not want that and can use other selectors such as a **class selector** or an **attribute selector**, as shown below:

GreetComponent (greet.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-greet',
  template: `
    <div>
      <ng-content select=".headerText"></ng-content>
      <p>{{greetText}}</p>
      <ng-content select="[btn]"></ng-content>
    </div>
  `
})
export class GreetComponent {
  greetText: string = "Welcome to our Training Institution. We are best in service."
}
```

Next, you can project content to the **GreetComponent** as shown below:

Html Template (app.component.html):

```
<app-greet>
  <h1 class="headerText">Angular Training</h1>
  <button btn>Click Here for More Details !!!</button>
</app-greet>
<app-greet>
  <h1 class="headerText">ASP.NET MVC Training</h1>
  <button btn>Click Here for More Details !!!</button>
</app-greet>
```

You'll get the same output as above, however this time you are using the class name and attribute to project the content. When you inspect an element on the DOM, you will find the attribute name and the class name of the projected element as shown in the image below:

Content projection is very useful to insert shadow DOM in your components. To insert HTML elements or other components in a component, you need to use content projection. In AngularJS 1.X, content projection was achieved using **Transclusion**, however, in Angular, it is achieved using **<ng-content>**.

*ngTemplateOutlet in Angular:

*ngTemplateOutlet is used for two scenarios - to insert a common template in various sections of a view irrespective of loops or condition and to make a highly configured component.

An embedded view from a prepared TemplateRef can be inserted via ngTemplateOutlet.

For example, you can define a template for the logo of a company and insert it in several places in the page:

AppComponent Class (app.component.ts):

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  logoSourceUrl: string = '../assets/Images/CompanyLogo.jpg';
  employeeCount: number = 15;
}
```

HTML Template: (app.component.html):

```
<ng-template #companyLogoTemplate>
  <div>
    <img [src]="logoSourceUrl" width="150" height="150" />
    <br />
    <label>At present {{employeeCount}} employees are working here !!!</label>
    <hr />
  </div>
</ng-template>

<div>
  <ng-container *ngTemplateOutlet="companyLogoTemplate"></ng-container>
</div>

<div>
  <ng-container *ngTemplateOutlet="companyLogoTemplate"></ng-container>
</div>

<div>
  <ng-container *ngTemplateOutlet="companyLogoTemplate"></ng-container>
</div>
```

As you can see we just wrote the logo template once and used it three times on the same page with single line of code.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Output:



The image shows three separate browser windows side-by-side, each displaying the same Angular application. The application features a header with the RakeshSoftNet logo and text, followed by a message: "At present 15 employees are working here !!!". The differences between the three windows are in the header configuration:

- Top Window:** Shows a standard header with the RakeshSoftNet logo, text, and a "Real Time Live Project" link.
- Middle Window:** Shows a header where the "Real Time Live Project" link has been removed.
- Bottom Window:** Shows a header where the entire top navigation bar (including the logo and text) has been removed, leaving only the message at the bottom.

Another example, you can define a template for the list of a company and insert it in several places in the page:

AppComponent Class (app.component.ts):

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  companies = [
    {companyId:101, name:'Microsoft', url:'https://www.microsoft.com'},
    {companyId:102, name:'IBM', url:'https://www.ibm.com'},
    {companyId:103, name:'TCS', url:'https://www.tcs.com'},
    {companyId:104, name:'Wipro', url:'https://www.wipro.com'},
    {companyId:105, name:'Infosys', url:'https://www.infosys.com'}
  ];
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

HTML Template (app.component.html):

```

<ng-template #companyListTemplate>
  <b>List of Company</b>
  <table border="1" style="border-collapse: collapse;">
    <tr>
      <th>Company Id</th>
      <th>Name</th>
      <th>Url</th>
    </tr>
    <tr *ngFor="let company of companies">
      <td>{{company.companyId}}</td>
      <td>{{company.name}}</td>
      <td>
        <a [href]="company.url" target="_blank">{{company.url}}</a>
      </td>
    </tr>
  </table>
</ng-template>

<div>
  <ng-container *ngTemplateOutlet="companyListTemplate"></ng-container>
</div>
<hr />
<div>
  <ng-container *ngTemplateOutlet="companyListTemplate"></ng-container>
</div>

```

As you can see we just wrote the company list template once and used it two times on the same page with single line of code.

Output:



The screenshot shows a browser window titled "DemoAngularApp1" with the URL "localhost:4200". There are two identical tables displayed, each with three columns: "Company Id", "Name", and "Url". Both tables contain the same five rows of data, which are:

Company Id	Name	Url
101	Microsoft	https://www.microsoft.com
102	IBM	https://www.ibm.com
103	TCS	https://www.tcs.com
104	Wipro	https://www.wipro.com
105	Infosys	https://www.infosys.com

Template Reference Variable in Angular:

A template reference variable is a reference to a DOM element or directive within a template. Using template reference variable we access the values of DOM element properties. Template reference variable is declared using # and ref- as prefix, for example #myVar and ref-myVar. We should not duplicate template reference variable name because in this way it may give unpredictable value.

Template Reference Variable Syntax:

We can use template reference variable by two ways.

1. Using

Example:

```
<input type="text" #myVar />
```

Here myVar will be a template reference variable.

2. Using ref-

Example:

```
<input type="text" ref-myVar />
```

Here myVar will be a template reference variable.

Template Reference Variable using Input Text Box:

Here we will discuss template reference variable using input text box. Template reference variable is a variable using which we can access DOM properties. In our example we are using following DOM properties of input box.

1. placeholder

2. type

3. value

Now find the code snippet.

```
<input type="text" #mobile placeholder="Enter Mobile Number" />
```

In the above input text box #mobile is a template reference variable. To fetch DOM properties, we do as follows.

mobile.placeholder: It will give placeholder of our text box if we have specified.

mobile.value: It will give value of our text box.

mobile.type: It will give type of input element. In our example type is text.

Template Reference Variable using Select Box:

Here we will discuss template reference variable with select box.

```
<select #myColor (change)="setData(myColor.value)">
</select>
```

Look at the code snippet, #myColor is a template reference variable. The selected value of select box can be accessed by myColor.value.

Template Reference Variable with NgForm:

Now we will discuss how to access NgForm directive using template reference variable. We are creating a sample form here.

```
<form (ngSubmit)="onSubmitPersonForm(myForm)" #myForm="ngForm">
  <input name="name" required [(ngModel)]="person.pname">
  <button type="submit" [disabled]="!myForm.form.valid">Submit</button>
</form>
```

ngSubmit: It enables binding angular expressions to onsubmit event of the form. Here on form submit onSubmitPersonForm component method will be called.

ngForm: It is the nestable alias of form directive

Here we are using template reference variable for ngForm as #myForm="ngForm". Now we can use myForm in place of ngForm such as to check form validity and we can also use it in our angular expressions.



Complete Program:

AppComponent Class (app.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  colors: string[] = ['Red', 'Green', 'Blue'];
  getData(mobile : number, state : string, country: string) {
    console.log("Mobile:",mobile);
    console.log("State:",state);
    console.log("Country:",country);
  }
  setData(color: string) {
    console.log("Color:", color);
  }
  person = new Person();
  onSubmitPersonForm(fm) {
    console.log('Form Validated: ' + fm.form.valid);
    console.log("Name:", this.person.name);
    console.log("Age:", this.person.age);
    console.log('Form is being submitted');
  }
}
class Person {
  name : string;
  age: number
}
```

HTML Template (app.component.html):

```
<b>Mobile Number:</b><br/>
<input type="text" #mobile placeholder="Enter Mobile Number">
<br/><br/>
<b>Enter State:</b><br/>
<input type="text" ref-state>
<br/><br/>
<b>Enter Country:</b><br/>
<input type="text" #country>
<br/><br/>
<button (click)="getData(mobile.value, state.value, country.value)">Get Data</button>
<br/><br/>
<b>Mobile Placeholder Value: </b>{{mobile.placeholder}}<br/>
<b>Mobile Input Type: </b>{{mobile.type}} <br/>
<b>Mobile TextBox Value: </b>{{mobile.value}}<br/>
<b>State: </b>{{state.value}}<br/>
<b>Country: </b>{{country.value}}<br/>
```

```

<b>Select Color: </b>
<select #myColor (change)="setData(myColor.value)">
  <option *ngFor="let color of colors">
    {{color}}
  </option>
</select>
<p [style.color]="myColor.value">Hello, It is {{myColor.value}} Color</p>

<form (ngSubmit)="onSubmitPersonForm(myForm)" #myForm="ngForm" style="border:1px dotted maroon; padding:5px">
  <b>Name: </b><br/>
  <input type="text" name="name" required [(ngModel)]="person.name">
  <br/><br/>
  <b>Age: </b><br/>
  <input type="number" name="age" required [(ngModel)]="person.age">
  <br/><br/>
  <button type="submit" [disabled]="!myForm.form.valid">Submit</button>
</form>

```

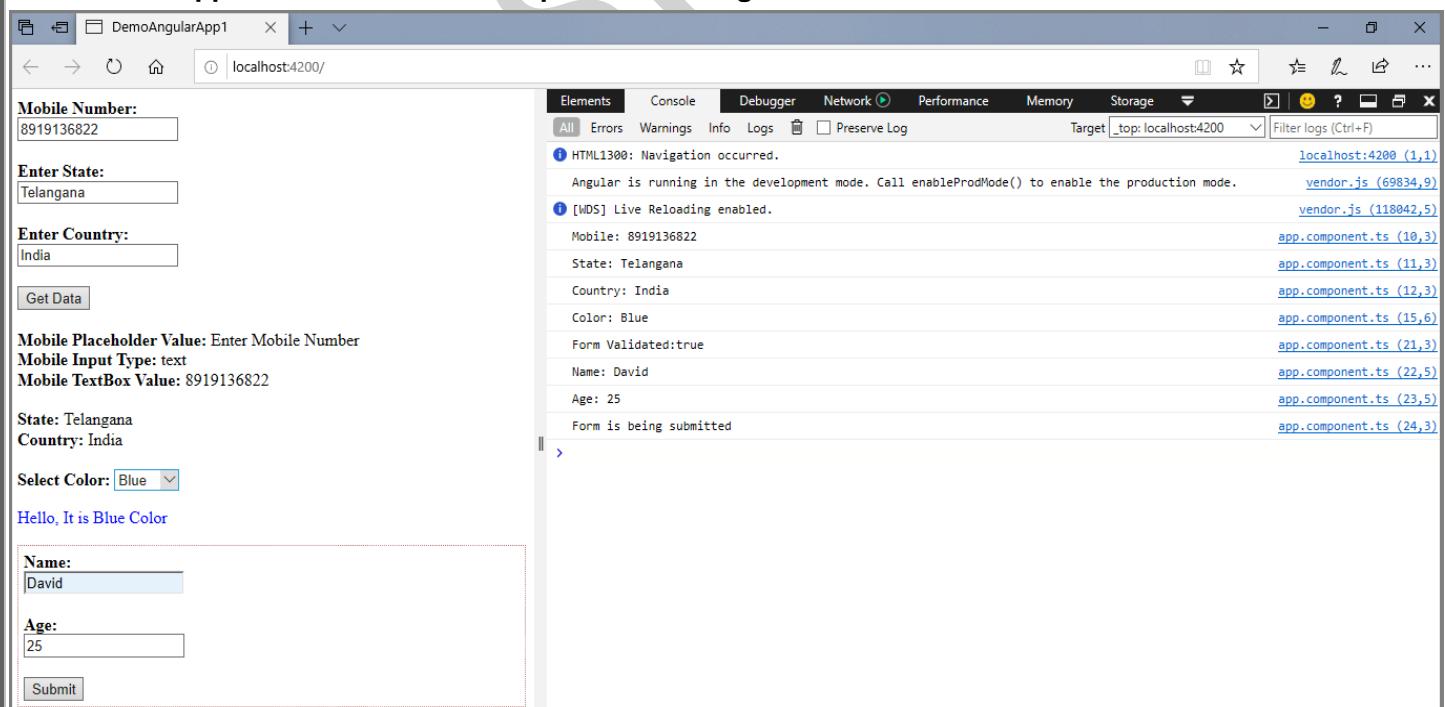
AppModule (app.module.ts):

```

import { AppComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [...],
  providers: [...],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Now run the application and see the output as following below screen:



The screenshot shows a browser window titled "DemoAngularApp1" at "localhost:4200". On the left, there is a form with fields for "Mobile Number" (8919136822), "Enter State" (Telangana), "Enter Country" (India), and a "Get Data" button. Below the form, status messages are displayed: "Mobile Placeholder Value: Enter Mobile Number", "Mobile Input Type: text", and "Mobile TextBox Value: 8919136822". Further down, it shows "State: Telangana" and "Country: India". A "Select Color:" dropdown is set to "Blue", and the message "Hello, It is Blue Color" is shown. On the right, the browser's developer tools are open, specifically the "Console" tab of the "Elements" panel. The console log shows the following entries:

- HTML1300: Navigation occurred.
- Angular is running in the development mode. Call enableProdMode() to enable the production mode.
- [WDS] Live Reloading enabled.
- Mobile: 8919136822
- State: Telangana
- Country: India
- Color: Blue
- Form Validated:true
- Name: David
- Age: 25
- Form is being submitted

ngPlural in Angular:

Adds/removes DOM sub-trees based on a numeric value. Tailored for pluralization.

Description:

Displays DOM sub-trees that match the switch expression value, or failing that, DOM sub-trees that match the switch expression's pluralization category.

To use this directive you must provide a container element that sets the **[ngPlural]** attribute to a switch expression. Inner elements with a **[ngPluralCase]** will display based on their expression:

if **[ngPluralCase]** is set to a value starting with `=`, it will only display if the value matches the switch expression exactly, otherwise, the view will be treated as a "category match", and will only display if exact value matches aren't found and the value maps to its category for the defined locale.

See Plural Rules for more details: <http://cldr.unicode.org/index/cldr-spec/plural-rules>

Example:

AppComponent Class (app.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  value: number = 1;
}
```

HTML Template (app.component.html):

```
<input [(ngModel)]="value" name="value">
<div [ngPlural]="value">
  <ng-template ngPluralCase="=0">No messages</ng-template>
  <ng-template ngPluralCase="=1">One message</ng-template>
  <ng-template ngPluralCase="other">{{ value }} messages</ng-template>
</div>
```

Output:



QueryList in Angular:

QueryList is unmodifiable list of items that Angular keeps up to date when the state of the application changes. **ViewChildren** and **ContentChildren** uses **QueryList** to store elements or directives from view DOM and content DOM respectively. We can subscribe to the changes of **QueryList** to get the current state of **QueryList**. It provides methods such as **map**, **filter**, **find**, **forEach** etc. **QueryList** can contain the elements of the type **directive**, **component**, **ElementRef**, **any** etc. Here we will discuss a complete example for **QueryList**. We will use **QueryList** with **ViewChildren** and **ContentChildren** in our example. We will discuss how to subscribe the changes of **QueryList**. We will also discuss how to use **QueryList map**, **filter**, **find**, **forEach** etc.

QueryList is used with `@ViewChildren` as given below:

```
@ViewChildren('bkWriter') writers: QueryList<WriterComponent>
```

QueryList is used with `@ContentChildren` as given below.

```
@ContentChildren(BookDirective) books: QueryList<BookDirective>
```

QueryList: length, first, last

We can get the length of **QueryList** of current state, first element and last element as following:

```
@ViewChildren('bkWriter') writers: QueryList<WriterComponent>
```

```
let len = this.writers.length;
let firstElement = this.writers.first;
let lastElement = this.writers.last;
```

QueryList.changes

To listen the changes in **QueryList**, we can subscribe to its changes. **QueryList** provides **changes** getter property that returns the instance of **Observable**.

```
@ViewChildren('bkWriter') writers: QueryList<WriterComponent>
```

```
this.writers.changes.subscribe(list => {
  list.forEach(writer => console.log(writer.writerName + ' - ' + writer.bookName));
});
```

In the **list** we will get current state of **QueryList** at any time.

QueryList.forEach()

To iterate **QueryList**, it provides **forEach()** method.

```
@ViewChildren('bkWriter') writers: QueryList<WriterComponent>
```

```
this.writers.forEach(writer => console.log(writer.writerName + ' - ' + writer.bookName));
```

QueryList.find()

To find an element with a specific value, **QueryList** provides **find()** method that returns the matching object.

```
@ViewChildren('bkWriter') writers: QueryList<WriterComponent>
```

```
let angularWriter = this.writers.find(writer => writer.bookName === 'Angular Tutorials');
```

QueryList.map()

To map the elements of a **QueryList**, it provides **map()** method that returns an array of mapped elements.

```
@ViewChildren('bkWriter') writers: QueryList<WriterComponent>
```

```
let writerNames = this.writers.map(writer => writer.writerName);
```

QueryList.filter()

To filter a **QueryList**, it provides **filter()** method that returns an array of elements matching the filter condition.

```
@ViewChildren('bkWriter') writers: QueryList<WriterComponent>
```

```
let selectedWriters = this.writers.filter(writer => writer.writerName === 'Fleming');
```

Dynamic Component Loader in Angular:

Here, we will learn how to load a component dynamically. In various scenarios, you may need to load a component dynamically.

Generally a component is loaded using component selector in component template that is identified at Angular compile time. Component can also be loaded dynamically at runtime using **ComponentFactory**, **ComponentFactoryResolver** and **ViewContainerRef**. Those components which need to be loaded dynamically should also be configured in **entryComponents** metadata of **@NgModule** decorator in module file. To load a dynamic component in a template we need an insert location and to get it we need **ViewContainerRef** of a decorator or a component.

ComponentFactory and ComponentFactoryResolver:

ComponentFactory is used to create instance of components. **ComponentFactoryResolver** resolves a **ComponentFactory** for a specific component. It is used as follows.

```
let componentFactory = this.componentFactoryResolver.resolveComponentFactory(component);
```

ViewContainerRef:

ViewContainerRef represents a container where we can attach one or more views. Some important methods of **ViewContainerRef** are **createEmbeddedView()** and **createComponent()**. **createEmbeddedView()** is used to attach embedded views based on **TemplateRef**. **createComponent()** instantiates a single component and inserts its host view into the view container at a specified index. In our dynamic component loader example, we will load component using **createComponent()** of **ViewContainerRef**.

```
let componentRef = viewContainerRef.createComponent(componentFactory);
```

We get a **ComponentRef** of newly created component as a return of the above method.

clear() method of **ViewContainerRef** destroys existing view in the container.

Dynamic Component Loader Example:

Now, we will create 2 components as listed below with below CLI command, which we will load dynamically on change of dropdown.

```
> ng g c studentinfo
> ng g c parentinfo
```

StudentInfoComponent Class (student-info.component.ts):

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-student-info',
  templateUrl: './student-info.component.html',
  styleUrls: ['./student-info.component.css']
})
export class StudentInfoComponent implements OnInit {
  message: string;
  ngOnInit() {
    alert(this.message);
  }
}
```

HTML Template (student-info.component.html):

```
<h1>
  Welcome to Student Info !!!!!
</h1>
```

ParentInfoComponent Class (parent-info.component.ts):

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-parent-info',
  templateUrl: './parent-info.component.html',
  styleUrls: ['./parent-info.component.css']
})
export class ParentInfoComponent implements OnInit {
  message: string;
  ngOnInit() {
    alert(this.message);
  }
}
```

HTML Template (parent-info.component.html):

```
<h1>
  Welcome to Parent Info !!!!!
</h1>
```

To load **StudentInfoComponent** and **ParentInfoComponent** dynamically we need a container. If we want to load **StudentInfoComponent** and **ParentInfoComponent** inside **AppComponent**, we require a **container element** in the **AppComponent**.

The template for **AppComponent.html** is as below in which a dropdown is available:

AppComponent Class (app.component.ts):

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title: string = 'Welcome to Dynamic Component Loader Example';
  data = [
    {
      "Id": 1,
      "Name": "Student Info"
    },
    {
      "Id": 2,
      "Name": "Parent Info"
    }
  ];
  selectName(id: number) {

  }
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

HTML Template (app.component.html):

```
<div align="center">
<h2>{{title}}</h2>
<b>Select Component to Load Dynamically: </b>
<select (change)="selectName($event.target.value)">
  <option value="0">--Select--</option>
  <option *ngFor="let obj of data" [value]="obj.Id">
    {{obj.Name}}
  </option>
</select>

<ng-template #loadComponent>
</ng-template>
</div>
```

As you can see, we have an entry point template or a container template in which we will load *StudentInfoComponent* and *ParentInfoComponent* dynamically.

In AppComponent, we need to import the following:

- **ViewChild**, **ViewContainerRef**, and **ComponentFactoryResolver** from `@angular/core`.
- **ComponentRef** and **ComponentFactory** from `@angular/core`.
- **StudentInfoComponent** from `student-info.component`.
- **ParentInfoComponent** from `parent-info.component`.

After importing the required elements, `app.component.ts` will look like as the following:

```
import { Component,
         ViewChild,
         ViewContainerRef,
         ComponentFactoryResolver,
         ComponentRef,
         ComponentFactory
       } from '@angular/core';
import { StudentInfoComponent } from './student-info.component';
import { ParentInfoComponent } from './parent-info.component';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title: string = 'Welcome to Dynamic Component Loader Example';
  data = [
    {
      "Id": 1,
      "Name": "Student Info"
    },
    {
      "Id": 2,
      "Name": "Parent Info"
    }
  ];
  selectName(id: number) {
  }
}
```

Inside the Component class, we can access template using **ViewChild**. The template is a container in which we load the component dynamically. Therefore, we have to access the template with the **ViewContainerRef**.

ViewContainerRef represents a container where one or more views can be attached. This can contain two types of views.

Embedded Views are created by creating an instance of **TemplateRef** using the **createEmbeddedView()** method.

Host Views are created by creating an instance of a component using the **createComponent()** method.

We will use Host Views is used to dynamically load **StudentInfoComponent** and **ParentInfoComponent**.

Let us create a variable called **entry** which will refer to the template element. In addition, we have injected **ComponentFactoryResolver** services to the component class, which will be needed to dynamically load the component.

app.component.ts:

```
export class AppComponent {
    .....
    @ViewChild('loadComponent', {static: true, read: ViewContainerRef}) entry: ViewContainerRef;
    componentRef: any;
    constructor(private resolver: ComponentFactoryResolver) { }
    .....
}
```

Keep in mind that the **entry** variable, which is a reference to a template element, has an API to create components, destroy components, etc.

To create a component, let us first create a function. Inside the function, we need to perform the following tasks:

- Clear the container.
- Create a factory for **StudentInfoComponent** and **ParentInfoComponent**.
- Create a component using the factory.
- Pass the value for @Input properties using a component reference instance method.

Putting everything together, the **createComponent** function will look like the below code:

```
createComponent(Id: number) {
    this.entry.clear();
    if (Id == 1) {
        const factory = this.resolver.resolveComponentFactory(StudentInfoComponent);
        this.componentRef = this.entry.createComponent(factory);
    }
    else if (Id == 2) {
        const factory = this.resolver.resolveComponentFactory(ParentInfoComponent);
        this.componentRef = this.entry.createComponent(factory);
    }
    this.componentRef.instance.message = "It is Called by AppComponent !!!";
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

We can call the `createComponent()` function on **change** event of dropdown and internally this method will call the `create()` method from the factory and will append the component as a sibling to our container.

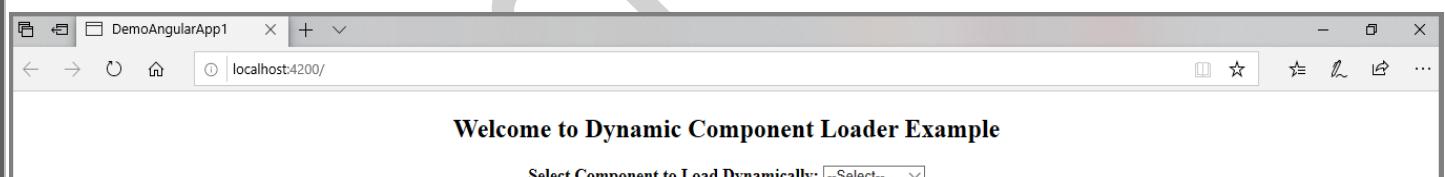
While running the application, we will get an error because we have not set the **entryComponents** in **AppModule**.

We can set this as shown below:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { StudentInfoComponent } from './student-info.component';
import { ParentInfoComponent } from './parent-info.component';

@NgModule({
  declarations: [
    AppComponent,
    StudentInfoComponent,
    ParentInfoComponent
  ],
  imports: [
    BrowserModule
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent],
  entryComponents: [StudentInfoComponent, ParentInfoComponent]
})
export class AppModule { }
```

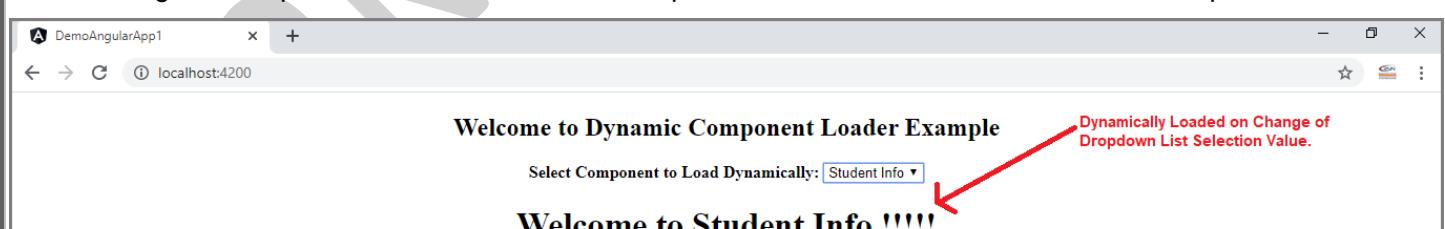
In the output, we can see that component is getting loaded dynamically on the selection of the dropdown.



Welcome to Dynamic Component Loader Example

Select Component to Load Dynamically:

As we change the dropdown selection value, the component will be reloaded with a different component.



Welcome to Dynamic Component Loader Example

Select Component to Load Dynamically:

Welcome to Student Info !!!!!

Dynamically Loaded on Change of Dropdown List Selection Value.



Welcome to Dynamic Component Loader Example

Select Component to Load Dynamically:

Welcome to Parent Info !!!!!

Dynamically Loaded on Change of Dropdown List Selection Value.

A component can be destroyed using the `destroy()` method on the `componentRef`.

app.component.ts:

```
destroyComponent() {
  this.componentRef.destroy();
}
```

We can manually destroy a dynamically loaded component by calling the function or by placing it within the `ngOnDestroy()` life cycle hook of the component so that the dynamically loaded component will also be destroyed when the host component is destroyed.

keeping everything together, AppComponent will look like as shown below:

```
import {
  Component, ViewChild,
  ViewContainerRef, ComponentFactoryResolver,
  ComponentRef, ComponentFactory
} from '@angular/core';
import { StudentInfoComponent } from './student-info.component';
import { ParentInfoComponent } from './parent-info.component';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title: string = 'Welcome to Dynamic Component Loader Example';
  componentRef: any;
  @ViewChild('loadComponent', {static: true, read: ViewContainerRef}) entry: ViewContainerRef;
  constructor(private resolver: ComponentFactoryResolver) { }
  createComponent(Id: number) {
    this.entry.clear();
    if (Id == 1) {
      const factory = this.resolver.resolveComponentFactory(StudentInfoComponent);
      this.componentRef = this.entry.createComponent(factory);
    } else if (Id == 2) {
      const factory = this.resolver.resolveComponentFactory(ParentInfoComponent);
      this.componentRef = this.entry.createComponent(factory);
    }
    this.componentRef.instance.message = "It is Called by AppComponent !!!";
  }
  destroyComponent() {
    this.componentRef.destroy();
  }
  data = [
    {
      "Id": 1, "Name": "Student Info"
    },
    {
      "Id": 2, "Name": "Parent Info"
    }
  ];
  selectName(id: number) {
    this.createComponent(id);
  }
}
```

This is all you need to do to load a component dynamically in Angular.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Port 4200 Is Already In Use. Use '--Port' To Specify A Different Port Error in Angular:

Fix port 4200 is already in use error:

Port 4200 is already in use. Use '-port' to specify a different port error occurs sometimes when we run our angular app using ng serve

That means there is another existing service already running on port 4200.

Port 4200 is already in use. Use '-port' to specify a different port error Reasons:

- An existing application(not angular) in your system using the port number 4200. This is a very rare scenario. In this case, you need to change the port number of angular application as mentioned below.

To close or terminate the angular instance always use Control+C (Ctrl+c)

`ng serve` uses default port number 4200 to run the angular application. And our application URL will be <http://localhost:4200/>. To avoid this error we need to change the port number to other port which is free.

According to RFC 793 (Internet Standard) port numbers is a 16-bit unassigned integer. The minimum value is 0 and the maximum value is 65535. And, within this range, ports 0 – 1023 are reserved for specific purposes(mostly).

To change the port number for our angular application use the below command:

- `ng serve --port 4201`
- OR
- `ng serve --port=4201`
- OR
- `ng serve --port 4201 --open [with browser launch]`
- OR
- `ng serve --open --port 4201 [with browser launch]`
- OR
- `ng serve --port=4201 --open [with browser launch]`
- OR
- `ng serve --open --port=4201 [with browser launch]`

Now, our angular application will be running on <http://localhost:4201/>

Note: If you have two or more Applications then with the help of the above approach now you are able to run both of them simultaneously in your development environment.

These all above approaches are temporarily. It means every time you need to run the application using --port option with the given port number. So for a permanent setup you can change the port by editing angular.json file as following:

angular.json:

```
"serve": {
  "options": {
    "port": 4201
  }
}
```

Now just run the command "ng serve" or "ng serve --open" as no need to specify the --port option everytime to run the application.

Alternatively we can also specify "`ng serve --port port_number`" in `package.json` like this:

```
"scripts": {
  "start": "ng serve --port 4201",
}
```

Now, in command prompt or terminal just write "`npm start`" to run the application

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Angular cli ng serve options:

Here, we will discuss some of the common options that we can use with `ng serve` command.

To see the list of all options that we can use with "ng serve" command use `--help` option as below:

➤ `ng serve --help`

Builds and serves your app, rebuilding on file changes.

usage:

`ng serve <project> [options]`

OR

`ng s <project> [options]`

arguments:

project

The name of the project to build. Can be an application or a library.

options:

There are many options available to use it with "ng serve" command of angular cli.

The following page also shows all the options that can be used with "ng serve"

<https://github.com/angular/angular-cli/wiki/serve>

The following command, builds and launches the application in your default browser.

`ng serve --open`

Many of the people saying their application is using Internet Explorer or Microsoft Edge browser by default, but they want to use Google Chrome instead. So thier question is how to change my default browser. Well that's simple and it really depends on the operating system you have. For example, on a Windows operatin system here are the steps to change your default browser.

- ❖ Click on the Windows Start Button and in the "Search programs and files" text box type: Control Panel
- ❖ Control Panel would appear in the list. Click on it.
- ❖ In the "Control Panel" window, click on "Default Programs"
- ❖ In the "Default Programs" window, click on "Set your default programs"
- ❖ In the list of programs that appear, select the "Browser" that you want to be the default browser and then click on the link that says "Set this program as default"

That's it. At this point, execute "`ng serve --open`" command and you will have your application launched in your specified default browser.

Instead of using the full option name `--open`, you can also use it's alias `-o`

The following table shows the **common options, alias, default value & their purpose**

Option	Alias	Default	Purpose
--watch		true	Run build when files change
--live-reload		true	Whether to reload the page on change
--open	-o	false	Opens the url in default browser
--port		4200	The port on which the server is listening
--host		localhost	Host to listen on.

Component Styles :host, :host-context, /deep/ Selector in Angular:

Here we will discuss angular component styles :host, :host-context, /deep/ selector example. Component styles can use few special selectors such as :host, :host-context and /deep/ that works using shadow DOM scoping. Shadow DOM makes things separate from DOM of main document. Shadow DOM provides encapsulation for DOM and CSS. Shadow DOM can also be used itself outside of the web component. Shadow DOM makes CSS management easy and maintainable in the big applications.

In Angular :host, :host-context, and /deep/ selectors are used in components that are in parent child relationship. The :host selector in Angular component plays the role to apply styles on host element. By default component style works within its component template. But by using :host selector we can apply styles to host element that resides in parent component.

The :host-context selector works same as :host selector but based on a given condition.

The /deep/ selector forces styles in its own components and in all child components.

Component Styles:

Every component can have its own HTML file as well as CSS file. Component can also use inline HTML and CSS code.

Use of some @Component metadata:

template: Write inline HTML code.

templateUrl: Configure HTML file.

styles: Write inline CSS.

styleUrls: Configure CSS file.

Find the advantage of component style over traditional CSS:

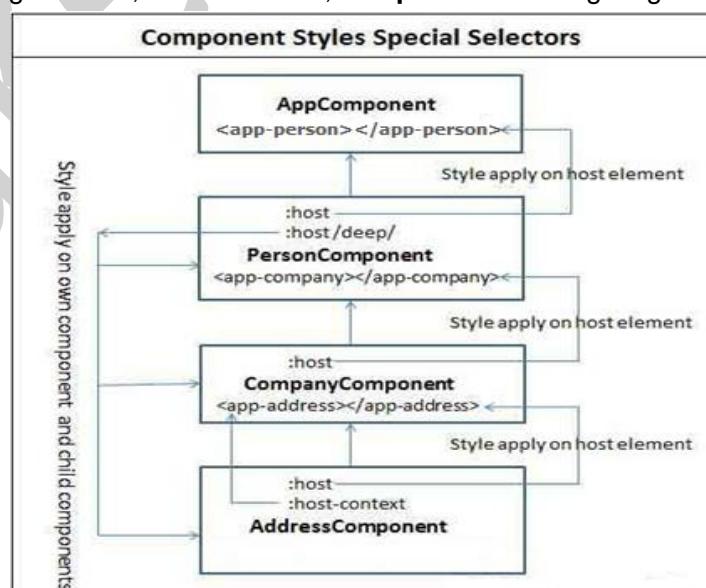
1. Class names and selectors are local to component and don't affect other part of application.
2. Changing style to other part of application does not affect the component style.
3. We can create CSS file component specific and can co-locate with them.
4. Removing and updating component CSS is easy because we need not to worry about whether else the CSS is being used in the application.

Component styles have few special selectors that has been taken from shadow DOM style scoping.

We will discuss here :host, :host-context, /deep/ selector with examples.

Diagram for :host, :host-context, /deep/ Selector:

Let us understand the working of :host, :host-context, /deep/ selector using diagram.



In the above diagram we have few components in parent-child relationship. Find the explanation of pseudo selectors.

:host: Style is written in child component but it applies on host element in parent component. Look at the diagram. The style written in `:host` selector in **PersonComponent** will be applied in `<app-person>` element of **AppComponent**. The style written in `:host` selector in **CompanyComponent** will be applied in `<app-company>` element of **PersonComponent** and so on.

:host-context: It applies style on host element based on some condition outside of a component view. Condition can be like if the given CSS class is available anywhere in parent tree, only when the style written in `:host-context` will be applied to host element in parent component.

/deep/: The style written in `/deep/` selector will be applied on its own component template and down through the child component tree into all the child component views. In the diagram **PersonComponent** is using `/deep/` selector. The style written in `/deep/` selector will be applied in **PersonComponent**, **CompanyComponent** and **AddressComponent** templates.

:host

`:host` is a pseudo-class selector that applies styles in the element that hosts the component. It means if a component has a child component using component binding then child component will use `:host` selector that will target host element in parent component. `:host` selector can be used in component with **styles** metadata as well as with **styleUrls** metadata of `@Component` decorator.

1. Example of `:host` selector with **styles**:

```
@Component({
  ...
  styles: [ ':host { position: absolute; top: 10%; }' ]
})
```

2. Example of `:host` selector with **styleUrls**. If we use CSS file as follows.

address.component.css

```
:host {
  position: absolute;
  top: 10%;
}
```

Then it is configured as follows.

```
@Component({
  ...
  styleUrls: ['./address.component.css']
})
```

Now suppose we have a parent component as given below:

person.component.ts:

```
@Component({
  template: `
    <app-company></app-company>
  `
})
export class PersonComponent {
```

Find the child component

company.component.ts:

```
@Component({
  selector: 'app-company',
  template: `
    <h3>Company</h3>
    `,
  styleUrls: ['./company.component.css']
})
export class CompanyComponent {
```

company.component.css

```
:host {
  position: absolute;
  margin: 5% 5%;
  border: 5px solid blue;
  width: 250px;
  height: 250px;
  background-color: grey;
}
```

The style written in **:host** selector will be applied to **<app-company>** element.

:host-context

:host-context selector is used in the same way as **:host** selector but **:host-context** is used when we want to apply a style on host element on some condition outside of the component view. For the example a style could be applied on host element only if a given CSS class is found anywhere in parent tree up to the document root. In our example we have following components in parent-child relationship.

AppComponent -> PersonComponent -> CompanyComponent -> AddressComponent

Suppose we have a CSS class in **AppComponent** as follows:

app.component.css:

```
.my-theme {
  background-color: blue;
}
```

Now in the last component of the child tree i.e. **AddressComponent**, we will use **:host-context** selector as follows:

address.component.css:

```
:host-context(.my-theme) h3 {
  background-color: green;
  font-style: normal;
}
```

The style given above will be applied on host element only when a CSS class named as **my-theme** will be found anywhere in parent tree up to the document root.

/deep/

/deep/ selector has alias as **>>>**. Component style normally applies only to the component's own template. Using **/deep/** selector we can force a style down through the child component tree into all child component views. **/deep/** selector forces its style to its own component, child component, nested component, view children and content children. Suppose we have components with parent child relationship as follows.

PersonComponent -> CompanyComponent -> AddressComponent

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

In PersonComponent we are using following CSS file with /deep/ selector.

person.component.css

```
:host /deep/ h3 {
  color: yellow;
  font-style: italic;
}

:host >>> p {
  color: white;
  font-style: Monospace;
  font-size: 20px;
}
```

The above style will be forced on templates of **PersonComponent**, **CompanyComponent** and **AddressComponent**.

Complete Example

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <div [ngClass]="'my-theme'">
      <app-person></app-person>
    </div>
  `,
  styleUrls: ['./app.component.css']
})
export class AppComponent {}
```

app.component.css:

```
.my-theme {
  background-color: blue;
}

.my-theme .active {
  color: white;
}
```

person.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: "app-person",
  template: `
    <h3>Person</h3>
    <p>Welcome to Person Home</p>
    <app-company></app-company>
  `,
  styleUrls: ['./person.component.css']
})
export class PersonComponent {}
```

person.component.css:

```
:host /deep/ h3 {
    color: yellow;
    font-style: italic;
}

:host >>> p {
    color: white;
    font-style: Monospace;
    font-size: 20px;
}

:host {
    position: absolute;
    top: 5%;
    border: 5px solid red;
    background-color: silver;
    width: 300px;
    height: 400px;
}
```

company.component.ts:



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

@Component({
    selector: 'app-company',
    template: `
        <h3>Company</h3>
        <p>Welcome to Company Home</p>
        <app-address></app-address>
    `,
    styleUrls: ['./company.component.css']
})
```

```
export class CompanyComponent {
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



company.component.css:

```
:host {
  position: absolute;
  margin: 5% 5%;
  border: 5px solid blue;
  width: 250px;
  height: 250px;
  background-color: grey;
}

h3 {
  font-size: 15px;
  background-color: black;
}
```

address.component.ts:

```
import { Component, ViewChild } from '@angular/core';
@Component({
  selector: 'app-address',
  template: `
    

### Address



Welcome to Address Home


  `,
  styleUrls: ['./address.component.css']
})
export class AddressComponent {}
```

address.component.css:

```
:host {
  position: absolute;
  margin: 5% 5%;
  border: 5px solid blue;
  width: 200px;
  height: 120px;
  background-color: blue;
}

:host-context(.my-theme) h3 {
  background-color: green;
  font-style: normal;
}

:host-context(.my-theme) p {
  color: red;
  font-size: 18px;
}
```

app.module.ts

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { PersonComponent } from './person.component';
import { CompanyComponent } from './company.component';
import { AddressComponent } from './address.component';

@NgModule({
  declarations: [
    AppComponent,
    PersonComponent,
    CompanyComponent,
    AddressComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Now finally run the application to see the output.

Output:





Angular with Bootstrap (Working with Bootstrap in Angular Project):

Here, we will see how to use Bootstrap to style apps built using Angular. We'll see how to integrate Angular with Bootstrap, in various ways. We'll also discuss examples of how to use Bootstrap in Angular.

Bootstrap is the world's most popular free and open-source CSS framework for building responsive, and mobile-first front-end web development sites. It was developed by **Mark Otto** and **Jacob Thornton** at **Twitter** in **August 2011** as a framework to encourage consistency across internal tools. Before Bootstrap, various libraries were used for interface development, which led to inconsistencies and a high maintenance burden.

Bootstrap provides several CSS classes for making responsive website for your mobile and optionally JavaScript-based design templates for **typography, forms, buttons, navigation** and other interface components.

Current version of Bootstrap is **v4.4.1** as of today date (At the time of writing this notes). It uses **jQuery** (and **Popper.js** sometimes) as a dependency. Bootstrap uses jQuery for JavaScript plugins (like modals, tooltips, etc). However, if you just use the CSS part of Bootstrap, you don't need jQuery.

What is Responsive Web Design?

Responsive web design is about creating web sites which automatically adjust themselves to look good on all devices, from small phones to large desktops.

Why Use Bootstrap?

Advantages of Bootstrap:

- **Easy to use:** Anybody with just basic knowledge of HTML and CSS can start using Bootstrap
- **Responsive features:** Bootstrap's responsive CSS adjusts to phones, tablets, and desktops
- **Mobile-first approach:** In Bootstrap 3, mobile-first styles are part of the core framework
- **Browser compatibility:** Bootstrap is compatible with all modern browsers (Chrome, Firefox, Internet Explorer, Edge, Safari, and Opera)

A combination of Bootstrap & Angular will make an application fast, visually appealing and modern.

These are following major ways to add bootstrap to our angular project.

This can be done in multiple ways:

- Including the Bootstrap CSS and JavaScript files in the **<head>** section of the **index.html** file of your Angular project with a **<link>** and **<script>** tags.
- Importing the Bootstrap CSS file in the global **styles.css** file of your Angular project with an **@import** keyword.
- Adding the Bootstrap CSS and JavaScript files in the **styles** and **scripts** arrays of the **angular.json** file of your project.
- Using CDN (Content Delivery Network or Content Distribution Network)

Advantage of using the Bootstrap CDN:

Many users already have downloaded Bootstrap from any available CDN when visiting another site. As a result, it will be loaded from cache when they visit your site, which leads to faster loading time. Also, most CDN's will make sure that once a user requests a file from it, it will be served from the server closest to them, which also leads to faster loading time.

Installing Bootstrap in Angular Project:

There are various ways that you can use to install Bootstrap in your project:

- Installing Bootstrap from npm using the **npm install** command,
- Downloading Bootstrap files and adding them to the **src/assets** folder of your Angular project,
- Using Bootstrap from a CDN.

Let's proceed with the first method. Go back to your command-line interface and install Bootstrap via npm as follows:

First of all switch to the project created, which is in the directory DemoAngularApp1. Change the directory in the command line - cd DemoAngularApp1.

PS D:\AngularApps> cd DemoAngularApp1

PS D:\AngularApps\DemoAngularApp1>

Install Bootstrap via npm:

```
➤ npm install --save bootstrap
OR
➤ npm install bootstrap --save
```

If you want to use bootstrap Javascript function, you need to install JQuery and popperjs with it. BootstrapJS depends on JQuery.

```
➤ npm install --save jquery
OR
➤ npm install jquery --save
```

If you need the functionality of a popover in angular application, you can add popper.js.

```
➤ npm install --save popper.js
OR
➤ npm install popper.js --save
```

This will also add the **bootstrap**, **jquery** & **popper** package to **package.json** in dependencies section as follows:

```
"dependencies": {
  "@angular/animations": "~8.2.0",
  "@angular/common": "~8.2.0",
  "@angular/compiler": "~8.2.0",
  "@angular/core": "~8.2.0",
  "@angular/forms": "~8.2.0",
  "@angular/platform-browser": "~8.2.0",
  "@angular/platform-browser-dynamic": "~8.2.0",
  "@angular/router": "~8.2.0",
  "bootstrap": "^4.4.1",
  "jquery": "^3.4.1",
  "popper.js": "^1.16.0",
  "rxjs": "~6.4.0",
  "tslib": "^1.10.0",
  "zone.js": "~0.9.1"
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



The Bootstrap assets will be installed in the **node_modules/bootstrap** folder.

```

    <ul style="list-style-type: none; padding-left: 0;">
        <li><span style="color: #ccc;">></span> node_modules
            <li><span style="color: #ccc;">></span> <span style="color: #ccc;">> bootstrap
                <li><span style="color: #ccc;">></span> <span style="color: #ccc;">> dist
                    <li><span style="color: #ccc;">></span> <span style="color: #ccc;">> css
                        <li><span style="color: #ccc;">#></span> bootstrap-grid.css
                        <li><span style="color: #ccc;">#></span> bootstrap-grid.css.map
                        <li><span style="color: #ccc;">#></span> bootstrap-grid.min.css
                        <li><span style="color: #ccc;">#></span> bootstrap-grid.min.css.map
                        <li><span style="color: #ccc;">#></span> bootstrap-reboot.css
                        <li><span style="color: #ccc;">#></span> bootstrap-reboot.css.map
                        <li><span style="color: #ccc;">#></span> bootstrap-reboot.min.css
                        <li><span style="color: #ccc;">#></span> bootstrap-reboot.min.css.map
                        <li style="background-color: black; color: white; padding: 2px 5px; border: 1px solid black; margin-bottom: 5px;">#> bootstrap.css
                        <li><span style="color: #ccc;">#></span> bootstrap.css.map
                        <li style="background-color: black; color: white; padding: 2px 5px; border: 1px solid black; margin-bottom: 5px;">#> bootstrap.min.css
                        <li><span style="color: #ccc;">#></span> bootstrap.min.css.map
                    </ul>
    
```

The jQuery assets will be installed in the **node_modules/jquery** folder.

```

    <ul style="list-style-type: none; padding-left: 0;">
        <li><span style="color: #ccc;">></span> node_modules
            <li><span style="color: #ccc;">></span> jquery
                <li><span style="color: #ccc;">></span> <span style="color: #ccc;">> dist
                    <li><span style="color: #ccc;">JS></span> core.js
                    <li style="background-color: black; color: white; padding: 2px 5px; border: 1px solid black; margin-bottom: 5px;">JS> jquery.js
                    <li style="background-color: black; color: white; padding: 2px 5px; border: 1px solid black; margin-bottom: 5px;">JS> jquery.min.js
                    <li><span style="color: #ccc;">= </span> jquery.min.map
                    <li><span style="color: #ccc;">JS></span> jquery.slim.js
                    <li><span style="color: #ccc;">JS></span> jquery.slim.min.js
                    <li><span style="color: #ccc;">= </span> jquery.slim.min.map
                </ul>
    
```

The Popper assets will be installed in the `node_modules/popper.js` folder.

```

    <ul style="list-style-type: none; padding-left: 0;">
        <li><span style="font-size: 1.2em;">> node_modules
            <li><span style="font-size: 1.2em;">> popper.js
                <li><span style="font-size: 1.2em;">> dist
                    <ul style="list-style-type: none; padding-left: 20px;">
                        <li>> esm
                        <li>> umd
                        <li style="background-color: #f0f0f0; border: 1px solid black; padding: 2px;">JS popper-utils.js
                        <li style="background-color: #f0f0f0; border: 1px solid black; padding: 2px;">JS popper-utils.js.map
                        <li style="background-color: #f0f0f0; border: 1px solid black; padding: 2px;">JS popper-utils.min.js
                        <li style="background-color: #f0f0f0; border: 1px solid black; padding: 2px;">JS popper-utils.min.js.map
                        <li style="background-color: #f0f0f0; border: 1px solid black; padding: 2px;">JS popper.js
                        <li style="background-color: #f0f0f0; border: 1px solid black; padding: 2px;">JS popper.js.map
                        <li style="background-color: #f0f0f0; border: 1px solid black; padding: 2px;">JS popper.min.js
                        <li style="background-color: #f0f0f0; border: 1px solid black; padding: 2px;">JS popper.min.js.map
                    </ul>
                </li>
            </li>
        </li>
    </ul>

```

That's it & Installation is Completed.

Adding Bootstrap to Angular Using angular.json:

Open the angular.json file of your project and include.

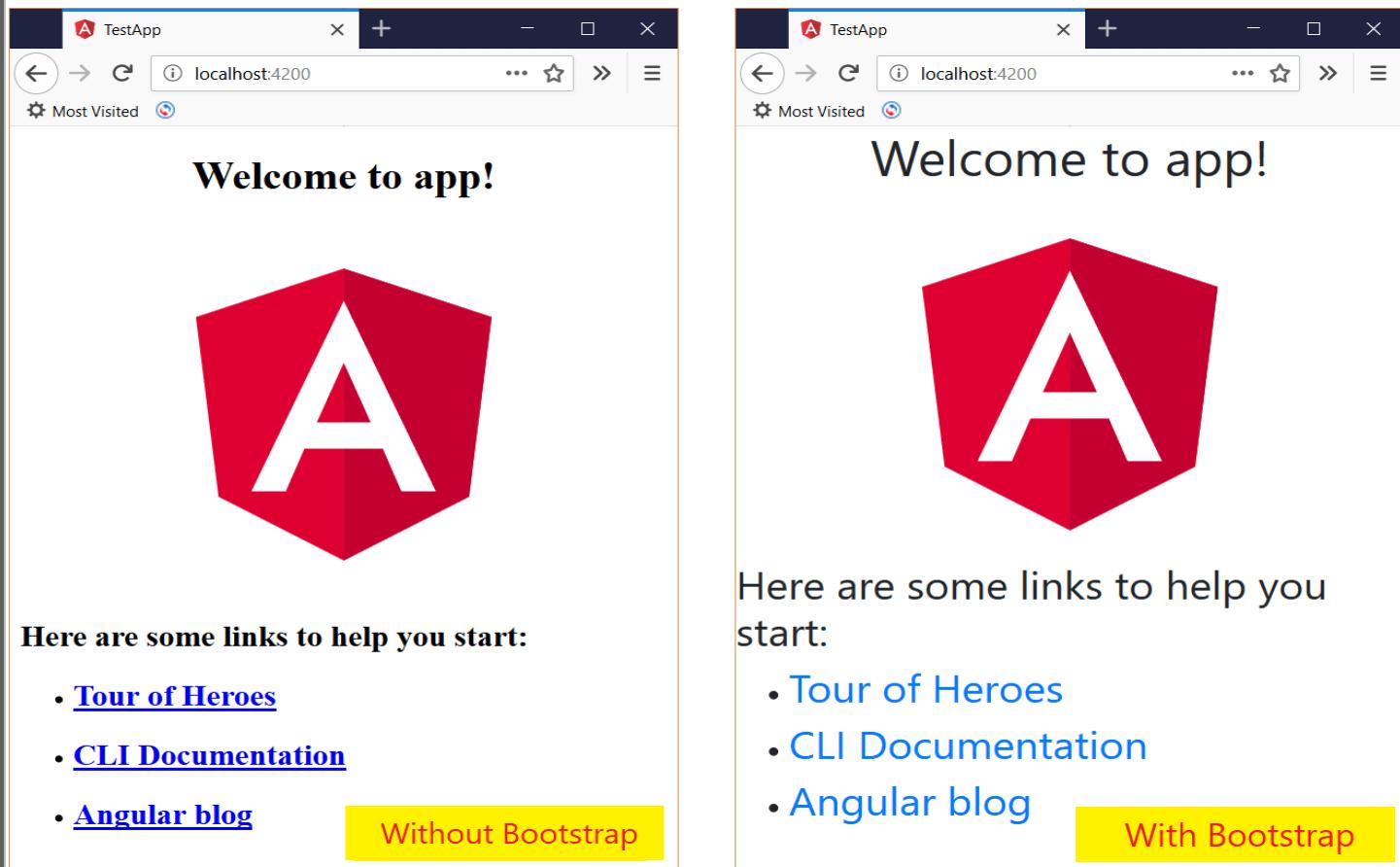
Reference the path in **angular.json** file. Make sure you reference it under **build node** as follows:

```

"projects": {
  "DemoAngularApp1": {
    ...
    "architect": {
      "build": {
        "builder": "@angular-devkit/build-angular:browser",
        "options": {
          ...
          "styles": [
            "src/styles.css",
            "node_modules/bootstrap/dist/css/bootstrap.min.css"
          ],
          ...
          "scripts": [
            "node_modules/jquery/dist/jquery.min.js",
            "node_modules/popper.js/dist/popper.js",
            "node_modules/bootstrap/dist/js/bootstrap.min.js"
          ]
        },
        ...
      }
    }
  }
}

```

That's it. Run the app and you should see bootstrap styles applied to the angular app. See below image.

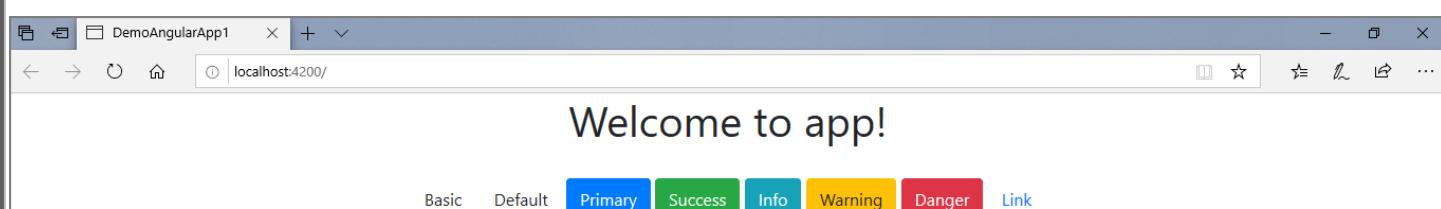


The image shows two side-by-side browser windows. Both windows have a title bar 'TestApp' and a URL 'localhost:4200'. The left window is labeled 'Without Bootstrap' and displays a large red hexagon with a white letter 'A' in the center. Below it, there is a heading 'Welcome to app!' and some text: 'Here are some links to help you start:' followed by a list of three links: 'Tour of Heroes', 'CLI Documentation', and 'Angular blog'. The right window is labeled 'With Bootstrap' and also displays a large red hexagon with a white letter 'A'. Below it, there is a heading 'Welcome to app!' and some text: 'Here are some links to help you start:' followed by a list of three links: 'Tour of Heroes', 'CLI Documentation', and 'Angular blog'.

Let's use bootstrap's button styles to the app to verify things. Add the following HTML in the `app.component.html` file to create a different style of buttons.

```
<div style="text-align: center;">
<h1>
  Welcome to {{ title }}!
</h1>
<br/>
<button type="button" class="btn">Basic</button>
<button type="button" class="btn btn-default">Default</button>
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-link">Link</button>
</div>
```

Output:



The image shows a browser window titled 'DemoAngularApp1' with the URL 'localhost:4200/'. The page content is identical to the previous 'Without Bootstrap' screenshot, featuring a large red hexagon with a white letter 'A'. Below it, there is a heading 'Welcome to app!' and some text: 'Here are some links to help you start:' followed by a list of three links: 'Tour of Heroes', 'CLI Documentation', and 'Angular blog'. At the bottom of the page, there is a row of seven buttons with different colors and labels: 'Basic' (light blue), 'Default' (light green), 'Primary' (blue), 'Success' (green), 'Info' (teal), 'Warning' (yellow), and 'Danger' (red).

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

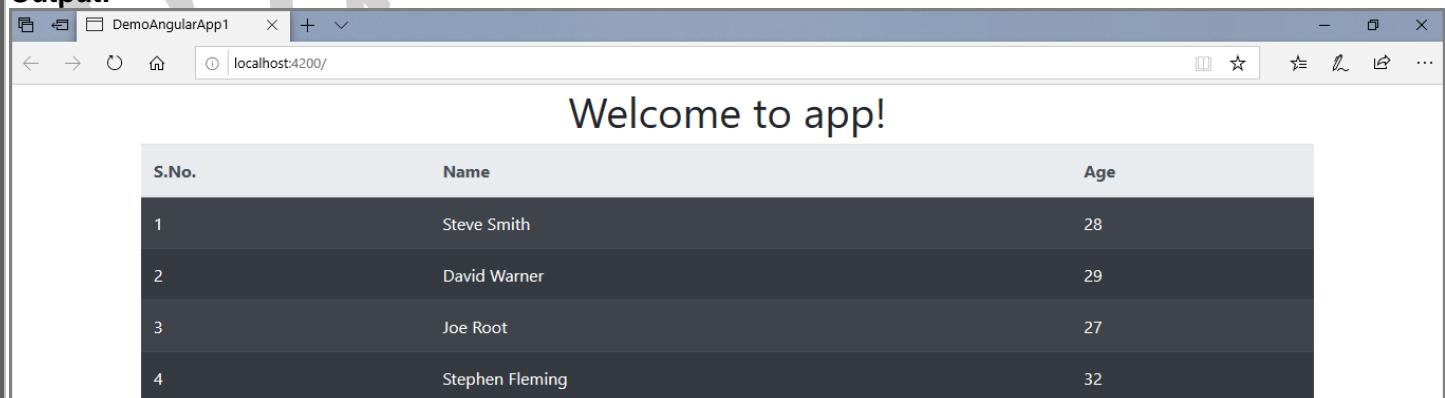
Let's add some bootstrap 4 stuff to the app to verify things. One of the new features of bootstrap 4 is support for dark tables. Add the following HTML in the `app.component.html` file to create a dark style table.

```

<div style="text-align: center;">
<h1>
    Welcome to {{ title }}!
</h1>
</div>
<div class="container">
    <table class="table table-dark table-hover table-striped">
        <thead class="thead-light">
            <tr>
                <th>S.No.</th>
                <th>Name</th>
                <th>Age</th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td>1</td>
                <td>Steve Smith</td>
                <td>28</td>
            </tr>
            <tr>
                <td>2</td>
                <td>David Warner</td>
                <td>29</td>
            </tr>
            <tr>
                <td>3</td>
                <td>Joe Root</td>
                <td>27</td>
            </tr>
            <tr>
                <td>4</td>
                <td>Stephen Fleming</td>
                <td>32</td>
            </tr>
        </tbody>
    </table>
</div>

```

Output:



The screenshot shows a browser window with the URL `localhost:4200/`. The page displays a heading "Welcome to app!" followed by a dark-themed Bootstrap table with four rows of data.

S.No.	Name	Age
1	Steve Smith	28
2	David Warner	29
3	Joe Root	27
4	Stephen Fleming	32

Adding Bootstrap to Angular Using index.html:

You can also include Bootstrap files from **assets/bootstrap** using the **index.html** file.

Open the **src/index.html** file and add the following:

- A **<link>** tag for adding the **bootstrap.css** or **bootstrap.min.css** file in the **<head>** section,
- A **<script>** tag for adding the **jquery.js** or **jquery.min.js** file before the closing **</body>** tag,
- A **<script>** tag for adding the **bootstrap.js** or **bootstrap.min.js** file before the **</body>** tag.

index.html:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>DemoAngularApp1</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="stylesheet" href="assets/bootstrap/dist/css/bootstrap.css"/>
</head>
<body>
  <app-root></app-root>
  <script src="assets/jquery/dist/jquery.js"></script>
  <script src="assets/bootstrap/dist/js/bootstrap.js"></script>
</body>
</html>
```

Adding Bootstrap to Angular Using styles.css:

We can also use the **styles.css** file to add the CSS file of Bootstrap to our project.

Open the **src/styles.css** file of your Angular project and import the **bootstrap.css** file as follows:

```
@import "~bootstrap/dist/css/bootstrap.css";
```

OR

```
@import url("~bootstrap/dist/css/bootstrap.css");
```

This replaces the previous method(s) so you don't need to add the file to the **styles** array of the **angular.json** file or to the **index.html** file.

Note: The JS file(s) can be added using the **scripts** array or the **<script>** tag in **angular.json** file as the previous methods.

Adding Bootstrap to Angular Using CDN:

To include Bootstrap in our project we need to add two files:

- Bootstrap CCS file
- Bootstrap JavaScript file

The JavaScript parts of Bootstrap are depending on jQuery. So we need the jQuery JavaScript library file too.

All those files can be directly added from a CDN (Content Delivery Network) to our project. The CDN links for Bootstrap can be found at <http://getbootstrap.com/getting-started/> and the link to jQuery can be found at <https://code.jquery.com/>.

Open file src/index.html and insert

- the `<link>` element at the end of the head section to include the Bootstrap CSS file
- a `<script>` element to include jQuery at the bottom of the body section
- a `<script>` element to include the Bootstrap JavaScript file at the bottom of the body section

Now our index.html file should look like the following:

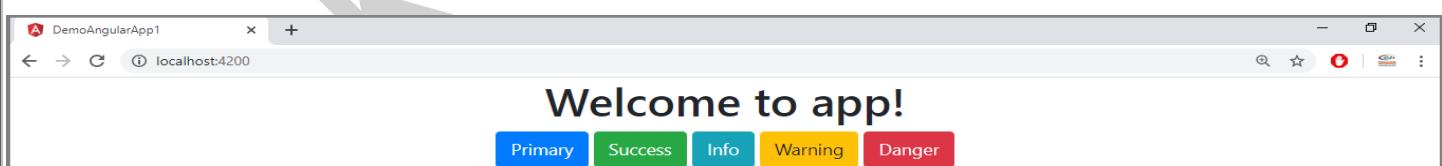
```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>DemoAngularApp1</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css" />
</head>
<body>
  <app-root></app-root>
  <script src="https://code.jquery.com/jquery-3.4.1.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"></script>
  <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js"></script>
</body>
</html>
```

Now we're ready to make use of Bootstrap in one of our component templates. Let's try it out by opening file `src/app/app.component.html` and insert the following HTML template code:

`app.component.html`:

```
<div style="text-align: center;">
  <h1>
    Welcome to {{ title }}!
  </h1>
  <button type="button" class="btn btn-primary">Primary</button>
  <button type="button" class="btn btn-success">Success</button>
  <button type="button" class="btn btn-info">Info</button>
  <button type="button" class="btn btn-warning">Warning</button>
  <button type="button" class="btn btn-danger">Danger</button>
</div>
```

The result in the browser now looks like the following:

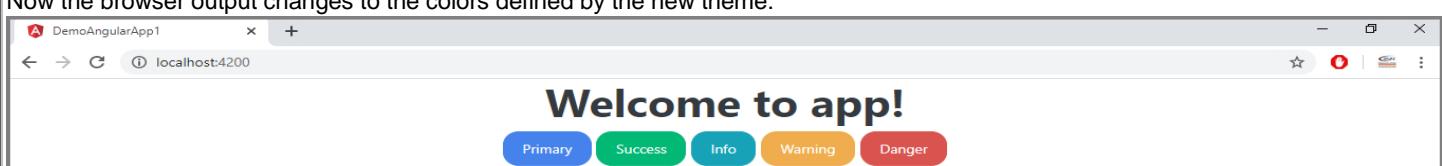


You can also change to another Bootstrap theme, e.g. from Bootswatch (<https://bootswatch.com/>). On the Website just select a theme and click on the "Download" button. The corresponding bootstrap.min.css file opens in another Browser window, so that you can copy the URL.

Go back to `index.html` and replace the string which is assigned to the href attribute of the `<link>` element with this new URL:

```
<link rel="stylesheet" href="https://bootswatch.com/4/litera/bootstrap.min.css" />
```

Now the browser output changes to the colors defined by the new theme:



Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Using ng-bootstrap:

In general Bootstrap depends on jQuery and Popper.js libraries, and if you don't include them in your project, any Bootstrap components that rely on JavaScript will not work.

Why not include those libs? For Angular it's better to avoid using libraries that make direct manipulation of the DOM (like jQuery) and let Angular handle that.

Now what if you need the complete features of Bootstrap 4 without the JS libraries?

A better way is to use component libraries created for the sake of making Bootstrap work seamlessly with Angular such as **ng-bootstrap** or **ngx-bootstrap**

ng-bootstrap contains a set of native Angular directives based on Bootstrap's markup and CSS. As a result no dependency on jQuery or Bootstrap's JavaScript is required.

Official Website to use ng-bootstrap in Angular: <https://ng-bootstrap.github.io/#/home>

Angular widgets built from the ground up using only Bootstrap 4 CSS with APIs designed for the Angular ecosystem.

No dependencies on 3rd party JavaScript.

Here is a list of minimal required versions of Angular and Bootstrap CSS for ng-bootstrap:

ng-bootstrap	Angular	Bootstrap CSS
1.x.x	5.0.2	4.0.0
2.x.x	6.0.0	4.0.0
3.x.x	6.1.0	4.0.0
4.x.x	7.0.0	4.0.0
5.x.x	8.0.0	4.3.1

Note:

Should I add **bootstrap.js** or **bootstrap.min.js** to my project?

No, the goal of **ng-bootstrap** is to completely replace JavaScript implementation for components. Nor should you include other dependencies like **jQuery** or **popper.js**. It is not necessary and might interfere with ng-bootstrap code.

Supported Browsers:

We strive to support the same browsers and versions as supported by both Bootstrap 4 and Angular, whichever is more restrictive. Check browser support notes for **Angular** (<https://angular.io/guide/browser-support>) and **Bootstrap** (<https://getbootstrap.com/docs/4.0/getting-started/browser-support/>).

ng-bootstrap is based on Bootstrap 4 and can be added to your Angular project in the following way:

ng-bootstrap is available as a NPM package, so the installation can be done by using the following command in the project directory:

➤ `npm install --save @ng-bootstrap/ng-bootstrap`

Furthermore ng-bootstrap required Bootstrap 4 to be added to our project. Install it via:

➤ `npm install --save bootstrap`

Once installed you need to import ng-bootstrap's main module **NgbModule** from the package `@ng-bootstrap/ng-bootstrap`.

Add the following import statement to **app.module.ts**:

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

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Next, we need to add this module to the imports array of the `@NgModule` decorator.

```
import { NgbModule } from '@ng-bootstrap/ng-bootstrap';
@NgModule({
  declarations: [/*...*/],
  imports: [/*...*/, NgbModule],
  /*...*/
})
export class AppModule { }
```

Alternatively you could only import modules with components you need, ex. `accordion` and `alert`. The resulting bundle will be smaller in this case.

```
import { NgbAccordionModule, NgbAlertModule } from '@ng-bootstrap/ng-bootstrap';
@NgModule({
  declarations: [/*...*/],
  imports: [/*...*/, NgbAccordionModule, NgbAlertModule],
  /*...*/
})
export class AppModule { }
```

Please note that `ng-bootstrap` requires the Bootstrap 4 CSS file to be present.

You can add it in the `styles` array of the `angular.json` file like that:

```
"styles": [
  "src/styles.css",
  "node_modules/bootstrap/dist/css/bootstrap.min.css"
]
```

Now you can use Bootstrap 4 in your Angular application.

ng-bootstrap Components;

Having imported `NgbModule` in your Angular application you can now make use of the `ng-bootstrap` components in your templates. The following components are available:

- Accordion
- Alert
- Buttons
- Carousel
- Collapse
- Datepicker
- Dropdown
- Modal
- Pagination
- Popover
- Progressbar
- Rating
- Tabs
- Timepicker
- Tooltip
- Typeahead

Let's try it out and use some of these components in app.component.html:

1. Accordion:

NgbAccordion:

Accordion is a collection of collapsible panels (bootstrap cards).

It can ensure only one panel is opened at a time and allows to customize panel headers.

Selector: `ngb-accordion`

Exported as `ngbAccordion`

Examples:

```
<ngb-accordion #acc="ngbAccordion" activeIds="ngb-panel-0">
  <ngb-panel title="User's Personal Information">
    <ng-template ngbPanelContent>
      <p style="background-color: yellow;">
        This is User's Personal Information
        <br /><br /><br />
      </p>
    </ng-template>
  </ngb-panel>
  <ngb-panel>
    <ng-template ngbPanelTitle>
      <span>User's Address Information</b> </span>
    </ng-template>
    <ng-template ngbPanelContent>
      <p style="background-color: orange;">
        This is User's Address Information
        <br /><br /><br />
      </p>
    </ng-template>
  </ngb-panel>
  <ngb-panel title="User's Contact Information" [disabled]="true">
    <ng-template ngbPanelContent>
      <p style="background-color: aqua;">
        This is User's Contact Information
        <br /><br /><br />
      </p>
    </ng-template>
  </ngb-panel>
</ngb-accordion>
```

Output:

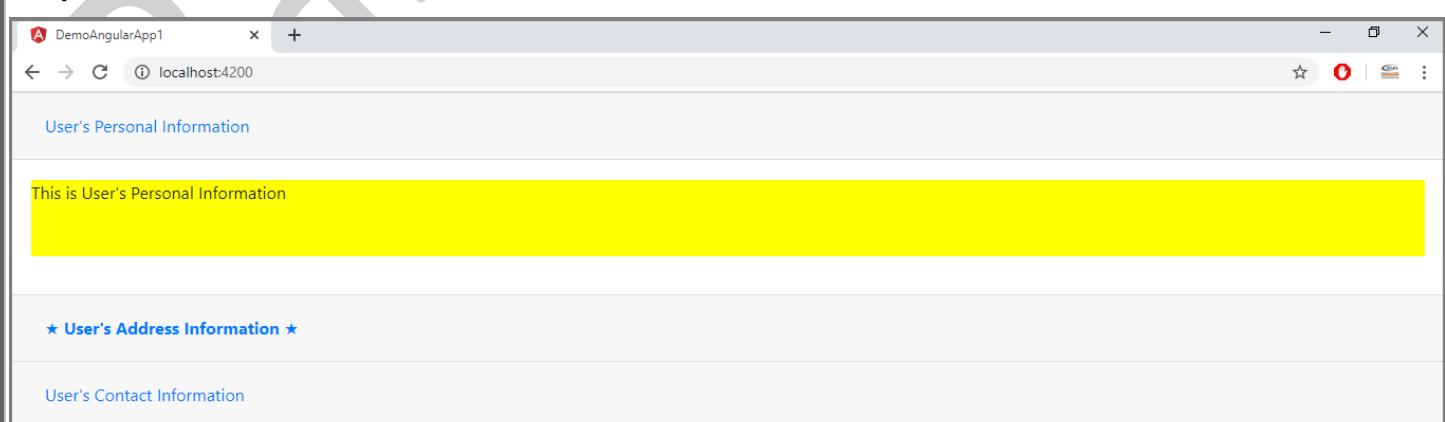


One open panel at a time:

app.component.html:

```
<ngb-accordion [closeOthers]="true" activeIds="panel-1">
  <ngb-panel id="panel-1" title="User's Personal Information">
    <ng-template ngbPanelContent>
      <p style="background-color: yellow;">
        This is User's Personal Information
        <br /><br /><br />
      </p>
    </ng-template>
  </ngb-panel>
  <ngb-panel id="panel-2">
    <ng-template ngbPanelTitle>
      <span>User's Address Information</b> </span>
    </ng-template>
    <ng-template ngbPanelContent>
      <p style="background-color: orange;">
        This is User's Address Information
        <br /><br /><br />
      </p>
    </ng-template>
  </ngb-panel>
  <ngb-panel id="panel-3" title="User's Contact Information">
    <ng-template ngbPanelContent>
      <p style="background-color: aqua;">
        This is User's Contact Information
        <br /><br /><br />
      </p>
    </ng-template>
  </ngb-panel>
</ngb-accordion>
```

Output:



The screenshot shows a browser window titled "DemoAngularApp1" with the URL "localhost:4200". The page displays three panels from an accordion. The first panel, "User's Personal Information", is open and its content ("This is User's Personal Information") is highlighted with a yellow background. The second panel, "User's Address Information", is collapsed and indicated by a blue link. The third panel, "User's Contact Information", is also collapsed.

Toggle panels:

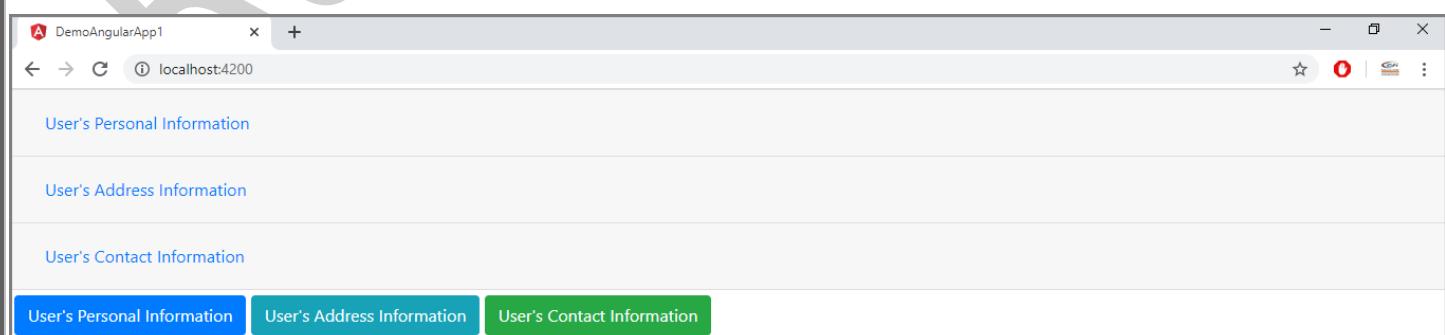
app.component.html:

```
<ngb-accordion #acc="ngbAccordion">
  <ngb-panel id="panel-1" title="User's Personal Information">
    <ng-template ngbPanelContent>
      <p style="background-color: yellow;">
        This is User's Personal Information
        <br /><br /><br />
      </p>
    </ng-template>
  </ngb-panel>
  <ngb-panel id="panel-2" title="User's Address Information">
    <ng-template ngbPanelContent>
      <p style="background-color: orange;">
        This is User's Address Information
        <br /><br /><br />
      </p>
    </ng-template>
  </ngb-panel>
  <ngb-panel id="panel-3" title="User's Contact Information">
    <ng-template ngbPanelContent>
      <p style="background-color: aqua;">
        This is User's Contact Information
        <br /><br /><br />
      </p>
    </ng-template>
  </ngb-panel>
</ngb-accordion>


<button class="btn btn-primary" (click)="acc.toggle('panel-1')">User's Personal Information</button>
  <button class="btn btn-info" (click)="acc.toggle('panel-2')">User's Address Information</button>
  <button class="btn btn-success" (click)="acc.toggle('panel-3')">User's Contact Information</button>


```

Output:



Custom header:

app.component.html:

```
<ngb-accordion #acc="ngbAccordion" activeIds="panel-1">
  <ngb-panel id="panel-1">
    <ng-template ngbPanelHeader let-opened="opened">
      <div class="d-flex align-items-center justify-content-between">
        <h5 class="m-0">User's Personal Information - {{opened?'Opened':'Collapsed'}}</h5>
        <button ngbPanelToggle class="btn btn-link p-0">Toggle Personal Info</button>
      </div>
    </ng-template>
    <ng-template ngbPanelContent>
      <p style="background-color: yellow;">This is User's Personal Information<br/><br/><br/></p>
    </ng-template>
  </ngb-panel>
  <ngb-panel>
    <ng-template ngbPanelHeader>
      <div class="d-flex align-items-center justify-content-between">
        <h5 class="m-0">User's Address Information</h5>
        <div>
          <button ngbPanelToggle class="btn btn-sm btn-outline-primary ml-2">
            Toggle Address Information
          </button>
          <button type="button" class="btn btn-sm btn-outline-secondary ml-2"
                 (click)="disabled = !disabled">
            {{ disabled ? 'En' : 'Dis' }}able Contact Info
          </button>
          <button type="button" class="btn btn-sm btn-outline-danger ml-2"
                 (click)="CollapseExpandAll()">
            {{buttonText}}
          </button>
        </div>
      </div>
    </ng-template>
    <ng-template ngbPanelContent>
      <p style="background-color: orange;">This is User's Address Information<br/><br/><br/></p>
    </ng-template>
  </ngb-panel>
  <ngb-panel [disabled]="disabled">
    <ng-template ngbPanelHeader>
      <div class="d-flex align-items-center justify-content-between">
        <button ngbPanelToggle class="btn btn-link container-fluid text-left pl-0">
          User's Contact Information
        </button>
        <p *ngIf="disabled" class="text-muted m-0 small">[I'm &nbsp;disabled]</p>
      </div>
    </ng-template>
    <ng-template ngbPanelContent>
      <p style="background-color: aqua;">
        This is User's Contact Information
        <br /><br /><br />
      </p>
    </ng-template>
  </ngb-panel>
</ngb-accordion>
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



app.component.ts:

```
import { Component, ViewChild, } from '@angular/core';
import { NgbAccordion } from '@ng-bootstrap/ng-bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  disabled: boolean = false;
  buttonText: string = "Collpase All";
  flag: boolean = false;
  @ViewChild("acc", {static: true}) accordion: NgbAccordion;
  CollpaseExpandAll(){
    if(this.flag){
      this.buttonText = "Collapse All";
      this.flag = false;
      this.accordion.expandAll();
    }
    else{
      this.buttonText = "Expand All";
      this.flag = true;
      this.accordion.collapseAll();
    }
  }
}
```

Output:

Prevent panel toggle:

app.component.html:

```
<ngb-accordion (panelChange)="beforeChange($event)">
  <ngb-panel id="preventchange-1" title="Simple">
    <ng-template ngbPanelContent>
      <p style="background-color: yellow;">This is Simple Information<br /><br /><br /></p>
    </ng-template>
  </ngb-panel>
  <ngb-panel id="preventchange-2" title="I can't be toggled...">
    <ng-template ngbPanelContent>
      <p style="background-color: orange;">This is Another Information<br /><br /><br /></p>
    </ng-template>
  </ngb-panel>
  <ngb-panel id="preventchange-3" title="I can be opened, but not closed...">
    <ng-template ngbPanelContent>
      <p style="background-color: aqua;">
        This is Containing More Information<br /><br /><br /></p>
    </ng-template>
  </ngb-panel>
</ngb-accordion>
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

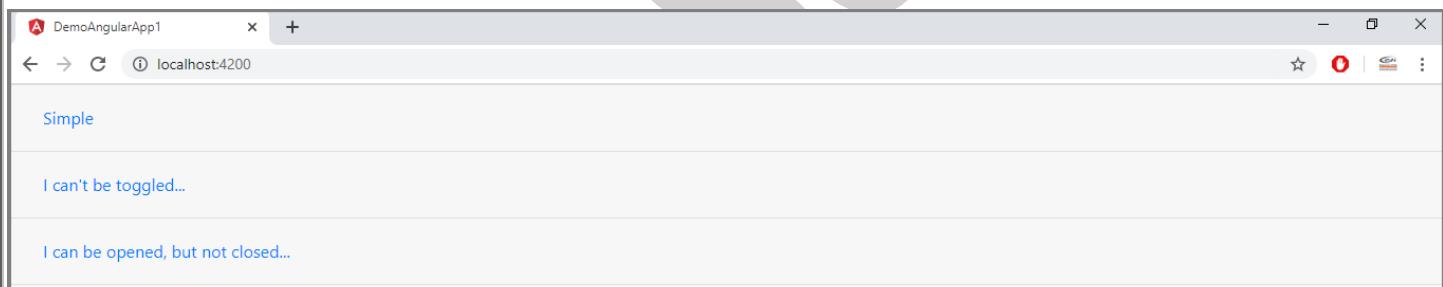
app.component.ts:

```
import { Component } from '@angular/core';
import { NgbPanelChangeEvent } from '@ng-bootstrap/ng-bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  public beforeChange($event: NgbPanelChangeEvent) {

    if ($event.panelId === 'preventchange-2') {
      $event.preventDefault();
    }

    if ($event.panelId === 'preventchange-3' && $event.nextState === false) {
      $event.preventDefault();
    }
  }
}
```

Output:



Global configuration of accordions:

This accordion uses customized default values.

app.component.html:

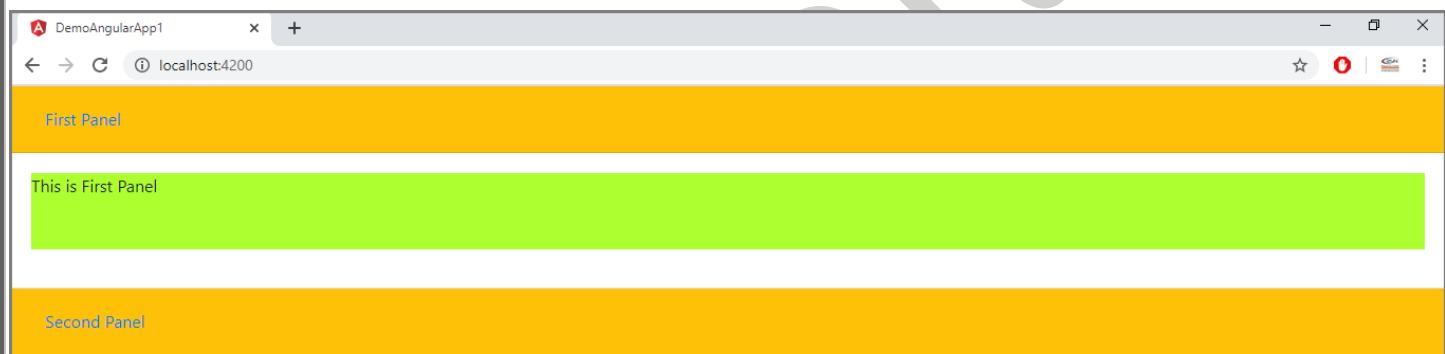
```
<ngb-accordion #acc="ngbAccordion" activeIds="panel-one">
  <ngb-panel id="panel-one" title="First Panel">
    <ng-template ngbPanelContent>
      <p style="background-color:greenyellow">This is First Panel<br/><br/><br/></p>
    </ng-template>
  </ngb-panel>
  <ngb-panel title="Second Panel">
    <ng-template ngbPanelContent>
      <p style="background-color:aqua;">This is Second Panel<br/><br/><br/></p>
    </ng-template>
  </ngb-panel>
</ngb-accordion>
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.component.ts:

```
import { Component, } from '@angular/core';
import { NgbAccordionConfig } from '@ng-bootstrap/ng-bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [NgbAccordionConfig] // add the NgbAccordionConfig to the component providers
})
export class AppComponent {
  constructor(config: NgbAccordionConfig) {
    // customize default values of accordions used by this component tree
    config.closeOthers = true;
    config.type = 'warning';
  }
}
```

Output:



2. Alert:

NgbAlert:

Alert is a component to provide contextual feedback messages for user.

It supports several alert types and can be dismissed.

Selector: ngb-alert

Examples:

Basic Alert:

app.component.html:

```
<p>
  <ngb-alert [dismissible]="false">
    <strong>Warning!</strong> Better check yourself, you're not looking too good.
  </ngb-alert>
</p>
```

Output:



Closable Alert:

app.component.ts:

```
import { Component } from '@angular/core';
interface Alert {
  type: string;
  message: string;
}
const ALERTS: Alert[] = [
  {
    type: 'success',
    message: 'This is an success alert',
  },
  {
    type: 'info',
    message: 'This is an info alert',
  },
  {
    type: 'warning',
    message: 'This is a warning alert',
  },
  {
    type: 'danger',
    message: 'This is a danger alert',
  },
  {
    type: 'primary',
    message: 'This is a primary alert',
  },
  {
    type: 'secondary',
    message: 'This is a secondary alert',
  },
  {
    type: 'light',
    message: 'This is a light alert',
  },
  {
    type: 'dark',
    message: 'This is a dark alert',
  }
];
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  alerts: Alert[];
  constructor() {
    this.reset();
  }
  close(alert: Alert) {
    this.alerts.splice(this.alerts.indexOf(alert), 1);
  }
  reset() {
    this.alerts = Array.from(ALERTS);
  }
}
```

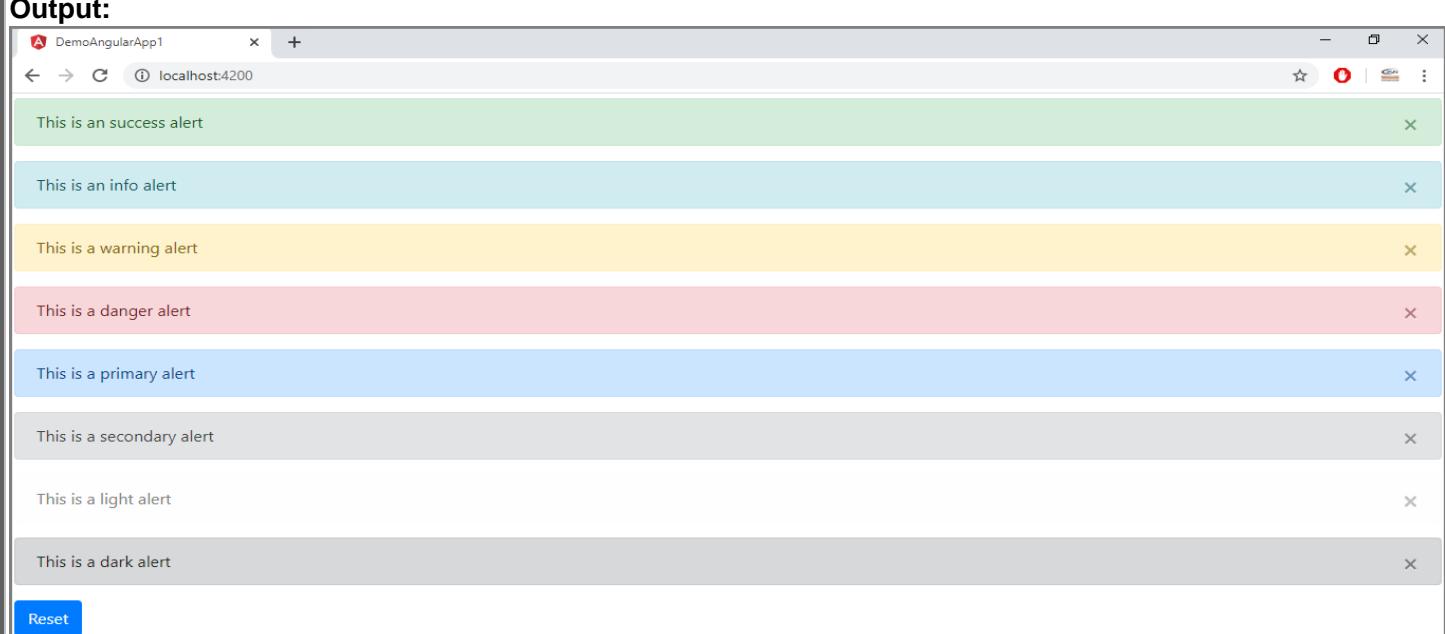
app.component.html:

```

<div style="padding: 5px;">
  <p *ngFor="let alert of alerts">
    <ngb-alert [type]="alert.type" (close)="close(alert)">{{ alert.message }}</ngb-alert>
  </p>
  <p>
    <button type="button" class="btn btn-primary" (click)="reset()">Reset</button>
  </p>
</div>

```

Output:



DemoAngularApp1

This is an success alert

This is an info alert

This is a warning alert

This is a danger alert

This is a primary alert

This is a secondary alert

This is a light alert

This is a dark alert

Reset

Closable Single Alert:

app.component.ts:

```

import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  isSuccessAlert: boolean = true;
  type="success";
}

```

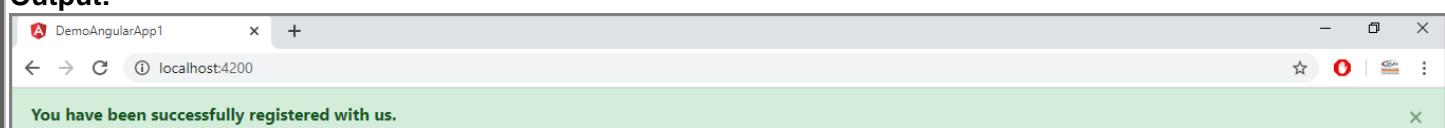
app.component.html:

```

<p>
  <ngb-alert *ngIf="isSuccessAlert"
    [type]="type" [dismissible]="true" (close)="isSuccessAlert=false">
    <strong>You have been successfully registered with us.</strong>
  </ngb-alert>
</p>

```

Output:



DemoAngularApp1

You have been successfully registered with us.

Self Closing Alert:

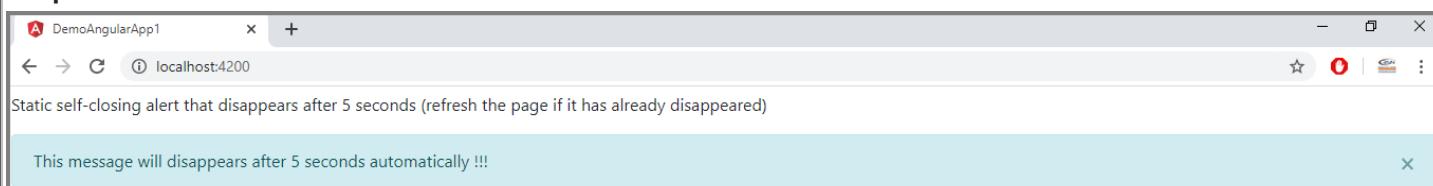
app.component.ts:

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  staticAlertClosed = true;
  ngOnInit(): void{
    setTimeout(() => this.staticAlertClosed = false, 5000);
  }
}
```

app.component.html:

```
<div style="padding: 5px;">
<p>
  Static self-closing alert that disappears after 5 seconds
  (refresh the page if it has already disappeared)
</p>
<ngb-alert *ngIf="staticAlertClosed" type="info" (close)="staticAlertClosed=false">
  This message will disappears after 5 seconds automatically !!!
</ngb-alert>
</div>
```

Output:



Custom alert:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: [
    .alert-custom {
      color: white;
      background-color: #f169b4;
      border-color: #800040;
    }
  ]
})
export class AppComponent {}
```

app.component.html:

```
<p style="padding: 5px;">
  <ngb-alert type="custom" [dismissible]="false">
    <strong>Wow !!!</strong>
    This is a custom alert.
  </ngb-alert>
</p>
```

Output:



Global configuration of alerts:

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbAlertConfig } from '@ng-bootstrap/ng-bootstrap';

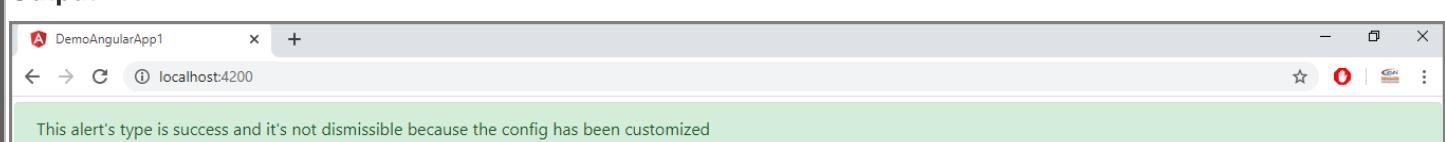
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [NgbAlertConfig] // add NgbAlertConfig to the component providers
})

export class AppComponent {
  constructor(alertConfig: NgbAlertConfig) {
    // customize default values of alerts used by this component tree
    alertConfig.type = 'success';
    alertConfig.dismissible = false;
  }
}
```

app.component.html:

```
<p style="padding: 5px;">
  <ngb-alert>
    This alert's type is success and it's not dismissible because the config has been customized
  </ngb-alert>
</p>
```

Output:



3. Buttons:

NgbCheckBox:

Allows to easily create Bootstrap-style checkbox buttons.

Integrates with forms, so the value of a checked button is bound to the underlying form control either in a reactive or template-driven way.

Selector: [ngbButton][type=checkbox]

NgbRadio:

A directive that marks an input of type "radio" as a part of the NgbRadioGroup.

Selector: [ngbButton][type=radio]

NgbRadioGroup

Allows to easily create Bootstrap-style radio buttons.

Integrates with forms, so the value of a checked button is bound to the underlying form control either in a reactive or template-driven way.

Selector: [ngbRadioGroup]

Examples:

Checkbox buttons:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  model = {
    left: true,
    middle: false,
    right: false
  };
}
```

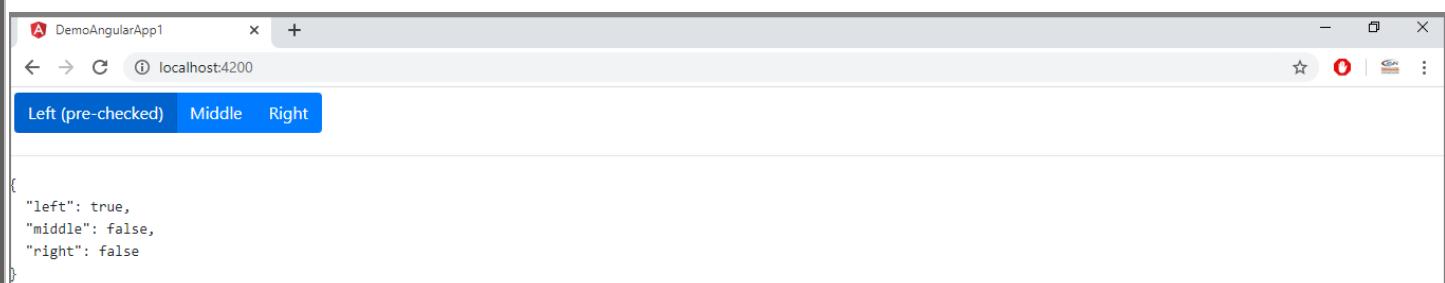
app.component.html:

```
<div class="btn-group btn-group-toggle" style="padding: 5px;">
  <label class="btn-primary" ngbButtonLabel>
    <input type="checkbox" ngbButton [(ngModel)]="model.left"> Left (pre-checked)
  </label>
  <label class="btn-primary" ngbButtonLabel>
    <input type="checkbox" ngbButton [(ngModel)]="model.middle"> Middle
  </label>
  <label class="btn-primary" ngbButtonLabel>
    <input type="checkbox" ngbButton [(ngModel)]="model.right"> Right
  </label>
</div>


---


<pre>{{model | json}}</pre>
```

Output:



```

A DemoAngularApp1      x  +
← → C ① localhost:4200
Left (pre-checked)  Middle  Right

{
  "left": true,
  "middle": false,
  "right": false
}

```

Checkbox buttons (Reactive Forms):

app.component.ts:

```

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  public checkboxGroupForm: FormGroup;

  constructor(private formBuilder: FormBuilder) {}

  ngOnInit() {
    this.checkboxGroupForm = this.formBuilder.group({
      left: true,
      middle: false,
      right: false
    });
  }
}

```

app.component.html:

```

<form [FormGroup]="checkboxGroupForm">
  <div class="btn-group btn-group-toggle" style="padding: 5px;">
    <label class="btn-primary" ngbButtonLabel>
      <input type="checkbox" formControlName="left" ngbButton> Left (pre-checked)
    </label>
    <label class="btn-primary" ngbButtonLabel>
      <input type="checkbox" formControlName="middle" ngbButton> Middle
    </label>
    <label class="btn-primary" ngbButtonLabel>
      <input type="checkbox" formControlName="right" ngbButton> Right
    </label>
  </div>
</form>
<hr>
<pre>{{checkboxGroupForm.value | json}}</pre>

```

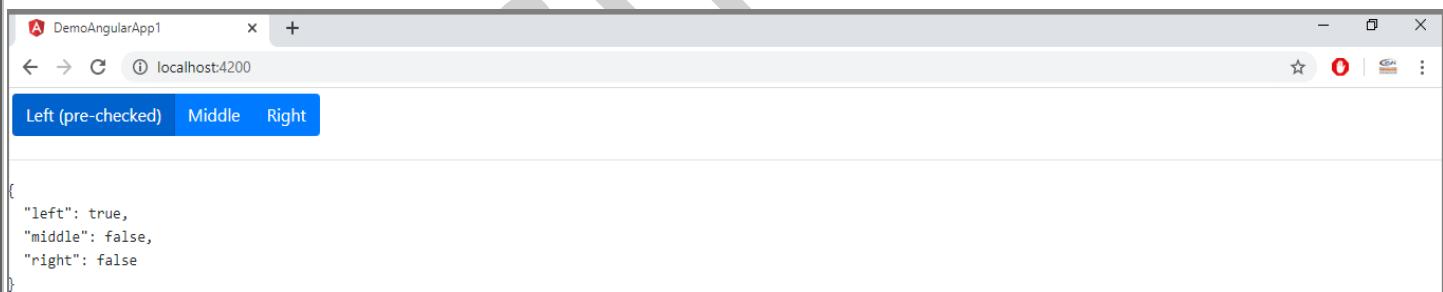


app.module.ts: Import & declare the **ReactiveFormsModule** in root module as shown below:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { NgbModule } from '@ng-bootstrap/ng-bootstrap';
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule, FormsModule, NgbModule, ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Output:



```
{
  "left": true,
  "middle": false,
  "right": false
}
```

Radio buttons:

app.component.ts:

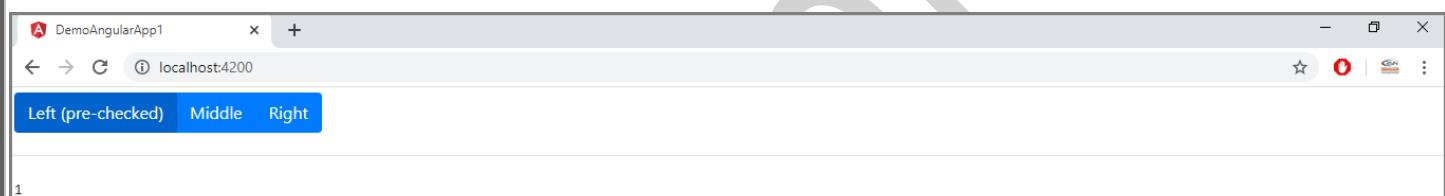
```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  model = 1;
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.component.html:

```
<div style="padding: 5px;" class="btn-group btn-group-toggle"
    ngbRadioGroup name="radioBasic" [(ngModel)]="model">
  <label ngbButtonLabel class="btn-primary">
    <input ngbButton type="radio" [value]="1"> Left (pre-checked)
  </label>
  <label ngbButtonLabel class="btn-primary">
    <input ngbButton type="radio" value="middle"> Middle
  </label>
  <label ngbButtonLabel class="btn-primary">
    <input ngbButton type="radio" [value]="false"> Right
  </label>
</div>
<hr>
<pre style="padding: 5px;">{{model}}
```

Output:



Radio buttons (Reactive Forms):

app.component.ts:

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup } from '@angular/forms';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  public radioGroupForm: FormGroup;

  constructor(private formBuilder: FormBuilder) {}

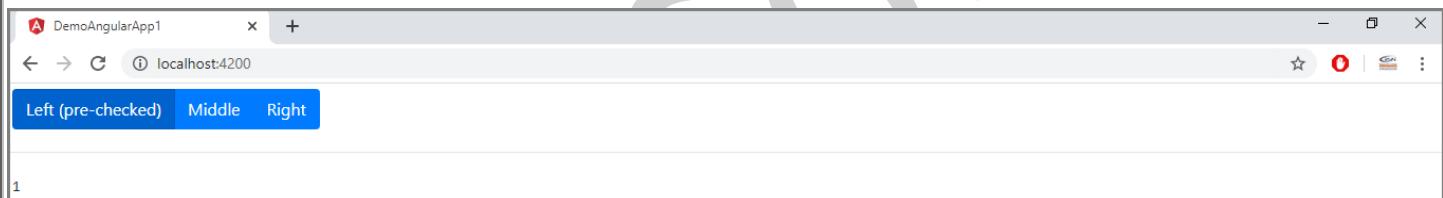
  ngOnInit() {
    this.radioGroupForm = this.formBuilder.group({
      'model': 1
    });
  }
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.component.html:

```
<form [FormGroup]="radioGroupForm">
  <div style="padding: 5px;" class="btn-group btn-group-toggle"
    ngbRadioGroup name="radioBasic"
    formControlName="model">
    <label ngbButtonLabel class="btn-primary">
      <input ngbButton type="radio" [value]="1" checked="checked" /> Left (pre-checked)
    </label>
    <label ngbButtonLabel class="btn-primary">
      <input ngbButton type="radio" value="middle" /> Middle
    </label>
    <label ngbButtonLabel class="btn-primary">
      <input ngbButton type="radio" [value]=false /> Right
    </label>
  </div>
</form>
<hr>
<pre style="padding: 5px;">{{radioGroupForm.value['model']}}</pre>
```

Output:



4. Carousel:

NgbCarousel:

Carousel is a component to easily create and control slideshows.

Allows to set intervals, change the way user interacts with the slides and provides a programmatic API.

Selector: ngb-carousel

Exported as: ngbCarousel

Examples:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  images: string[] = [
    "../assets/Images/banner1.jpg",
    "../assets/Images/banner2.jpg",
    "../assets/Images/banner3.jpg"
  ];
}
```

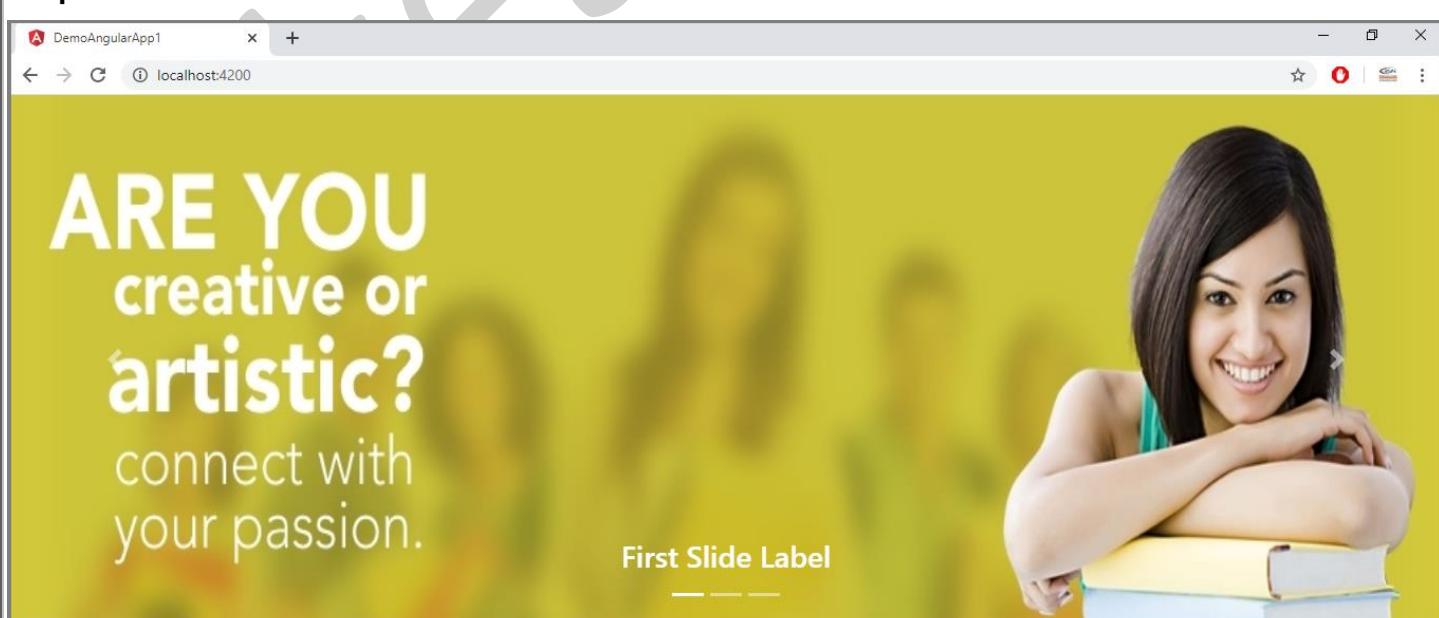
Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



app.component.html:

```
<ngb-carousel *ngIf="images">
  <ng-template ngbSlide>
    <div>
      <img [src]="images[0]" width="100%" height="500">
    </div>
    <div class="carousel-caption">
      <h3>First Slide Label</h3>
    </div>
  </ng-template>
  <ng-template ngbSlide>
    <div>
      <img [src]="images[1]" width="100%" height="500">
    </div>
    <div class="carousel-caption">
      <h3>Second Slide Label</h3>
    </div>
  </ng-template>
  <ng-template ngbSlide>
    <div>
      <img [src]="images[2]" width="100%" height="500">
    </div>
    <div class="carousel-caption">
      <h3>Third Slide Label</h3>
    </div>
  </ng-template>
</ngb-carousel>
```

Output:





Navigation arrows and indicators:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  images: string[] = [
    "../assets/Images/banner1.jpg",
    "../assets/Images/banner2.jpg",
    "../assets/Images/banner3.jpg"
  ];
  showNavigationArrows = false;
  showNavigationIndicators = false;
}
```

app.component.html:

```
<ngb-carousel *ngIf="images" [showNavigationArrows]="showNavigationArrows"
              [showNavigationIndicators]="showNavigationIndicators">
  <ng-template ngbSlide *ngFor="let image of images">
    <div class="picsum-img-wrapper">
      <img [src]="image" width="100%" height="500">
    </div>
    <div class="carousel-caption">
      <h3>{{image.split('/')[3]}} Slide</h3>
    </div>
  </ng-template>
</ngb-carousel>
<hr>
<div class="btn-group" role="group" aria-label="Carousel toggle controls">
  <button type="button" (click)="showNavigationArrows = !showNavigationArrows"
          class="btn btn-outline-dark btn-sm">Toggle navigation arrows</button>
  <button type="button" (click)="showNavigationIndicators = !showNavigationIndicators"
          class="btn btn-outline-dark btn-sm">Toggle navigation indicators</button>
</div>
```

The screenshot shows a web browser window with the URL localhost:4200. The page displays a carousel with three slides. The first slide is titled "banner1.jpg Slide" and features a woman smiling and holding books, with the text "ARE YOU creative or artistic? connect with your passion.". The second and third slides are blurred. At the bottom of the page, there are two buttons: "Toggle navigation arrows" and "Toggle navigation indicators".

Pause/Cycle:

app.component.ts:

```

import { Component, ViewChild } from '@angular/core';
import { NgbCarousel, NgbSlideEvent, NgbSlideEventSource } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  images: string[] = [
    "../assets/Images/banner1.jpg",
    "../assets/Images/banner2.jpg",
    "../assets/Images/banner3.jpg"
  ];

  paused = false;
  unpauseOnArrow = false;
  pauseOnIndicator = false;
  pauseOnHover = true;

  @ViewChild('carousel', {static : true}) carousel: NgbCarousel;

  togglePaused() {
    if (this.paused) {
      this.carousel.cycle();
    } else {
      this.carousel.pause();
    }
    this.paused = !this.paused;
  }

  onSlide(slideEvent: NgbSlideEvent) {
    if (this.unpauseOnArrow && slideEvent.paused &&
      (slideEvent.source === NgbSlideEventSource.ARROW_LEFT || slideEvent.source === NgbSlideEventSource.ARROW_RIGHT)) {
      this.togglePaused();
    }
    if (this.pauseOnIndicator && !slideEvent.paused && slideEvent.source === NgbSlideEventSource.INDICATOR) {
      this.togglePaused();
    }
  }
}

```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.component.html:

```

<ngb-carousel #carousel interval="1000" [pauseOnHover]="pauseOnHover"
              (slide)="onSlide($event)">
  <ng-template ngbSlide *ngFor="let img of images; index as i">
    <div class="carousel-caption">
      <h3>My Slide {{i + 1}}</h3>
    </div>
    <div>
      <img [src]="img" alt="My Image {{i + 1}}" width="100%" height="500">
    </div>
  </ng-template>
</ngb-carousel>
<hr>
<div style="padding: 5px;">
  <div class="form-check">
    <input type="checkbox" class="form-check-input" id="pauseOnHover"
           [(ngModel)]="pauseOnHover">
    <label class="form-check-label" for="pauseOnHover">Pause on hover</label>
  </div>

  <div class="form-check">
    <input type="checkbox" class="form-check-input" id="unpauseOnArrow"
           [(ngModel)]="unpauseOnArrow">
    <label class="form-check-label" for="unpauseOnArrow">
      Unpause when clicking on arrows
    </label>
  </div>

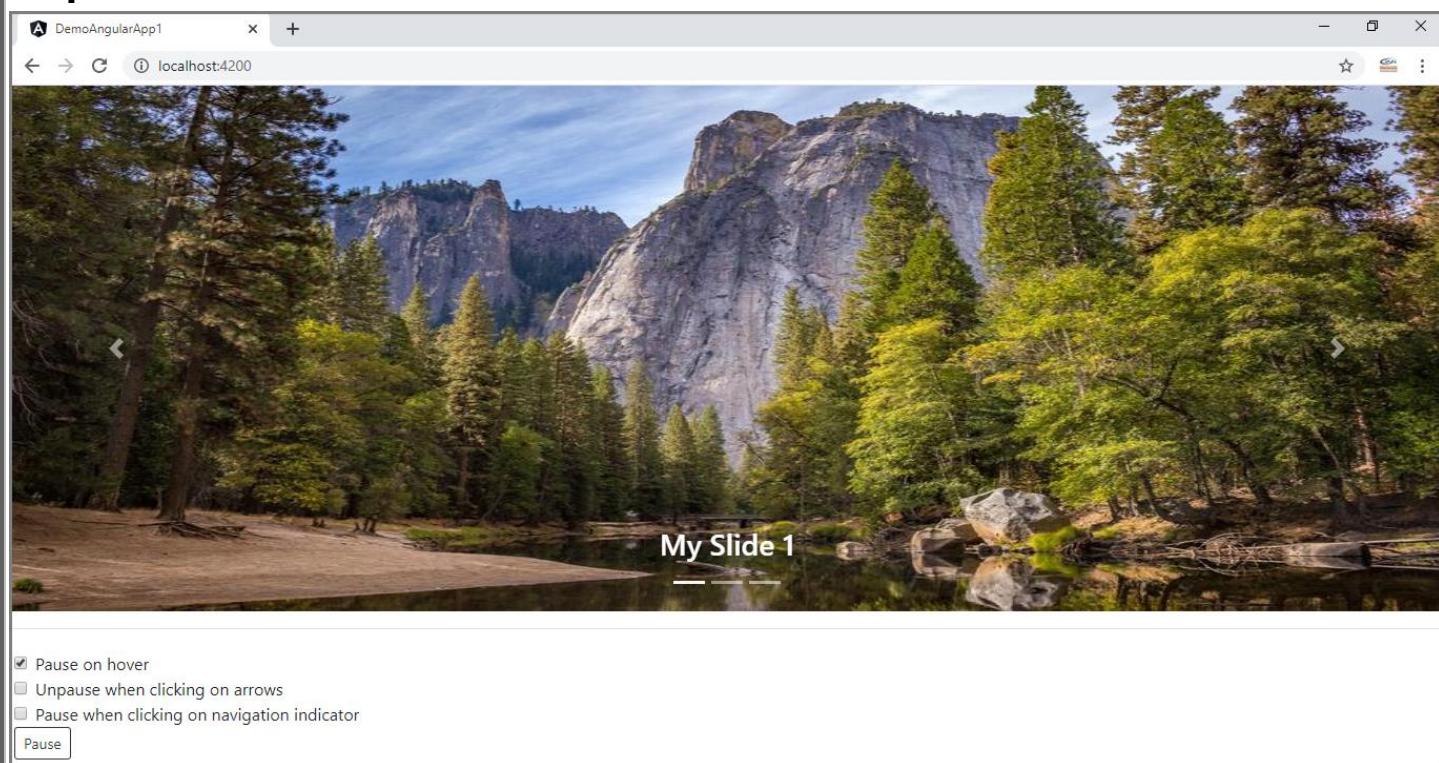
  <div class="form-check">
    <input type="checkbox" class="form-check-input" id="pauseOnIndicator"
           [(ngModel)]="pauseOnIndicator">
    <label class="form-check-label" for="pauseOnIndicator">
      Pause when clicking on navigation indicator
    </label>
  </div>

  <button type="button" (click)="togglePaused()" class="btn btn-outline-dark btn-sm">
    {{paused ? 'Cycle' : 'Pause' }}
  </button>
</div>

```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Output:



Global configuration of carousels:

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbCarouselConfig } from '@ng-bootstrap/ng-bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [NgbCarouselConfig] // add NgbCarouselConfig to the component providers
})
export class AppComponent {
  images: string[] = [
    "../assets/Images/banner1.jpg",
    "../assets/Images/banner2.jpg",
    "../assets/Images/banner3.jpg"
  ];
  constructor(config: NgbCarouselConfig) {
    // customize default values of carousels
    config.interval = 10000;
    config.wrap = false;
    config.keyboard = false;
    config.pauseOnHover = false;
  }
}
```

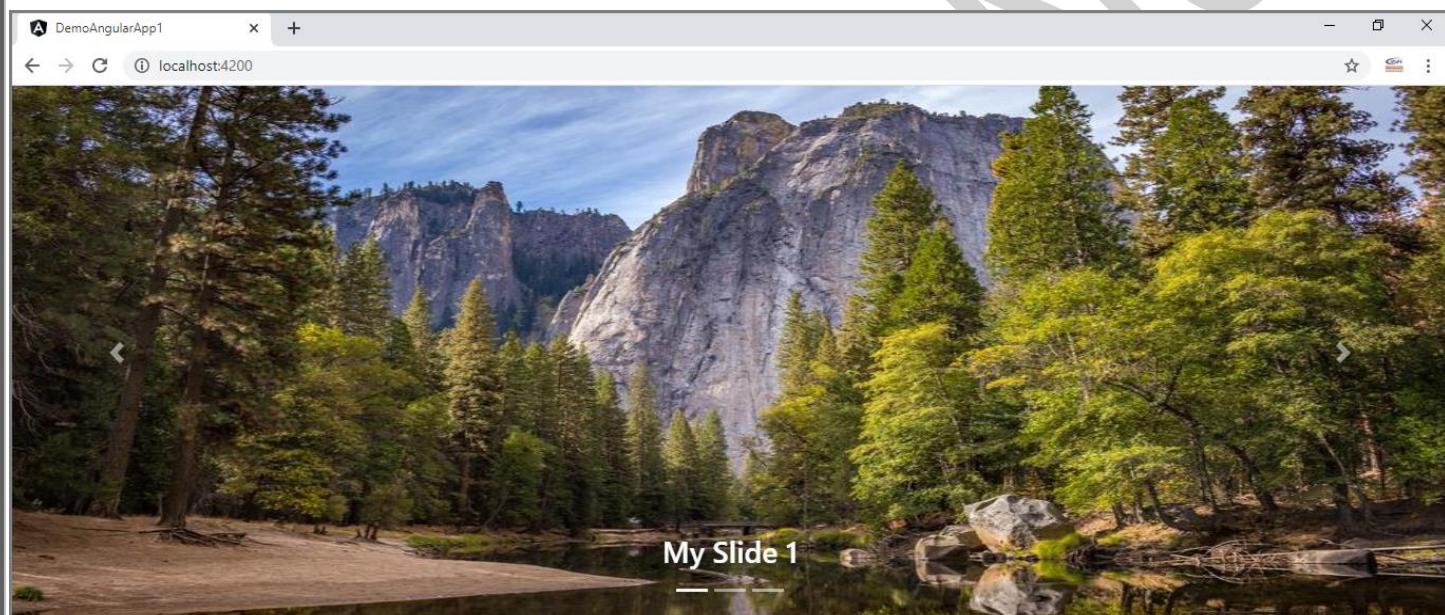
app.component.html:

```

<ngb-carousel>
  <ng-template ngbSlide *ngFor="let img of images; index as i">
    <div>
      <img [src]="img" alt="My Image {{i + 1}}" width="100%" height="500">
    </div>
    <div class="carousel-caption">
      <h3>My Slide {{i + 1}}</h3>
    </div>
  </ng-template>
</ngb-carousel>

```

Output:



5. Collapse

NgbCollapse

A directive to provide a simple way of hiding and showing elements on the page.

Selector: [ngbCollapse]

Exported as: NgbCollapse

Example 1:

app.component.ts:

```

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

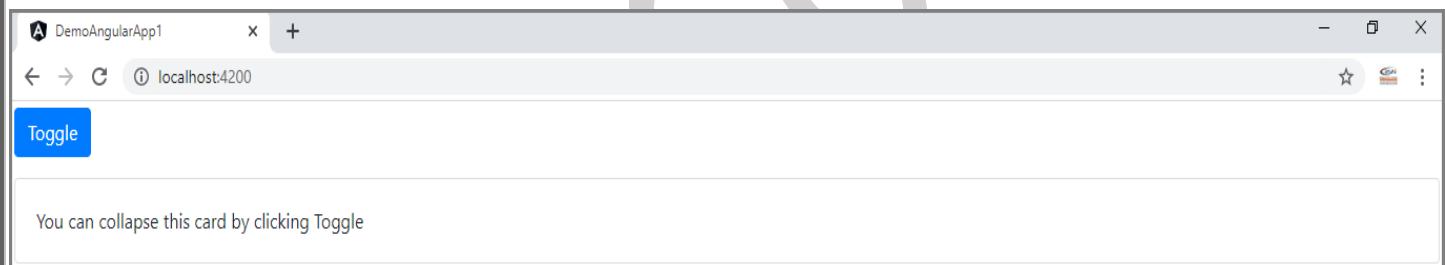
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.component.html:

```
<div style="padding: 5px;">
  <p>
    <button type="button" class="btn btn-primary" (click)="isCollapsed=!isCollapsed">
      Toggle
    </button>
  </p>
  <div [ngbCollapse]="isCollapsed">
    <div class="card">
      <div class="card-body">
        You can collapse this card by clicking Toggle
      </div>
    </div>
  </div>
</div>
```

Output:



Example2:

Responsive Navbar:

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

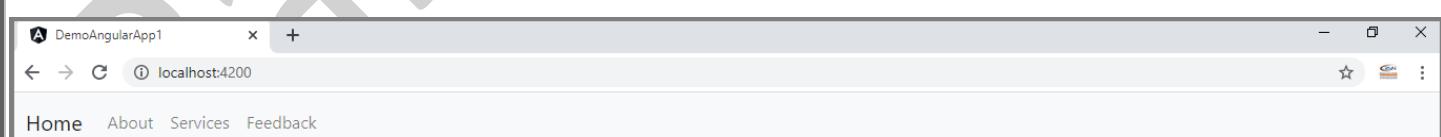
export class AppComponent {
  public isMenuCollapsed = true;
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

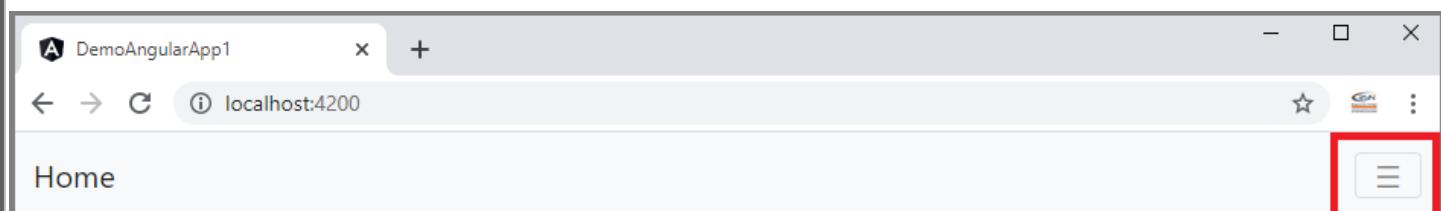
app.component.html:

```
<nav class="navbar navbar-expand-lg navbar-light bg-light mb-3">
  <a class="navbar-brand" [routerLink]="">Home</a>
  <!--Toggle the value of the property when the toggler button is clicked.-->
  <button class="navbar-toggler" type="button" (click)="isMenuCollapsed=!isMenuCollapsed">
    &#9776;
  </button>
  <!--Add the ngbCollapse directive to the element below.-->
  <div [ngbCollapse]="isMenuCollapsed" class="collapse navbar-collapse">
    <ul class="navbar-nav">
      <li class="nav-item">
        <a class="nav-link" [routerLink]="" (click)="isMenuCollapsed=true">
          About
        </a>
      </li>
      <li class="nav-item">
        <a class="nav-link" [routerLink]="" (click)="isMenuCollapsed=true">
          Services
        </a>
      </li>
      <li class="nav-item">
        <a class="nav-link" [routerLink]="" (click)="isMenuCollapsed=true">
          Feedback
        </a>
      </li>
    </ul>
  </div>
</nav>
```

Output:



Now resize your browser window to see it in action!



6. Datepicker

Datepicker will help you with date selection. It can be used either inline with NgbDatepicker component or as a popup on any input element with NgbInputDatepicker directive.

Basic datepicker:

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbDateStruct, NgbCalendar } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  preserveWhitespaces: true
})
export class AppComponent {
  model: NgbDateStruct;
  date: {year: number, month: number};

  constructor(private calendar: NgbCalendar) {}

  selectToday() {
    this.model = this.calendar.getToday();
  }
}
```

app.component.html:

```
<div style="padding: 5px;">
  <p>Simple Datepicker</p>
  <ngb-datepicker #dp [(ngModel)]="model" (navigate)="date=$event.next"></ngb-datepicker>
  <hr />
  <button class="btn btn-primary" (click)="selectToday()">Select Today</button>
  <button class="btn btn-primary" (click)="dp.navigateTo()">To Current Month</button>
  <button class="btn btn-primary" (click)="dp.navigateTo({year: 2019, month: 2})">
    To Feb 2019
  </button>
  <hr />
  <pre>Month: {{ date.month }}.{{ date.year }}</pre>
  <pre>Model: {{ model | json }}</pre>
</div>
```

Output:

```
Month: 1.2020
Model: {
  "year": 2020,
  "month": 1,
  "day": 17
}
```

Datepicker in a popup:

app.component.ts:

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

app.component.html:

```
<form class="form-inline">
  <div class="form-group">
    <div class="input-group">
      <input class="form-control" placeholder="yyyy-mm-dd" name="dp"
        [(ngModel)]="model" ngbDatepicker #d="ngbDatepicker">
      <div class="input-group-append">
        <button class="btn btn-outline-secondary fa fa-calendar"
          (click)="d.toggle()" type="button">
      </button>
    </div>
  </div>
</form>
<hr />
<pre *ngIf="model">Model: {{ model | json }}</pre>
```

Include 'Font Awesome' CSS CDN link in index.html for all types of icons like calendar icon to use in application as following:

```
<link rel="stylesheet"
  href="https://stackpath.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css"/>
```

Output:

Using Icons in Bootstrap 4

The new Bootstrap 4 version doesn't include an icon library by default, unlike the previous Bootstrap 3 version that includes Glyphicons in the font format in its core.

However, you can still include icons in your project using several external font based icon library. The most popular and highly compatible icon library for Bootstrap is Font Awesome. It provides 675+ icons which are available in font format for better usability and scalability.

Let's see how to include font-awesome icons in a web page.

How to Include Font Awesome:

You can simply use the freely available font-awesome CDN link to include the font-awesome icons in your project. This CDN link basically points to a remote CSS file that includes all the necessary classes to generate variety of icons. Let's take a look at the following example:

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
  <title>Including Font Awesome in Bootstrap Template</title>
  <!-- Bootstrap CSS -->
  <link rel="stylesheet"
    href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"/>
  <!-- Font Awesome CSS -->
  <link rel="stylesheet"
    href="https://stackpath.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css"/>
</head>
<body>
  <h1><i class="fa fa-globe"></i> Hello, world!</h1>
  <!-- JS files: jQuery first, then Popper.js, then Bootstrap JS -->
  <script src="https://code.jquery.com/jquery-3.4.1.slim.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.16.0/umd/popper.min.js">
    </script>
  <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js">
    </script>
</body>
</html>
```

How to Use Font Awesome Icons in Your Code

To use font-awesome icons in your code you'll require an `<i>` tag along with a base class `.fa` and an individual icon class `.fa-*`. The general syntax for using font-awesome icons is:

```
<i class="fa fa-class-name"></i>
```

Where `fa-class-name` is the name of the particular icon class (e.g. `fa-search`, `fa-user`, `fa-star`, `fa-calendar`, `fa-globe`, etc.) defined in `font-awesome.min.css` file.

For example, to use search icon you can place the following code just about anywhere:

```
<button type="submit" class="btn btn-primary"><span class="fa fa-search"></span> Search</button>
<button type="submit" class="btn btn-secondary"><span class="fa fa-search"></span> Search</button>
```

The output of the above example will look something like this:



Multiple months:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: [
    select.custom-select {
      margin: 0.5rem 0.5rem 0 0;
      width: auto;
    }
  ]
})
export class AppComponent {
  displayMonths = 2;
  navigation = 'select';
  showWeekNumbers = false;
  outsideDays = 'visible';
}
```

app.component.html:

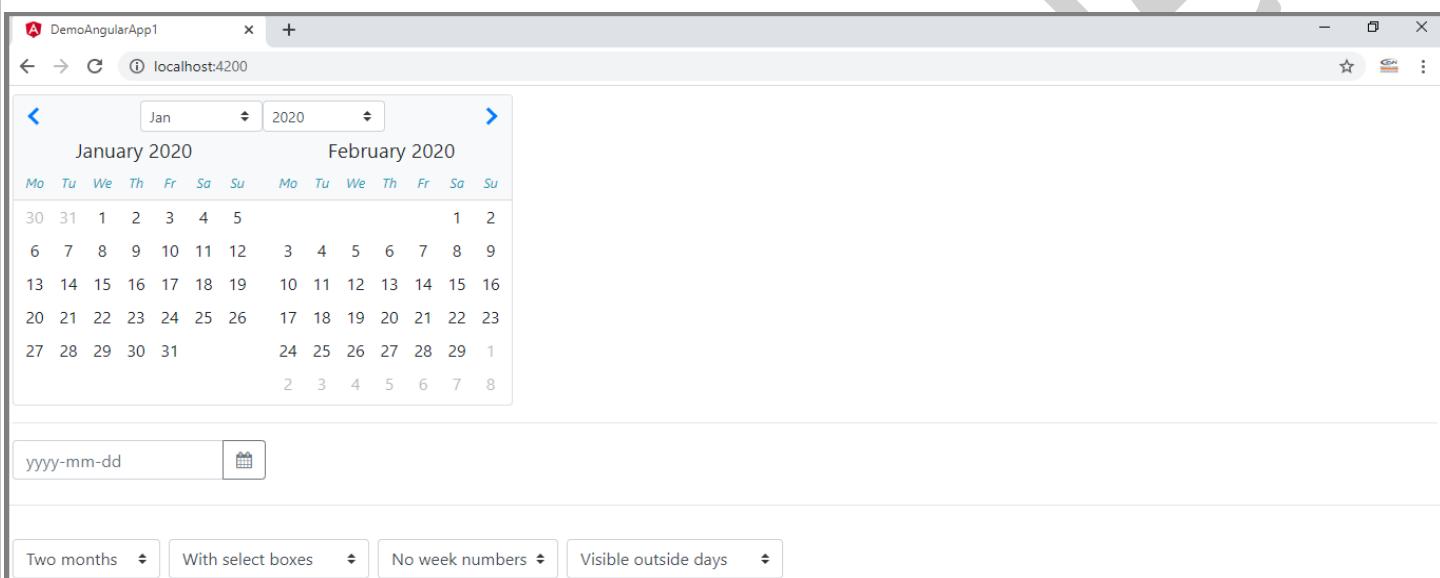
```
<div class="p-2">
  <ngb-datepicker [displayMonths]="displayMonths" [navigation]="navigation"
    [showWeekNumbers]="showWeekNumbers" [outsideDays]="outsideDays">
  </ngb-datepicker>
  <hr />
  <form class="form-inline">
    <div class="form-group">
      <div class="input-group">
        <input class="form-control" placeholder="yyyy-mm-dd" name="dp"
          [displayMonths]="displayMonths" [navigation]="navigation"
          [outsideDays]="outsideDays" [showWeekNumbers]="showWeekNumbers"
          ngbDatepicker #d="ngbDatepicker">
        <div class="input-group-append">
          <button type="button" class="btn btn-outline-secondary fa fa-calendar"
            (click)="d.toggle()">
          </button>
        </div>
      </div>
    </div>
  </form>
</div>
<hr />
<div class="d-flex flex-wrap align-content-between p-2">
  <select class="custom-select" [(ngModel)]="displayMonths">
    <option [ngValue]="1">One month</option>
    <option [ngValue]="2">Two months</option>
    <option [ngValue]="3">Three months</option>
  </select>
  <select class="custom-select" [(ngModel)]="navigation">
    <option value="none">Without navigation</option>
    <option value="select">With select boxes</option>
    <option value="arrows">Without select boxes</option>
  </select>
</div>
```

```

<select class="custom-select" [(ngModel)]="showWeekNumbers">
  <option [ngValue]="true">Week numbers</option>
  <option [ngValue]="false">No week numbers</option>
</select>
<select class="custom-select" [(ngModel)]="outsideDays">
  <option value="visible">Visible outside days</option>
  <option value="hidden">Hidden outside days</option>
  <option value="collapsed">Collapsed outside days</option>
</select>
</div>

```

Output:



Range selection:

app.component.ts:

```

import { Component } from '@angular/core';
import { NgbDate, NgbCalendar } from '@ng-bootstrap/ng-bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: [
    .custom-day {
      text-align: center;
      padding: 0.185rem 0.25rem;
      display: inline-block;
      height: 2rem;
      width: 2rem;
    }
    .custom-day.focused {
      background-color: #e6e6e6;
    }
  ]
})

```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

```

        .custom-day.range, .custom-day:hover {
            background-color: rgb(2, 117, 216);
            color: white;
        }
        .custom-day.faded {
            background-color: rgba(2, 117, 216, 0.5);
        }
    ]
})

export class AppComponent {
    hoveredDate: NgbDate;

    fromDate: NgbDate;
    toDate: NgbDate;

    constructor(calendar: NgbCalendar) {
        this.fromDate = calendar.getToday();
        this.toDate = calendar.getNext(calendar.getToday(), 'd', 10);
    }

    onDateSelection(date: NgbDate) {
        if (!this.fromDate && !this.toDate) {
            this.fromDate = date;
        } else if (this.fromDate && !this.toDate && date.after(this.fromDate)) {
            this.toDate = date;
        } else {
            this.toDate = null;
            this.fromDate = date;
        }
    }

    isHovered(date: NgbDate) {
        return this.fromDate && !this.toDate && this.hoveredDate && date.after(this.fromDate) && date.before(this.hoveredDate);
    }

    isInside(date: NgbDate) {
        return date.after(this.fromDate) && date.before(this.toDate);
    }

    isRange(date: NgbDate) {
        return date.equals(this.fromDate) || date.equals(this.toDate) || this.isInside(date) || this.isHovered(date);
    }
}

```

app.component.html:

```

<div class="p-2">
    <ngb-datepicker #dp (select)="onDateSelection($event)" [displayMonths]="2"
                    [dayTemplate]="t" outsideDays="hidden">
    </ngb-datepicker>

    <ng-template #t let-date let-focused="focused">
        <span class="custom-day"
              [class.focused]="focused" [class.range]="isRange(date)"
              [class.faded]="isHovered(date) || isInside(date)"
              (mouseenter)="hoveredDate=date" (mouseleave)="hoveredDate = null">
            {{ date.day }}
        </span>
    </ng-template>
    <hr>
    <pre>From: {{ fromDate | json }} </pre>
    <pre>To: {{ toDate | json }} </pre>
</div>

```

Output:



The screenshot shows a browser window with the URL localhost:4200. At the top, there is a date range picker component. The calendar displays two months: January 2020 and February 2020. The days are arranged in a grid. The days in January are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18 (highlighted in blue), 19 (highlighted in blue), 20, 21, 22, 23, 24, 25, 26, 27, 28 (highlighted in blue), 29, 30, 31. The days in February are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29.

```

From: {
  "year": 2020,
  "month": 1,
  "day": 18
}

To: {
  "year": 2020,
  "month": 1,
  "day": 28
}

```

Range selection in a popup:

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbDate, NgbCalendar, NgbDateParserFormatter } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: [
    .form-group.hidden {
      width: 0;
      margin: 0;
      border: none;
      padding: 0;
    }
    .custom-day {
      text-align: center;
      padding: 0.185rem 0.25rem;
      display: inline-block;
      height: 2rem;
      width: 2rem;
    }
    .custom-day.focused {
      background-color: #e6e6e6;
    }
    .custom-day.range, .custom-day:hover {
      background-color: rgb(2, 117, 216);
      color: white;
    }
    .custom-day.faded {
      background-color: rgba(2, 117, 216, 0.5);
    }
  ]
})
export class AppComponent {
  hoveredDate: NgbDate;

  fromDate: NgbDate;
  toDate: NgbDate;

  constructor(private calendar: NgbCalendar, public formatter: NgbDateParserFormatter) {
    this.fromDate = calendar.getToday();
    this.toDate = calendar.getNext(calendar.getToday(), 'd', 10);
  }
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

```

onDateSelection(date: NgbDate) {
  if (!this.fromDate && !this.toDate) {
    this.fromDate = date;
  } else if (this.fromDate && !this.toDate && date.after(this.fromDate)) {
    this.toDate = date;
  } else {
    this.toDate = null;
    this.fromDate = date;
  }
}

isHovered(date: NgbDate) {
  return this.fromDate && !this.toDate && this.hoveredDate && date.after(this.fromDate) &&
date.before(this.hoveredDate);
}

isInside(date: NgbDate) {
  return date.after(this.fromDate) && date.before(this.toDate);
}

isRange(date: NgbDate) {
  return date.equals(this.fromDate) || date.equals(this.toDate) || this.isInside(date) ||
this.isHovered(date);
}

validateInput(currentValue: NgbDate, input: string): NgbDate {
  const parsed = this.formatter.parse(input);
  return parsed && this.calendar.isValid(NgbDate.from(parsed)) ? NgbDate.from(parsed) :
currentValue;
}
}

```

app.component.html:

```

<div class="p-2">
  <form class="form-inline">
    <div class="form-group hidden">
      <div class="input-group">
        <input name="datepicker" class="form-control" ngbDatepicker
          #datepicker="ngbDatepicker"
          [autoClose]="'outside'" (dateSelect)="onDateSelection($event)"
          [displayMonths]="2" [dayTemplate]="t"
          [startDate]="'fromDate" outsideDays="hidden">

```

```

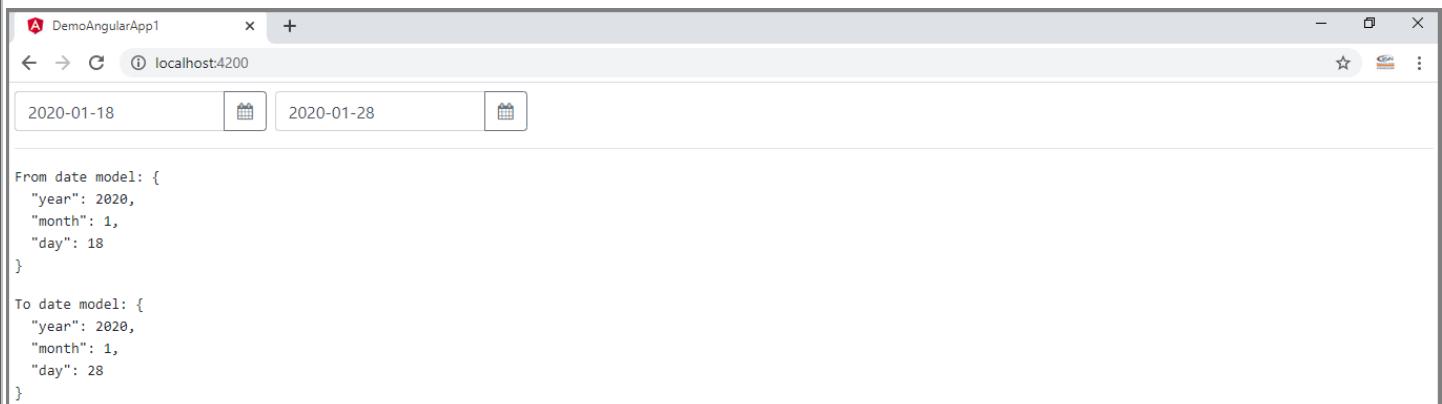
<ng-template #t let-date let-focused="focused">
  <span class="custom-day" [class.focused]="focused"
        [class.range]="isRange(date)"
        [class.faded]="isHovered(date) || isInside(date)"
        (mouseenter)="hoveredDate=date" (mouseleave)="hoveredDate=null">
    {{ date.day }}
  </span>
</ng-template>
</div>
</div>
<div class="form-group">
  <div class="input-group">
    <input #dpFromDate class="form-control" placeholder="yyyy-mm-dd"
           name="dpFromDate" [value]="formatter.format(fromDate)"
           (input)="fromDate=validateInput(fromDate, dpFromDate.value)">
    <div class="input-group-append">
      <button type="button" class="btn btn-outline-secondary fa fa-calendar"
              (click)="datepicker.toggle()">
      </button>
    </div>
  </div>
</div>
<div class="form-group ml-2">
  <div class="input-group">
    <input #dpToDate class="form-control" placeholder="yyyy-mm-dd"
           name="dpToDate" [value]="formatter.format(toDate)"
           (input)="toDate = validateInput(toDate, dpToDate.value)">
    <div class="input-group-append">
      <button type="button" class="btn btn-outline-secondary fa fa-calendar"
              (click)="datepicker.toggle()">
      </button>
    </div>
  </div>
</div>
</form>
<hr />
<pre>From date model: {{ fromDate | json }}</pre>
<pre>To date model: {{ toDate | json }}</pre>

```

</div>

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Output:



```
From date model: {
  "year": 2020,
  "month": 1,
  "day": 18
}

To date model: {
  "year": 2020,
  "month": 1,
  "day": 28
}
```

Disabled datepicker:

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbCalendar, NgbDateStruct } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  model: NgbDateStruct;
  disabled = true;
```

app.component.html:

```
<div class="p-2">
  <ngb-datepicker [(ngModel)]="model" [disabled]="disabled"></ngb-datepicker>
  <hr />
  <button class="btn btn-sm btn-outline-{{disabled ? 'danger' : 'success'}}"
    (click)="disabled = !disabled">
    {{ disabled ? "disabled" : "enabled"}}
  </button>
</div>
```



Output:

A DemoAngularApp1 localhost:4200

Real Time Live Project

Jan 2020

Mo Tu We Th Fr Sa Su

30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

disabled

Custom date adapter:

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbDateAdapter, NgbDateStruct, NgbDateNativeAdapter} from '@ng-bootstrap/ng-
bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [{provide: NgbDateAdapter, useClass: NgbDateNativeAdapter}]
})
export class AppComponent {
  model1: Date;
  model2: Date;

  get today() {
    return new Date();
  }
}
```

app.component.html:

```
<div class="p-2">
  <div class="row">
    <div class="col-6">
      <ngb-datepicker #d1 [(ngModel)]="model1" #c1="ngModel"></ngb-datepicker>
      <hr />
      <button class="btn btn-sm btn-outline-primary" (click)="model1=today">
        Select Today
      </button>
      <hr />
      <pre>Model: {{ model1 | json }}</pre>
      <pre>State: {{ c1.status }}</pre>
    </div>
  </div>
```



```

<div class="col-6">
    <form class="form-inline">
        <div class="form-group">
            <div class="input-group">
                <input class="form-control" placeholder="yyyy-mm-dd" name="d2"
                    #c2="ngModel"
                    [(ngModel)]="model2"
                    NgbDatepicker
                    #d2="ngbdatepicker">
                <div class="input-group-append">
                    <button class="btn btn-outline-secondary fa fa-calendar"
                        type="button" (click)="d2.toggle()">
                    </button>
                </div>
            </div>
        </div>
    </form>
    <hr />
    <button class="btn btn-sm btn-outline-primary"
        (click)="model2=today">
        Select Today
    </button>
    <hr />
    <pre>Model: {{ model2 | json }}</pre>
    <pre>State: {{ c2.status }}</pre>
</div>
</div>
</div>

```

Output:

DemoAngularApp1

localhost:4200

Jan 2020

Mo	Tu	We	Th	Fr	Sa	Su
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

yyyy-mm-dd

Select Today

Model:

State: VALID

Custom day view:

This datepicker uses a custom template to display days. All week-ends are displayed with an orange background.

app.component.ts:

```

import { Component } from '@angular/core';
import { NgbCalendar, NgbDate, NgbDateFormat } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles : [
    .custom-day {
      text-align: center;
      padding: 0.185rem 0.25rem;
      border-radius: 0.25rem;
      display: inline-block;
      width: 2rem;
    }
    .custom-day:hover, .custom-day.focused {
      background-color: #e6e6e6;
    }
    .weekend {
      background-color: #f0ad4e;
      border-radius: 1rem;
      color: white;
    }
    .hidden {
      display: none;
    }
  ]
})
export class AppComponent {
  model: NgbDateStruct;

  constructor(private calendar: NgbCalendar) {}

  isDisabled = (date: NgbDate, current: {month: number}) => date.month !== current.month;
  isWeekend = (date: NgbDate) => this.calendar.getWeekday(date) >= 6;
}

```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.component.html

```

<div class="p-2">
  <form class="form-inline">
    <div class="form-group">
      <div class="input-group">
        <input class="form-control" placeholder="yyyy-mm-dd" name="dp"
          [(ngModel)]="model" ngbDatepicker [dayTemplate]="customDay"
          [markDisabled]="isDisabled" #d="ngbDatepicker">
        <div class="input-group-append">
          <button class="btn btn-outline-secondary fa fa-calendar" type="button"
            (click)="d.toggle()">
        </button>
      </div>
    </div>
  </div>
</form>

<ng-template #customDay let-date let-currentMonth="currentMonth"
  let-selected="selected" let-disabled="disabled" let-focused="focused">
  <span class="custom-day" [class.weekend]="isWeekend(date)"
    [class.focused]="focused"
    [class.bg-primary]="selected"
    [class.hidden]="date.month!==currentMonth"
    [class.text-muted]="disabled">
    {{ date.day }}
  </span>
</ng-template>
</div>

```

Output:



Footer template:

This datepicker uses a footer template which is presented inside datepicker. Today and close buttons used as an example.

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbCalendar, NgbDateStruct } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls : ['./app.component.css']
})
export class AppComponent {
  model: NgbDateStruct;
  today = this.calendar.getToday();

  constructor(private calendar: NgbCalendar) {}
}
```

app.component.html:

```
<div class="p-2">
  <form class="form-inline">
    <div class="form-group">
      <div class="input-group">
        <input class="form-control" placeholder="yyyy-mm-dd" name="dp"
          [(ngModel)]="model" ngbdatepicker [footerTemplate]="footerTemplate"
          #d="ngbDatePicker">
        <div class="input-group-append">
          <button class="btn btn-outline-secondary fa fa-calendar" type="button"
            (click)="d.toggle()">
            </button>
        </div>
      </div>
    </div>
  </form>
<ng-template #footerTemplate>
  <hr class="my-0">
  <button class="btn btn-primary btn-sm m-2 float-left"
    (click)="model=today; d.close()">Today</button>
  <button class="btn btn-secondary btn-sm m-2 float-right"
    (click)="d.close()">Close</button>
</ng-template>
</div>
```

Output:



Position target:

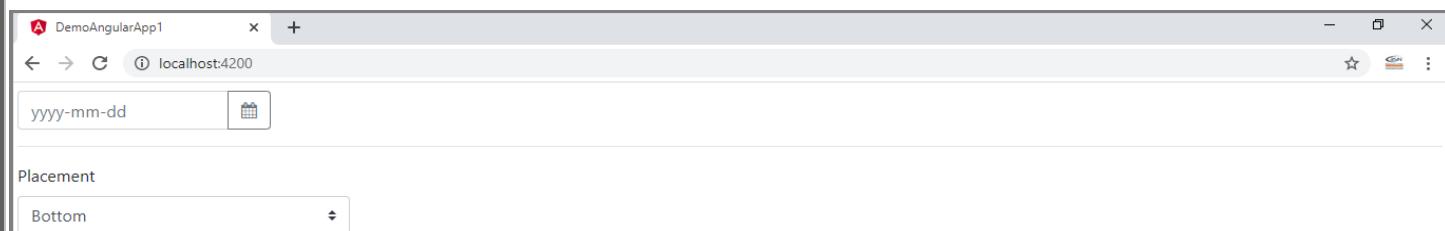
app.component.ts:

```
import { Component } from '@angular/core';
import { NgbDateStruct } from '@ng-bootstrap/ng-bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  model: NgbDateStruct;
  placement = 'bottom';
}
```

app.component.html:

```
<div class="p-2">
  <form class="form-inline">
    <div class="form-group">
      <div class="input-group">
        <input class="form-control" placeholder="yyyy-mm-dd"
              name="dp" [(ngModel)]="model" ngbdatepicker #d="ngbDatepicker"
              [placement]="placement" [positionTarget]="buttonEl">
        <div class="input-group-append">
          <button #buttonEl class="btn btn-outline-secondary calendar" type="button"
                 (click)="d.toggle()">
            </button>
        </div>
      </div>
    </div>
    <hr/>
    <div class="row">
      <div class="col-sm-3">
        <label for="placement">Placement</label>
        <select id="placement" class="custom-select form-control" [(ngModel)]="placement">
          <option value="top">Top</option>
          <option value="bottom">Bottom</option>
          <option value="left">Left</option>
          <option value="right">Right</option>
        </select>
      </div>
    </div>
  </div>
```

Output:



Global configuration of datepickers:

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbDatepickerConfig, NgbCalendar, NgbDate, NgbDateStruct } from '@ng-bootstrap/ng-bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls : ['./app.component.css'],
  providers: [NgbDatepickerConfig] // add NgbDatepickerConfig to the component providers
})
export class AppComponent {
  model: NgbDateStruct;
  constructor(config: NgbDatepickerConfig, calendar: NgbCalendar) {
    // customize default values of datepickers used by this component tree
    config.minDate = {year: 1900, month: 1, day: 1};
    config.maxDate = {year: 2020, month: 12, day: 31};
    // days that don't belong to current month are not visible
    config.outsideDays = 'hidden';
    // weekends are disabled
    config.markDisabled = (date: NgbDate) => calendar.getWeekday(date) >= 6;
  }
}
```

app.component.html:

```
<div class="p-2">
  <ngb-datepicker [(ngModel)]="model"></ngb-datepicker>
</div>
```

Output:



7. Dropdown

NgbDropdown - Directive

A directive that provides contextual overlays for displaying lists of links and more.

Selector: [ngbDropdown]

Exported as: ngbDropdown

Example1:

app.component.ts:

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

app.component.html:

```
<div ngbDropdown class="p-2">
  <button class="btn btn-outline-primary" id="dropdownBasic1" ngbDropdownToggle>
    Select Action
  </button>
  <div ngbDropdownMenu aria-labelledby="dropdownBasic1">
    <button ngbDropdownItem>Action - 1</button>
    <button ngbDropdownItem>Action - 2</button>
    <button ngbDropdownItem>Action - 3</button>
  </div>
</div>
```

Output:



ngbDropdownToggle – Directive:

A directive to mark an element that will toggle dropdown via the click event.

You can also use **NgbDropdownAnchor** as an alternative.

Selector [ngbDropdownToggle]

ngbDropdownMenu – Directive:

A directive that wraps dropdown menu content and dropdown items.

Selector [ngbDropdownMenu]

Example2:

How to configure NgbDropdown to display the selected item from the dropdown?

app.component.ts:

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

  ChangeSelection(newSelectionAction: string) {
    this.selectedAction = newSelectionAction;
  }
}
```

app.component.html:

```
<div ngbDropdown class="p-2">
  <button class="btn btn-outline-primary" id="dropdownBasic1" ngbDropdownToggle>
    {{selectedAction}}
  </button>
  <div ngbDropdownMenu aria-labelledby="dropdownBasic1">
    <button #b1 ngbDropdownItem (click)="ChangeSelection(b1.textContent)">
      Action - 1
    </button>
    <button #b2 ngbDropdownItem (click)="ChangeSelection(b2.textContent)">
      Action - 2
    </button>
    <button #b3 ngbDropdownItem (click)="ChangeSelection(b3.textContent)">
      Action - 3
    </button>
  </div>
</div>
```

Alternate Option:

app.component.ts:

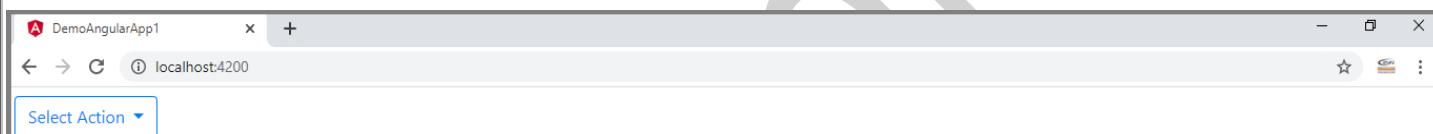
```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls : ['./app.component.css']
})
export class AppComponent {
  selectedAction: string = "Select Action";
  ChangeSelection(element: any) {
    this.selectedAction = element.textContent;
  }
}
```



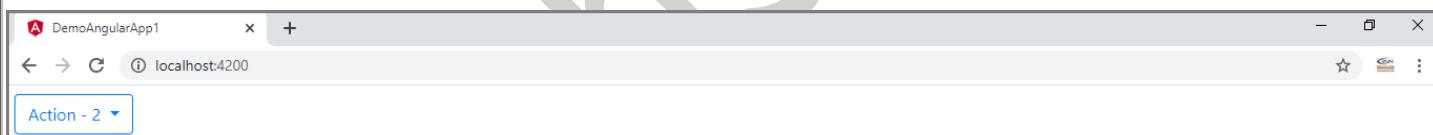
app.component.html:

```
<div ngbDropdown class="p-2">
  <button class="btn btn-outline-primary" id="dropdownBasic1" ngbDropdownToggle>
    {{selectedAction}}
  </button>
  <div ngbDropdownMenu aria-labelledby="dropdownBasic1">
    <button ngbDropdownItem (click)="ChangeSelection($event.target)">
      Action - 1
    </button>
    <button ngbDropdownItem (click)="ChangeSelection($event.target)">
      Action - 2
    </button>
    <button ngbDropdownItem (click)="ChangeSelection($event.target)">
      Action - 3
    </button>
  </div>
</div>
```

Output:



After Selecting Any of the Dropdown Option:



Example3:

Programmatically binding the Dropdown Items:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls : ['./app.component.css']
})
export class AppComponent {
  actions: string[] = ["Action - 1", "Action - 2", "Action - 3"];

  selectedAction: string = "Select Action";

  ChangeSelection(action:string) {
    this.selectedAction = action;
  }
}
```

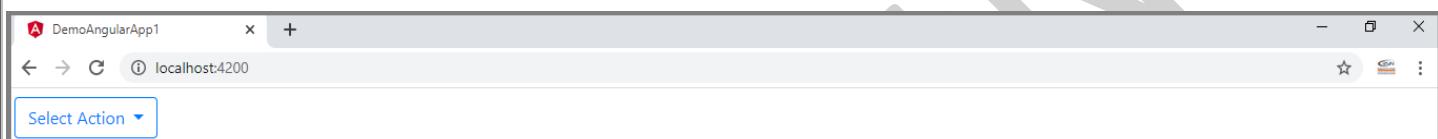
Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



app.component.html:

```
<div ngbDropdown class="p-2">
  <button class="btn btn-outline-primary" id="dropdownBasic1" ngbDropdownToggle>
    {{selectedAction}}
  </button>
  <div ngbDropdownMenu aria-labelledby="dropdownBasic1">
    <button ngbDropdownItem *ngFor="let action of actions"
           (click)="ChangeSelection(action)">
      {{action}}
    </button>
  </div>
</div>
```

Output:



Example4:

Dropdown with placement:

The preferred placement of the dropdown.

Possible values are "top", "top-left", "top-right", "bottom", "bottom-left", "bottom-right", "left", "left-top", "left-bottom", "right", "right-top", "right-bottom"

Accepts an array of strings or a string with space separated possible values.

The default order of preference is "bottom-left bottom-right top-left top-right"

Type: PlacementArray

Default value: - — initialized from NgbDropdownConfig service

NgbDropdownConfig – Configuration:

A configuration service for the **NgbDropdown** component.

You can inject this service, typically in your root component, and customize the values of its properties in order to provide default values for all the dropdowns used in the application.

Properties: autoClose, container, placement

app.component.html:

```
<div ngbDropdown placement="top-left" style="padding-left:5px;padding-top: 125px;">
  <button class="btn btn-outline-primary" id="dropdownBasic1" ngbDropdownToggle>
    Select Action
  </button>
  <div ngbDropdownMenu aria-labelledby="dropdownBasic1">
    <button ngbDropdownItem>Action - 1</button>
    <button ngbDropdownItem>Action - 2</button>
    <button ngbDropdownItem>Action - 3</button>
  </div>
</div>
```

Output:



Manual and custom triggers:

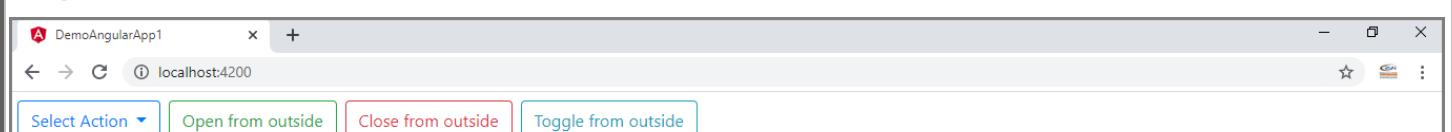
app.component.ts:

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

app.component.html:

```
<div ngbDropdown #myDrop="ngbDropdown" class="p-2">
  <button class="btn btn-outline-primary mr-2" id="dropdownManual"
    ngbDropdownAnchor (focus)="myDrop.open()">
    Select Action
  </button>
  <div ngbDropdownMenu aria-labelledby="dropdownManual">
    <button ngbDropdownItem>Action - 1</button>
    <button ngbDropdownItem>Action - 2</button>
    <button ngbDropdownItem>Action - 3</button>
  </div>
  <button class="btn btn-outline-success mr-2" (click)="event.stopPropagation(); myDrop.open()">
    Open from outside
  </button>
  <button class="btn btn-outline-danger mr-2" (click)="event.stopPropagation(); myDrop.close()">
    Close from outside
  </button>
  <button class="btn btn-outline-info mr-2" (click)="event.stopPropagation(); myDrop.toggle()">
    Toggle from outside
  </button>
</div>
```

Output:





NgbDropdownAnchor – Directive:

A directive to mark an element to which dropdown menu will be anchored.

This is a simple version of the NgbDropdownToggle directive. It plays the same role, but doesn't listen to click events to toggle dropdown menu thus enabling support for events other than click.

Selector: [ngbDropdownAnchor]

Methods of NgbDropdown:

open **open()** => void
Opens the dropdown menu.

close **close()** => void
Closes the dropdown menu.

toggle **toggle()** => void
Toggles the dropdown menu.

isOpen **isOpen()** => boolean
Checks if the dropdown menu is open.

\$event.stopPropagation():

Sometimes you have a click event on parent div as well as child elements of it. If you click on the child element, it automatically calls the parent click function along with its own function. This is called **event bubbling**. To restrict or stop this bubbling or propagation, we use stopPropagation method.

The **\$event.stopPropagation()** method stops the bubbling of an event to parent elements, preventing any parent event handlers from being executed.

Button groups and split buttons:

app.component.ts:

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

app.component.html:

```
<div class="btn-group p-2">
  <button type="button" class="btn btn-outline-success">Plain button</button>
  <div class="btn-group" ngbDropdown role="group"
    aria-label="Button group with nested dropdown">
    <button class="btn btn-outline-primary" ngbDropdownToggle>Drop me</button>
    <div class="dropdown-menu" ngbDropdownMenu>
      <button ngbDropdownItem>One</button>
      <button ngbDropdownItem>Two</button>
      <button ngbDropdownItem>Three!</button>
    </div>
  </div>
</div>
```

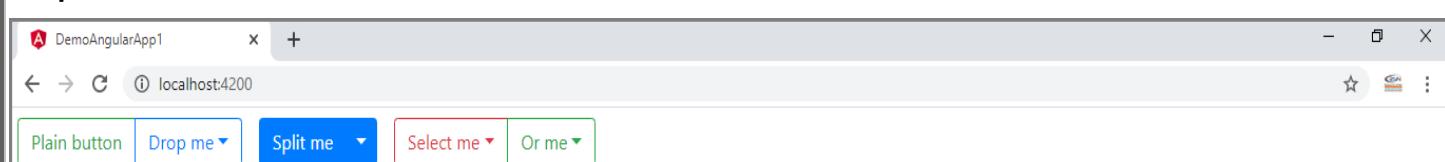
```

<div class="btn-group p-2">
  <button type="button" class="btn btn-primary">Split me</button>
  <div class="btn-group" ngbDropdown role="group"
    aria-label="Button group with nested dropdown">
    <button class="btn btn-primary dropdown-toggle-split" ngbDropdownToggle></button>
    <div class="dropdown-menu" ngbDropdownMenu>
      <button ngbDropdownItem>One</button>
      <button ngbDropdownItem>Two</button>
      <button ngbDropdownItem>Three!</button>
    </div>
  </div>
</div>

<div class="btn-group p-2">
  <div class="btn-group" ngbDropdown role="group"
    aria-label="Button group with nested dropdown">
    <button class="btn btn-outline-danger" ngbDropdownToggle>Select me</button>
    <div class="dropdown-menu" ngbDropdownMenu>
      <button ngbDropdownItem>One</button>
      <button ngbDropdownItem>Two</button>
      <button ngbDropdownItem>Three!</button>
    </div>
  </div>
  <div class="btn-group" ngbDropdown role="group"
    aria-label="Button group with nested dropdown">
    <button class="btn btn-outline-success" ngbDropdownToggle>Or me</button>
    <div class="dropdown-menu" ngbDropdownMenu>
      <button ngbDropdownItem>One</button>
      <button ngbDropdownItem>Two</button>
      <button ngbDropdownItem>Three!</button>
    </div>
  </div>
</div>

```

Output:



Mixed menu items and form:

app.component.ts:

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

app.component.html:

```
<div class="row p-2">
  <div class="col">
    <div ngbDropdown class="d-inline-block">
      <button class="btn btn-outline-primary" id="dropdownForm1" ngbDropdownToggle>
        Register Here
      </button>
      <div ngbDropdownMenu aria-labelledby="dropdownForm1">
        <form class="px-4 py-3">
          <div class="form-group">
            <label for="exampleDropdownFormEmail1">Email address</label>
            <input type="email" class="form-control"
                   id="exampleDropdownFormEmail1"
                   placeholder="email@example.com">
          </div>
          <div class="form-group">
            <label for="exampleDropdownFormPassword1">Password</label>
            <input type="password" class="form-control"
                   id="exampleDropdownFormPassword1"
                   placeholder="Password">
          </div>
          <div class="form-check">
            <input type="checkbox" class="form-check-input" id="dropdownCheck">
            <label class="form-check-label" for="dropdownCheck">
              Remember me
            </label>
          </div>
          <button type="submit" class="btn btn-primary">Sign in</button>
        </form>
        <div class="dropdown-divider"></div>
        <button ngbDropdownItem>New member here? Sign up</button>
        <button ngbDropdownItem>Forgot password?</button>
      </div>
    </div>
  </div>
</div>
```

Output:



A DemoAngularApp1

localhost:4200

Register Here ▾

Email address
email@example.com

Password

Remember me

Sign in

New member here? Sign up
Forgot password?

Dynamic positioning in a navbar:

app.component.ts:

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

app.component.html:

```
<nav class="navbar navbar-expand-md navbar-light bg-light">
  <span class="navbar-brand">Dropdowns in navbar</span>
  <button class="navbar-toggler" type="button" aria-controls="navbarContent"
    [attr.aria-expanded]="!collapsed" aria-label="Toggle navigation"
    (click)="collapsed=!collapsed">
    <span class="navbar-toggler-icon"></span>
  </button>

  <div class="navbar-collapse" [class.collapse]="collapsed" id="navbarContent">
    <ul class="navbar-nav ml-auto">
      <li class="nav-item" ngbDropdown>
        <a class="nav-link" style="cursor: pointer" ngbDropdownToggle
          id="navbarDropdown1" role="button">
          Static
        </a>
      </li>
    </ul>
  </div>
</nav>
```

```

<div ngbDropdownMenu aria-labelledby="navbarDropdown1" class="dropdown-menu">
    <a ngbDropdownItem href="#" (click)="event.preventDefault()">
        Action
    </a>
    <a ngbDropdownItem href="#" (click)="event.preventDefault()">
        Another action
    </a>
    <a ngbDropdownItem href="#" (click)="event.preventDefault()">
        Something else here
    </a>
</div>
</li>

<li class="nav-item ngbDropdown">
    <a class="nav-link" style="cursor: pointer" ngbDropdownToggle
       id="navbarDropdown2" role="button">
        Static right
    </a>
    <div ngbDropdownMenu aria-labelledby="navbarDropdown2"
         class="dropdown-menu dropdown-menu-right">
        <a ngbDropdownItem href="#" (click)="event.preventDefault()">
            Action
        </a>
        <a ngbDropdownItem href="#" (click)="event.preventDefault()">
            Another action
        </a>
        <a ngbDropdownItem href="#" (click)="event.preventDefault()">
            Something else here
        </a>
    </div>
</li>

<li class="nav-item ngbDropdown display="dynamic" placement="bottom-right">
    <a class="nav-link" style="cursor: pointer" ngbDropdownToggle
       id="navbarDropdown3" role="button">
        Dynamic
    </a>
    <div ngbDropdownMenu aria-labelledby="navbarDropdown3" class="dropdown-menu">
        <a ngbDropdownItem href="#" (click)="event.preventDefault()">
            Action
        </a>
        <a ngbDropdownItem href="#" (click)="event.preventDefault()">
            Another action
        </a>
        <a ngbDropdownItem href="#" (click)="event.preventDefault()">
            Something else here
        </a>
    </div>
</li>
</ul>
</div>
</nav>
```

Output:



display

Enable or disable the dynamic positioning. The default value is dynamic unless the dropdown is used inside a Bootstrap navbar. If you need custom placement for a dropdown in a navbar, set it to dynamic explicitly.

Type: "dynamic" | "static"

Global configuration of dropdowns

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbDropdownConfig } from '@ng-bootstrap/ng-bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [NgbDropdownConfig] // add NgbDropdownConfig to the component providers
})
export class AppComponent {
  constructor(config: NgbDropdownConfig) {
    // customize default values of dropdowns used by this component tree
    config.placement = 'bottom-left';
    config.autoClose = false;
  }
}
```

app.component.html:

```
<p class="p-2">This dropdown uses customized default values.</p>
<div ngbDropdown class="p-2">
  <button class="btn btn-outline-primary" id="dropdownConfig" ngbDropdownToggle>
    Toggle
  </button>
  <div ngbDropdownMenu aria-labelledby="dropdownConfig">
    <button ngbDropdownItem>Action - 1</button>
    <button ngbDropdownItem>Action - 2</button>
    <button ngbDropdownItem>Action - 3</button>
  </div>
</div>
```

Output:



autoClose

Indicates whether the dropdown should be closed when clicking one of dropdown items or pressing ESC.

true - the dropdown will close on both outside and inside (menu) clicks.

false - the dropdown can only be closed manually via **close()** or **toggle()** methods.

"**inside**" - the dropdown will close on inside menu clicks, but not outside clicks.

"**outside**" - the dropdown will close only on the outside clicks and not on menu clicks.

Type: boolean | "inside" | "outside"

Default value: true — initialized from NgbDropdownConfig service

8. Modal

Modal with default options:

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbModal, ModalDismissReasons } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  closeResult: string;

  constructor(private modalService: NgbModal) { }

  open(content) {
    this.modalService.open(content, { ariaLabelledBy: 'modal-basic-title' }).result.then((result) => {
      this.closeResult = `Closed with: ${result}`;
    }, (reason) => {
      this.closeResult = `Dismissed ${this.getDismissReason(reason)}`;
    });
  }

  private getDismissReason(reason: any): string {
    if (reason === ModalDismissReasons.ESC) {
      return 'by pressing ESC';
    } else if (reason === ModalDismissReasons.BACKDROP_CLICK) {
      return 'by clicking on a backdrop';
    } else {
      return `with: ${reason}`;
    }
  }
}
```

app.component.html:

```
<ng-template #content let-modal>
  <div class="modal-header">
    <h4 class="modal-title" id="modal-basic-title">Profile update</h4>
    <button type="button" class="close" aria-label="Close"
      (click)="modal.dismiss('Cross click')">
      <span aria-hidden="true">&times;</span>
    </button>
  </div>
```

```

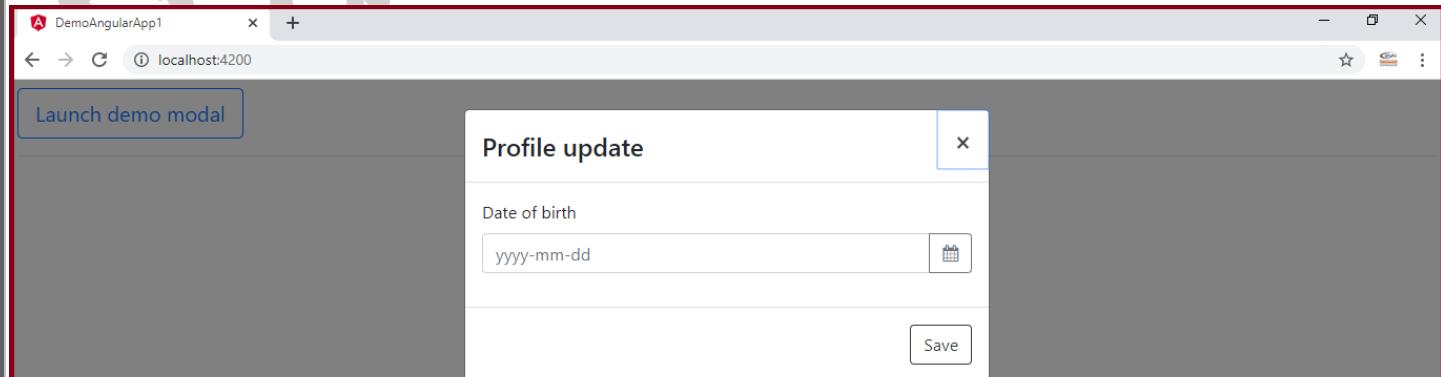
<div class="modal-body">
  <form>
    <div class="form-group">
      <label for="dateOfBirth">Date of birth</label>
      <div class="input-group">
        <input id="dateOfBirth" class="form-control" placeholder="yyyy-mm-dd"
               name="dp" ngbDatepicker #dp="ngbDatepicker">
        <div class="input-group-append">
          <button class="btn btn-outline-secondary fa fa-calendar"
                 (click)="dp.toggle()" type="button">
            </button>
        </div>
      </div>
    </div>
  </form>
</div>
<div class="modal-footer">
  <button type="button" class="btn btn-outline-dark"
         (click)="modal.close('Save click')">
    Save
  </button>
</div>
</ng-template>
<div class="p-2">
  <button class="btn btn-lg btn-outline-primary" (click)="open(content)">
    Launch demo modal
  </button>
  <hr>
  <pre>{{closeResult}}</pre>
</div>

```

Output:



After clicking the button:



Components as content:

You can pass an existing component as content of the modal window. In this case remember to add content component as an **entryComponents** section of your **NgModule**.

app.component.ts:

```
import { Component, Input } from '@angular/core';
import { NgbActiveModal, NgbModal } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'ngbd-modal-content',
  template: `
    <div class="modal-header">
      <h4 class="modal-title">Hi there!</h4>
      <button type="button" class="close" aria-label="Close"
        (click)="activeModal.dismiss('Cross click')">
        <span aria-hidden="true">&times;</span>
      </button>
    </div>
    <div class="modal-body">
      <p>Hello, {{name}}!</p>
    </div>
    <div class="modal-footer">
      <button type="button" class="btn btn-outline-dark"
        (click)="activeModal.close('Close click')">
        Close
      </button>
    </div>
  `
})

export class NgbdModalContent {
  @Input() name;

  constructor(public activeModal: NgbActiveModal) {}
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(private modalService: NgbModal) {}

  open() {
    const modalRef = this.modalService.open(NgbdModalContent);
    modalRef.componentInstance.name = 'World';
  }
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.component.html:

```
<div class="p-2">
  <button class="btn btn-lg btn-outline-primary" (click)="open()">
    Launch demo modal
  </button>
</div>
```

app.module.ts:

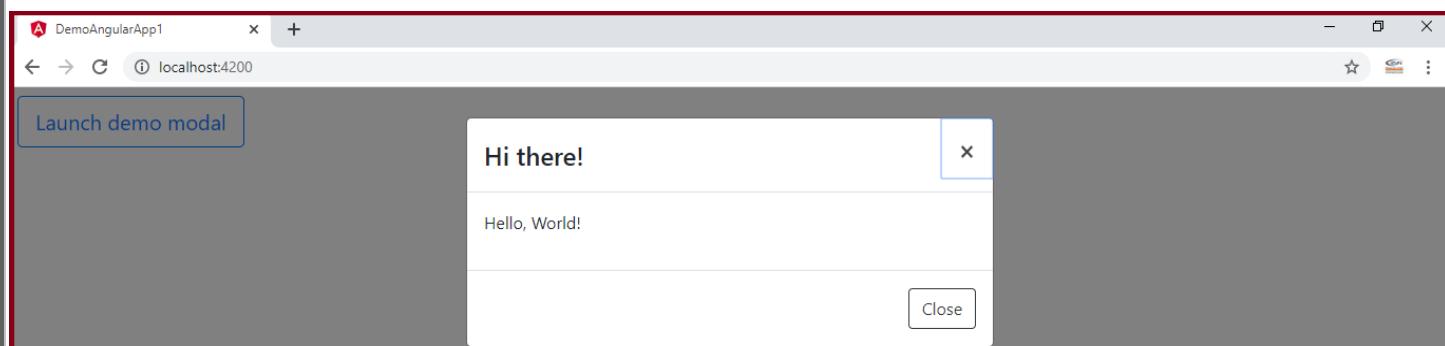
```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent, NgbModalContent } from './app.component';
import { NgbModule } from '@ng-bootstrap/ng-bootstrap';

@NgModule({
  declarations: [
    AppComponent, NgbModalContent
  ],
  imports: [
    BrowserModule, AppRoutingModule, NgbModule
  ],
  providers: [],
  bootstrap: [AppComponent],
  entryComponents:[NgbModalContent]
})
export class AppModule { }
```

Output:



After clicking the button:



Focus management:

First focusable element within the modal window will receive focus upon opening. This could be configured to focus any other element by adding an ngbAutofocus attribute on it.

```
<button type="button" ngbAutofocus class="btn btn-danger">
  (click)="modal.close('Ok click')>Ok</button>
```

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbActiveModal, NgbModal } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'ngbd-modal-confirm',
  template: `
    <div class="modal-header">
      <h4 class="modal-title" id="modal-title">Profile deletion</h4>
      <button type="button" class="close" aria-describedby="modal-title"
        (click)="modal.dismiss('Cross click')">
        <span aria-hidden="true">&times;</span>
      </button>
    </div>
    <div class="modal-body">
      <p>
        <strong>
          Are you sure you want to delete <span class="text-primary">"John Doe"</span> profile?
        </strong>
      </p>
      <p>
        All information associated to this user profile will be permanently deleted.
        <span class="text-danger">This operation can not be undone.</span>
      </p>
    </div>
    <div class="modal-footer">
      <button type="button" class="btn btn-outline-secondary"
        (click)="modal.dismiss('cancel click')">
        Cancel
      </button>
      <button type="button" class="btn btn-danger"
        (click)="modal.close('Ok click')">
        Ok
      </button>
    </div>
  `,
})

Copyright © 2019 https://www.facebook.com/groups/RakeshSoftNetAngular/ All Rights Reserved.
```

```

export class NgbModalConfirm {
  constructor(public modal: NgbActiveModal) { }
}

@Component({
  selector: 'ngbd-modal-confirm-autofocus',
  template: `
    <div class="modal-header">
      <h4 class="modal-title" id="modal-title">Profile deletion</h4>
      <button type="button" class="close" aria-label="Close button"
        aria-describedby="modal-title" (click)="modal.dismiss('Cross click')">
        <span aria-hidden="true">&times;</span>
      </button>
    </div>
    <div class="modal-body">
      <p>
        <strong>
          Are you sure you want to delete <span class="text-primary">"John Doe"</span> profile?
        </strong>
      </p>
      <p>
        All information associated to this user profile will be permanently deleted.
        <span class="text-danger">This operation can not be undone.</span>
      </p>
    </div>
    <div class="modal-footer">
      <button type="button" class="btn btn-outline-secondary"
        (click)="modal.dismiss('cancel click')">
        Cancel
      </button>
      <button type="button" ngbAutofocus class="btn btn-danger"
        (click)="modal.close('Ok click')">
        Ok
      </button>
    </div>
  `
})

export class NgbModalConfirmAutofocus {
  constructor(public modal: NgbActiveModal) { }
}

const MODALS = {
  focusFirst: NgbModalConfirm,
  autofocus: NgbModalConfirmAutofocus
};

```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

```

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(private _modalService: NgbModal) { }

  open(name: string) {
    this._modalService.open(MODALS[name]);
  }
}

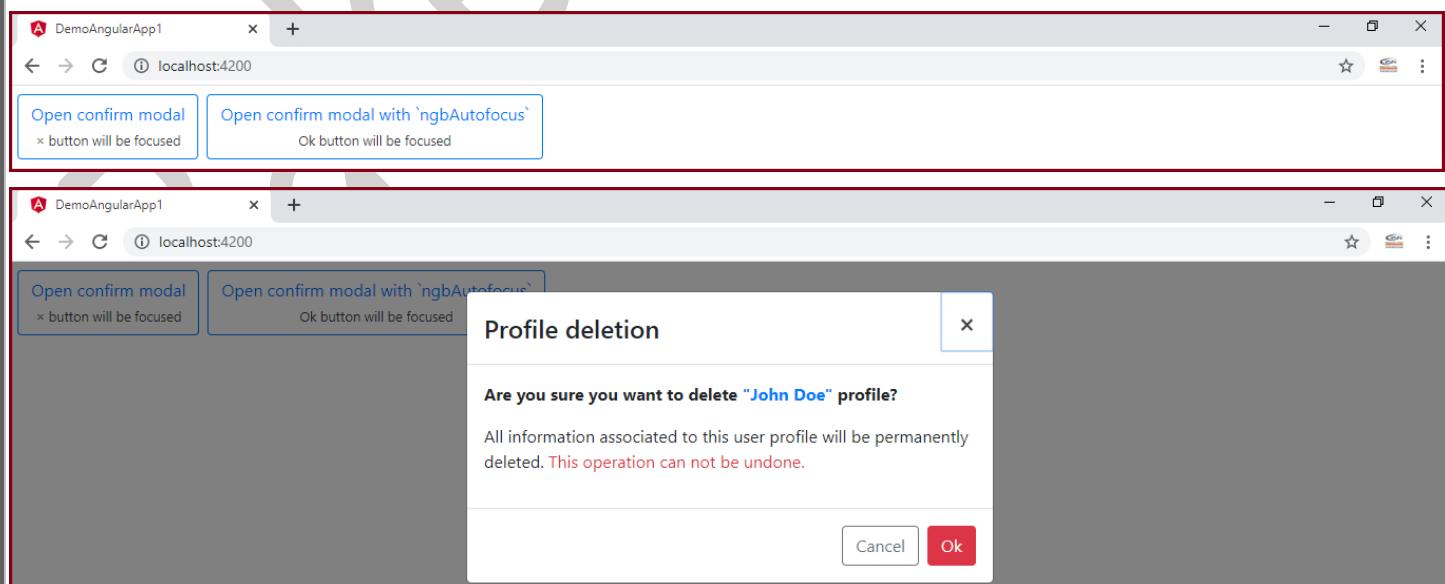
app.module.ts:

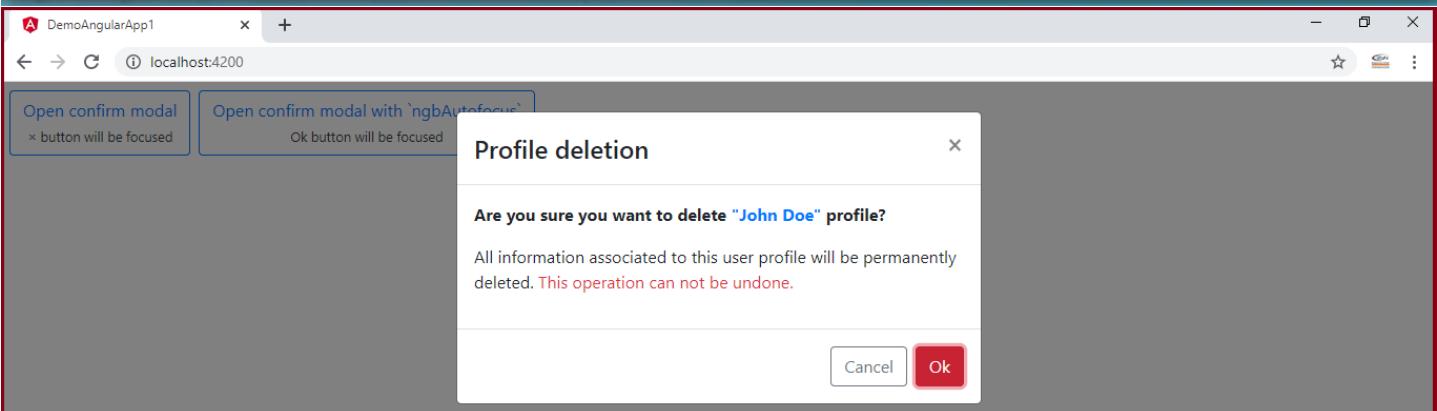
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent, NgbModalConfirm, NgbModalConfirmAutofocus } from './app.component';
import { NgbModule } from '@ng-bootstrap/ng-bootstrap';

@NgModule({
  declarations: [
    AppComponent, NgbModalConfirm, NgbModalConfirmAutofocus
  ],
  imports: [
    BrowserModule, AppRoutingModule, NgbModule
  ],
  providers: [],
  bootstrap: [AppComponent],
  entryComponents:[NgbModalConfirm, NgbModalConfirmAutofocus]
})
export class AppModule { }

```

Output:





Modal with options:

app.component.ts:

```
import { Component, ViewEncapsulation } from '@angular/core';
import { NgbModal } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  encapsulation: ViewEncapsulation.None,
  styles: [
    '.dark-modal .modal-content {
      background-color: #292b2c;
      color: white;
    }
    .dark-modal .close {
      color: white;
    }
    .light-blue-backdrop {
      background-color: #5cb3fd;
    }
  ]
})
export class AppComponent {
  closeResult: string;

  constructor(private modalService: NgbModal) {}

  openBackDropCustomClass(content) {
    this.modalService.open(content, {backdropClass: 'light-blue-backdrop'});
  }

  openWindowCustomClass(content) {
    this.modalService.open(content, { windowClass: 'dark-modal' });
  }
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



```

openSm(content) {
    this.modalService.open(content, { size: 'sm' });
}

openLg(content) {
    this.modalService.open(content, { size: 'lg' });
}

openXl(content) { this.modalService.open(content, {size: 'xl'}); }

openVerticallyCentered(content) {
    this.modalService.open(content, { centered: true });
}

openScrollableContent(longContent) {
    this.modalService.open(longContent, { scrollable: true });
}
}

```

app.component.html:

```

<ng-template #content let-modal>
    <div class="modal-header">
        <h4 class="modal-title">Modal title</h4>
        <button type="button" class="close" aria-label="Close"
            (click)="modal.dismiss('Cross click')">
            <span aria-hidden="true">&times;</span>
        </button>
    </div>
    <div class="modal-body">
        <p>This is a Modal Dialog&hellip;</p>
    </div>
    <div class="modal-footer">
        <button type="button" class="btn btn-light"
            (click)="modal.close('Close click')">
            Close
        </button>
    </div>
</ng-template>

<ng-template #longContent let-modal>
    <div class="modal-header">
        <h4 class="modal-title">Modal title</h4>
        <button type="button" class="close" aria-label="Close"
            (click)="modal.dismiss('Cross click')">
            <span aria-hidden="true">&times;</span>
        </button>
    </div>

```



```

<div class="modal-body">
  <p>This is a modal scrollable content</p>
  <p>This is a modal scrollable content</p>
</div>
<div class="modal-footer">
  <button type="button" class="btn btn-light"
    (click)="modal.close('Close click')">
    Close
  </button>
</div>
</div>
</ng-template>

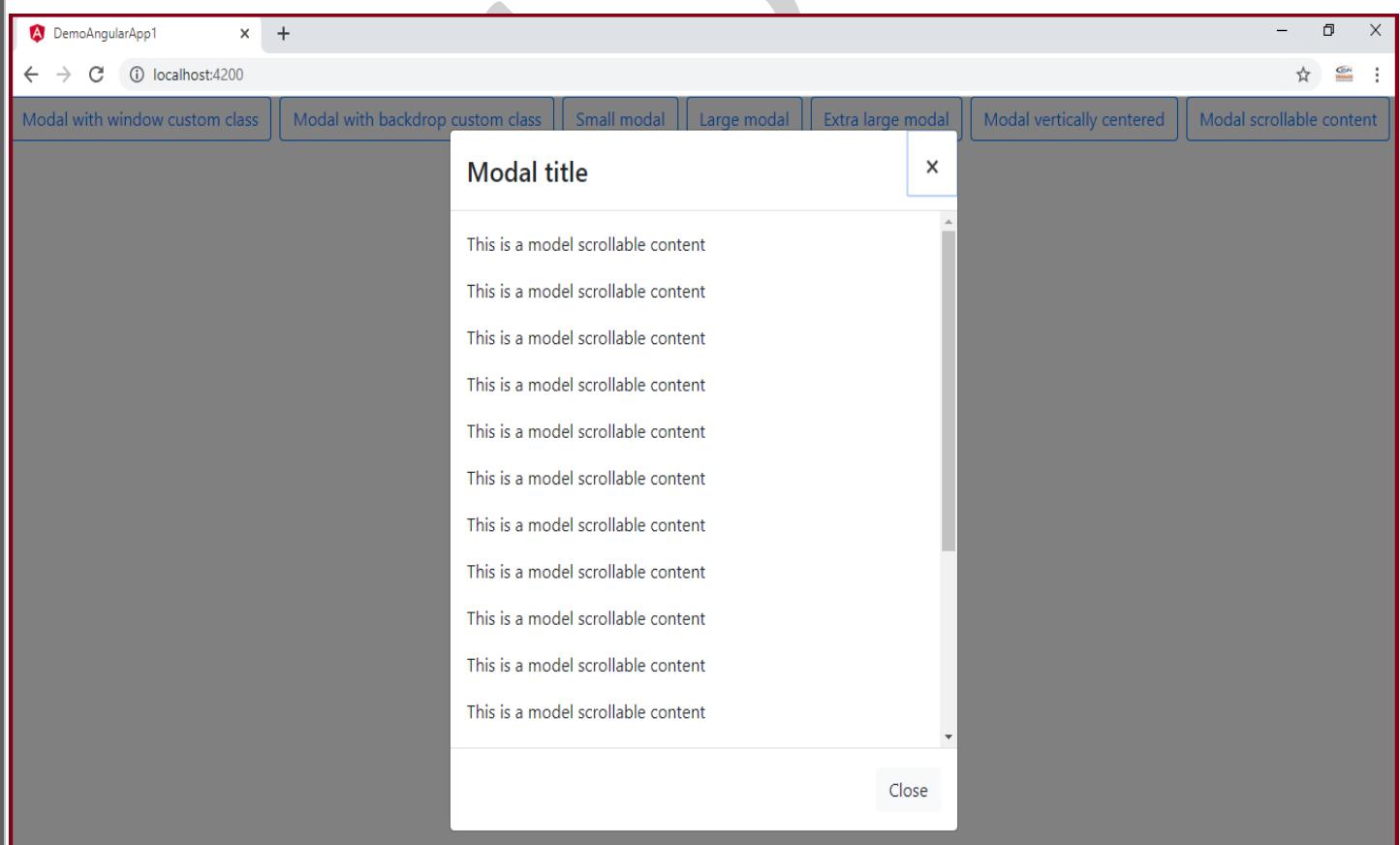
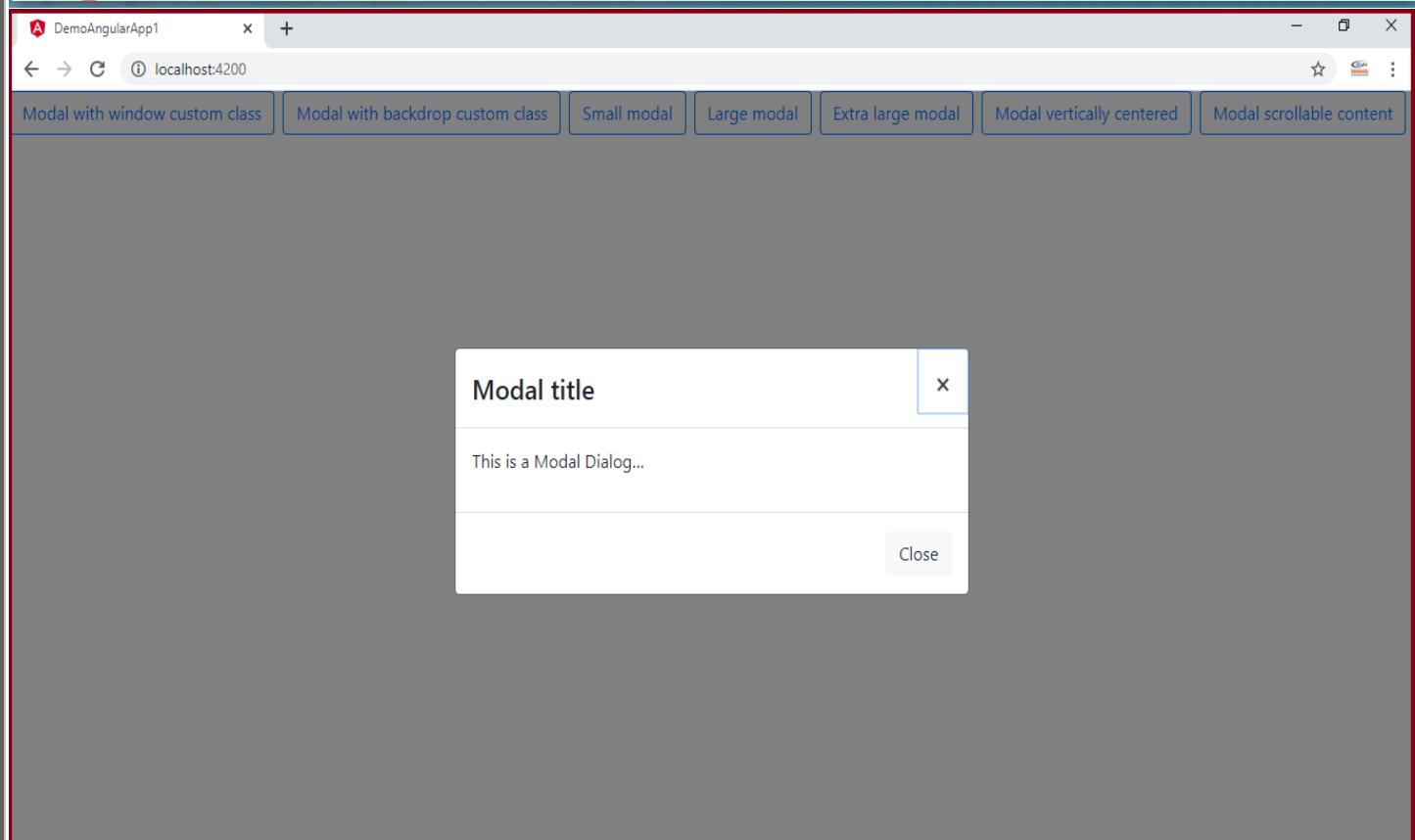
<button class="btn btn-outline-primary mb-2 mr-2"
  (click)="openWindowCustomClass(content)">
  Modal with window custom class
</button>
<button class="btn btn-outline-primary mb-2 mr-2"
  (click)="openBackDropCustomClass(content)">
  Modal with backdrop custom class
</button>
<button class="btn btn-outline-primary mb-2 mr-2"
  (click)="openSm(content)">
  Small modal
</button>
<button class="btn btn-outline-primary mb-2 mr-2"
  (click)="openLg(content)">
  Large modal
</button>
<button class="btn btn-outline-primary mb-2 mr-2"
  (click)="openXl(content)">
  Extra large modal
</button>
<button class="btn btn-outline-primary mb-2 mr-2"
  (click)="openVerticallyCentered(content)">
  Modal vertically centered
</button>
<button class="btn btn-outline-primary mb-2 mr-2"
  (click)="openScrollableContent(longContent)">
  Modal scrollable content
</button>

```

Output:

The screenshot displays five separate browser windows, each titled "DemoAngularApp1" and running on "localhost:4200". Each window shows a different type of modal dialog:

- Modal with window custom class:** A dark gray modal with a white header bar containing the title "Modal title" and a close button. The main content area contains the text "This is a Modal Dialog...".
- Modal with backdrop custom class:** A light gray modal with a white header bar containing the title "Modal title" and a close button. The main content area contains the text "This is a Modal Dialog...".
- Small modal:** A small, compact modal with a white header bar containing the title "Modal title" and a close button. The main content area contains the text "This is a Modal Dialog...".
- Large modal:** A large modal with a white header bar containing the title "Modal title" and a close button. The main content area contains the text "This is a Modal Dialog...".
- Extra large modal:** An extra large modal with a white header bar containing the title "Modal title" and a close button. The main content area contains the text "This is a Modal Dialog...".



Stacked modals:

app.component.ts:

```

import { Component } from '@angular/core';
import { NgbActiveModal, NgbModal } from '@ng-bootstrap/ng-bootstrap';

@Component({
  template: `
    <div class="modal-header">
      <h4 class="modal-title">Hi there!</h4>
      <button type="button" class="close" aria-label="Close"
        (click)="activeModal.dismiss('Cross click')">
        <span aria-hidden="true">&times;</span>
      </button>
    </div>
    <div class="modal-body">
      <p>Hello, World!</p>
      <p>
        <button class="btn btn-lg btn-outline-primary" (click)="open()">
          Launch demo modal
        </button>
      </p>
    </div>
    <div class="modal-footer">
      <button type="button" class="btn btn-outline-dark"
        (click)="activeModal.close('Close click')">
        Close
      </button>
    </div>
  `
})

export class NgbModal1Content {
  constructor(private modalService: NgbModal, public activeModal: NgbActiveModal) { }

  open() {
    this.modalService.open(NgbModal2Content, {
      size: 'lg'
    });
  }
}

```

```
@Component({
  template: `
    <div class="modal-header">
      <h4 class="modal-title">Hi there!</h4>
      <button type="button" class="close" aria-label="Close"
        (click)="activeModal.dismiss('Cross click')">
        <span aria-hidden="true">&times;</span>
      </button>
    </div>
    <div class="modal-body">
      <p>Hello, World!</p>
    </div>
    <div class="modal-footer">
      <button type="button" class="btn btn-outline-dark"
        (click)="activeModal.close('Close click')">
        Close
      </button>
    </div>
  `
})

export class NgbModal2Content {
  constructor(public activeModal: NgbActiveModal) { }
}
```

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  constructor(private modalService: NgbModal) { }

  open() {
    this.modalService.open(NgbModal1Content);
  }
}
```

app.component.html:

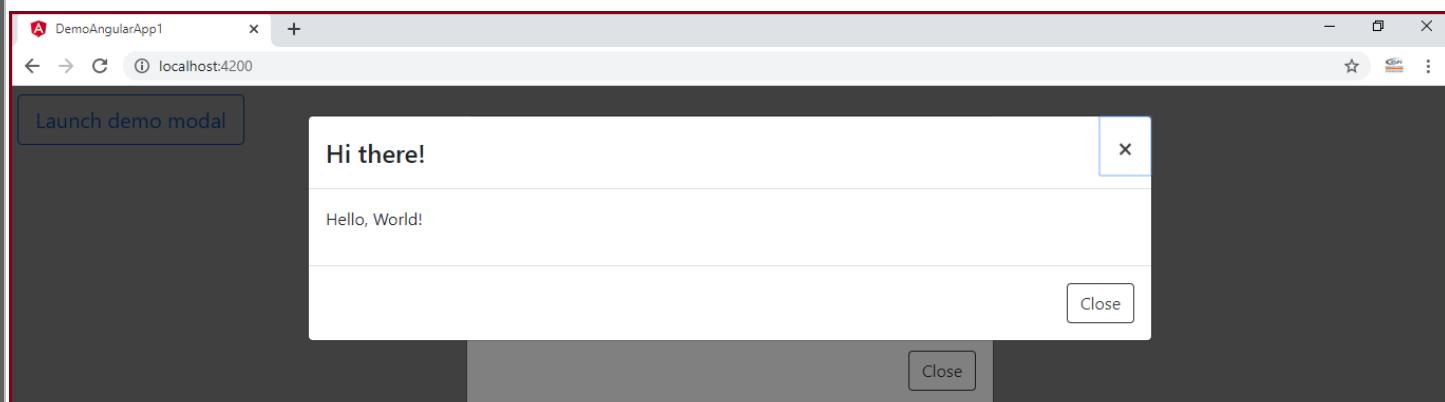
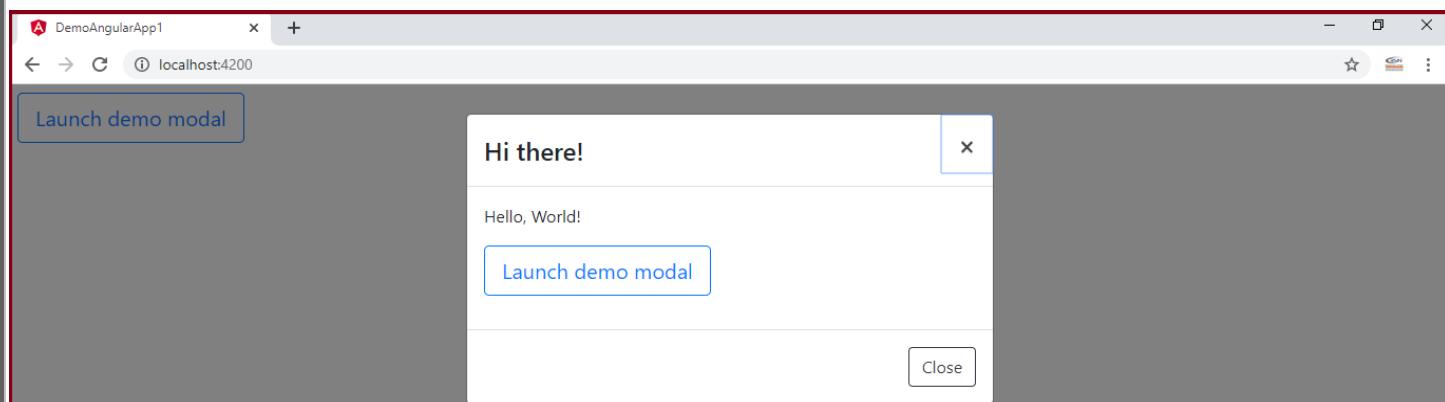
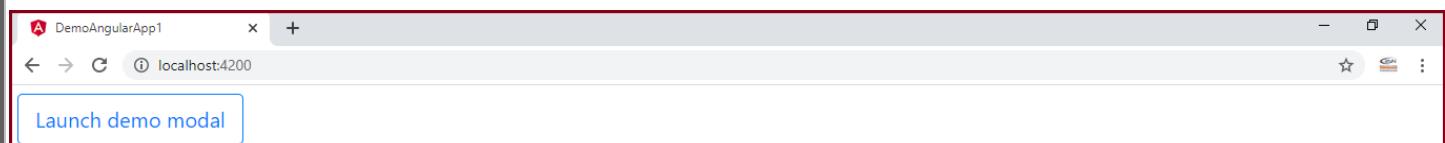
```
<div class="p-2">
  <button class="btn btn-lg btn-outline-primary" (click)="open()">
    Launch demo modal
  </button>
</div>
```

app.module.ts:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent, NgbModal1Content, NgbModal2Content } from './app.component';
import { NgbModule } from '@ng-bootstrap/ng-bootstrap';

@NgModule({
  declarations: [
    AppComponent, NgbModal1Content, NgbModal2Content
  ],
  imports: [
    BrowserModule, AppRoutingModule, NgbModule
  ],
  providers: [],
  bootstrap: [AppComponent],
  entryComponents: [NgbModal1Content, NgbModal2Content]
})
export class AppModule { }
```

Output:



Global configuration of modals:

app.component.ts:

```

import { Component } from '@angular/core';
import { NgbModalConfig, NgbModal } from '@ng-bootstrap/ng-bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  // add NgbModalConfig and NgbModal to the component providers
  providers: [NgbModalConfig, NgbModal]
})
export class AppComponent {
  constructor(config: NgbModalConfig, private modalService: NgbModal) {
    // customize default values of modals used by this component tree
    config.backdrop = 'static';
    config.keyboard = false;
  }
  open(content) {
    this.modalService.open(content);
  }
}

```

app.component.html:

```


<ng-template #content let-c="close" let-d="dismiss">
    <div class="modal-header">
      <h4 class="modal-title" id="modal-basic-title">Hi there!</h4>
      <button type="button" class="close" aria-label="Close"
        (click)="d('Cross click')">
        <span aria-hidden="true">&times;</span>
      </button>
    </div>
    <div class="modal-body">
      <p>Hello, World!</p>
    </div>
    <div class="modal-footer">
      <button type="button" class="btn btn-outline-dark"
        (click)="c('Save click')">
        Save
      </button>
    </div>
  </ng-template>
  <button class="btn btn-lg btn-outline-primary"
    (click)="open(content)">
    Launch demo modal
  </button>


```

9. Popover

NgbPopover – Directive:

A lightweight and extensible directive for fancy popover creation.

Selector: [ngbPopover]

Exported as: ngbPopover

Example1:

Quick and easy popovers:

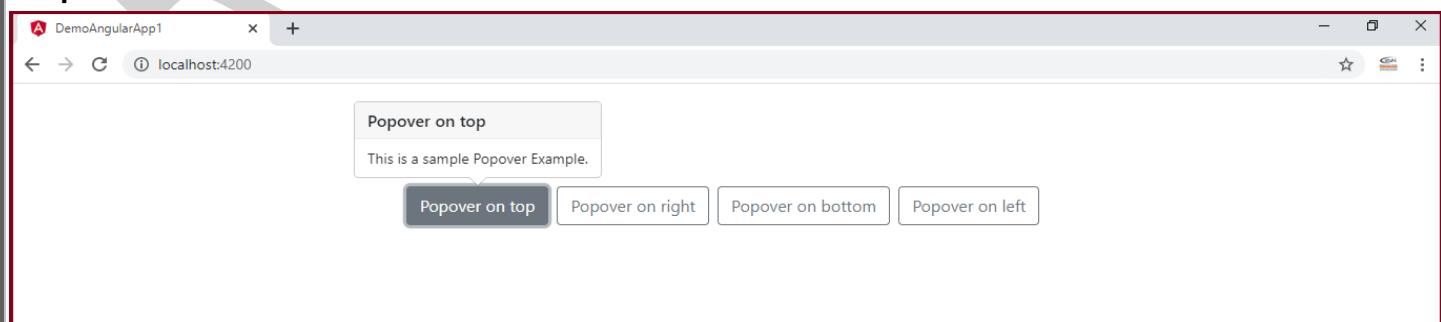
app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
```

app.component.html:

```
<div style="padding-top: 100px;text-align: center;">
  <button type="button" class="btn btn-outline-secondary mr-2" placement="top"
    ngbPopover="This is a sample Popover Example."
    popoverTitle="Popover on top">
    Popover on top
  </button>
  <button type="button" class="btn btn-outline-secondary mr-2" placement="right"
    ngbPopover="This is a sample Popover Example."
    popoverTitle="Popover on right">
    Popover on right
  </button>
  <button type="button" class="btn btn-outline-secondary mr-2" placement="bottom"
    ngbPopover="This is a sample Popover Example."
    popoverTitle="Popover on bottom">
    Popover on bottom
  </button>
  <button type="button" class="btn btn-outline-secondary mr-2" placement="left"
    ngbPopover="This is a sample Popover Example."
    popoverTitle="Popover on left">
    Popover on left
  </button>
</div>
```

Output:



Example2:

HTML and bindings in popovers:

Popovers can contain any arbitrary HTML, Angular bindings and even directives! Simply enclose desired content or title in a `<ng-template>` element.

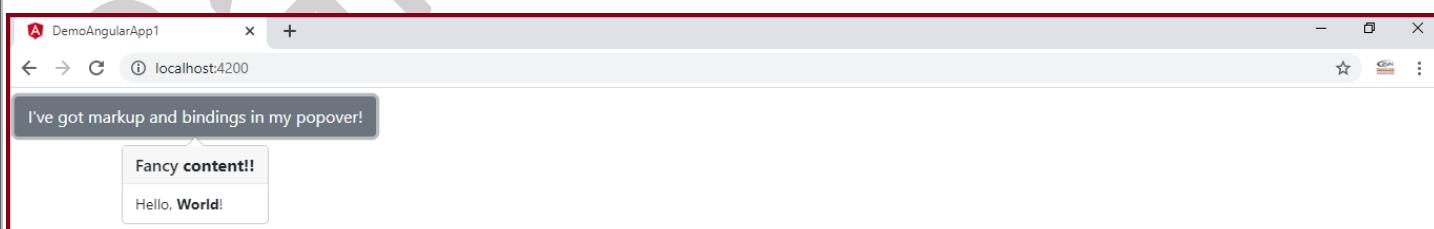
app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  name: string = 'World';
}
```

app.component.html:

```
<ng-template #popTitle>
  Fancy <b>content!!</b>
</ng-template>
<ng-template #popContent>
  Hello, <b>{{name}}</b>!
</ng-template>
<div class="p-2">
  <button type="button" class="btn btn-outline-secondary"
    [popoverTitle]="popTitle" [ngbPopover]="popContent">
    I've got markup and bindings in my popover!
  </button>
</div>
```

Output:



Example3:

Custom and manual triggers:

You can easily override open and close triggers by specifying event names (separated by :) in the `triggers` property.

Alternatively you can take full manual control over popover opening / closing events.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

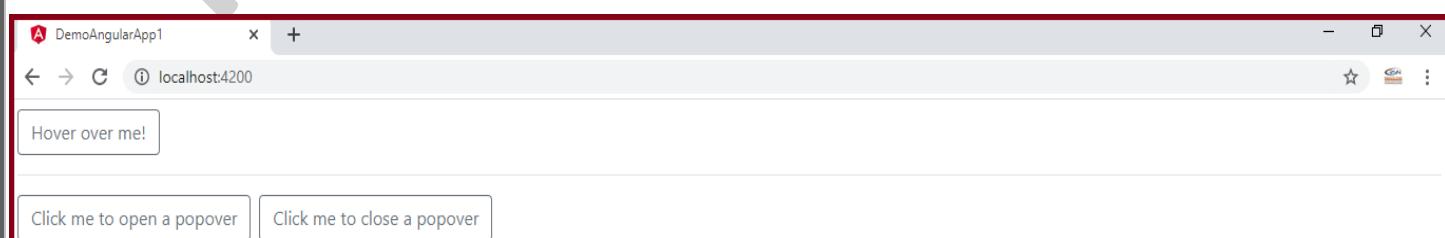
app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
```

app.component.html:

```
<div class="p-2">
  <button type="button" class="btn btn-outline-secondary"
    ngbPopover="You see, I show up on hover!"
    triggers="mouseenter:mouseleave"
    popoverTitle="Pop title">
    Hover over me!
  </button>
  <hr>
  <button type="button" class="btn btn-outline-secondary mr-2"
    ngbPopover="What a great tip!"
    [autoClose]= "false"
    triggers="manual"
    #p="ngbPopover"
    (click)="p.open()"
    popoverTitle="Pop title">
    Click me to open a popover
  </button>
  <button type="button" class="btn btn-outline-secondary mr-2"
    (click)="p.close()">
    Click me to close a popover
  </button>
</div>
```

Output:





Example4:

Automatic closing with keyboard and mouse:

As for some other popup-based widgets, you can set the popover to close automatically upon some events.

In the following examples, they will all close on **Escape** as well as:

- click inside: Click to toggle
- click outside: Click to toggle
- all clicks: Click to toggle

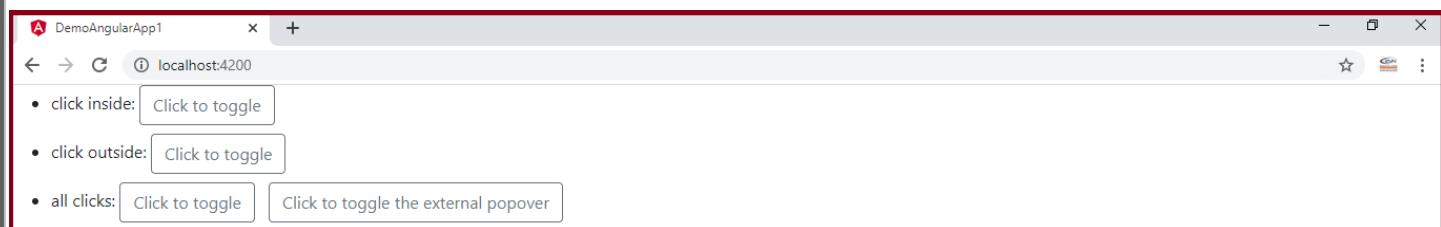
app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
```

app.component.html:

```
<ul>
  <li class="mb-2">
    click inside:
    <button type="button" class="btn btn-outline-secondary"
      popoverTitle="Pop title"
      [autoClose]="'inside'"
      ngbPopover="Click inside or press Escape to close">
      Click to toggle
    </button>
  </li>
  <li class="mb-2">
    click outside:
    <button type="button" class="btn btn-outline-secondary"
      popoverTitle="Pop title"
      [autoClose]="'outside'"
      ngbPopover="Click outside or press Escape to close">
      Click to toggle
    </button>
  </li>
  <li class="mb-2">
    all clicks:
    <button type="button" class="btn btn-outline-secondary"
      popoverTitle="Pop title"
      [autoClose]="true"
      ngbPopover="Click anywhere or press Escape to close (try the toggling element too)"
      #popover3="ngbPopover">
      Click to toggle
    </button>
    &nbsp;
    <button type="button" class="btn btn-outline-secondary mr-2"
      (click)="popover3.toggle()">
      Click to toggle the external popover
    </button>
  </li>
</ul>
```

Output:



Example5:

Open and close delays:

When using non-manual triggers, you can control the delay to open and close the popover through the `openDelay` and `closeDelay` properties. Note that the `autoClose` feature does not use the close delay, it closes the popover immediately.

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
```

app.component.html:

```
<div class="p-2">
  <button type="button" class="btn btn-outline-secondary mr-2"
    ngbPopover="You see, I show up after 300ms and disappear after 500ms!"
    [openDelay]="300"
    [closeDelay]="500"
    triggers="mouseenter:mouseleave">
    Hover 300ms here
  </button>

  <button type="button" class="btn btn-outline-secondary mr-2"
    ngbPopover="You see, I show up after 1s and disappear after 2s!"
    [openDelay]="1000"
    [closeDelay]="2000"
    triggers="mouseenter:mouseleave">
    Hover 1s here
  </button>
</div>
```

Output:



Example 6:

Popover visibility events:

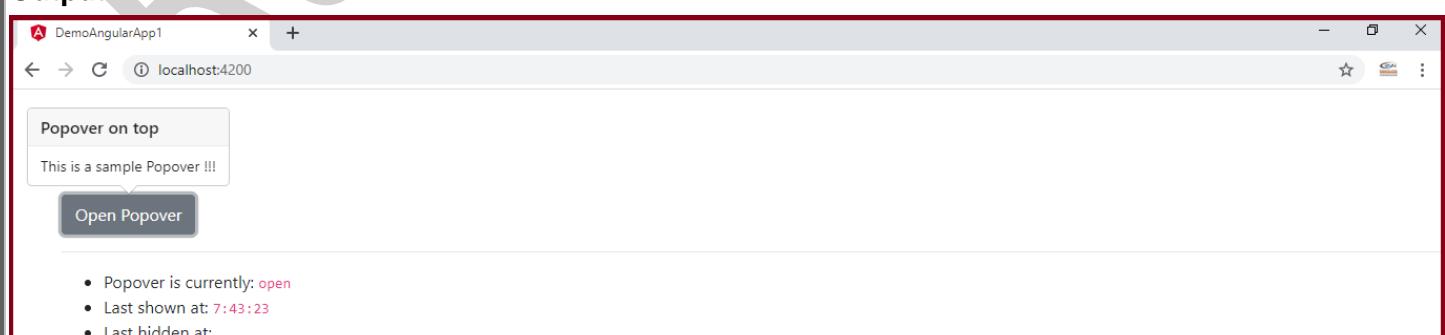
app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  lastShown: Date;
  lastHidden: Date;
  recordShown() {
    this.lastShown = new Date();
  }
  recordHidden() {
    this.lastHidden = new Date();
  }
}
```

app.component.html:

```
<div style="padding-left:50px;padding-top: 100px;">
  <button type="button" class="btn btn-outline-secondary"
    placement="top"
    ngbPopover="This is a sample Popover !!!"
    popoverTitle="Popover on top"
    #popover="ngbPopover"
    (shown)="recordShown()"
    (hidden)="recordHidden()">
    Open Popover
  </button>
  <hr>
  <ul>
    <li>Popover is currently:
      <code>{{ popover.isOpen() ? 'open' : 'closed' }}</code>
    </li>
    <li>Last shown at: <code>{{lastShown | date:'h:mm:ss'}}</code></li>
    <li>Last hidden at: <code>{{lastHidden | date:'h:mm:ss'}}</code></li>
  </ul>
</div>
```

Output:



DemoAngularApp1

localhost:4200

Popover on top

This is a sample Popover !!!

Open Popover

- Popover is currently: open
- Last shown at: 7:43:23
- Last hidden at:

Example 7:

Popover with custom class:

You can optionally pass in a custom class via `popoverClass`

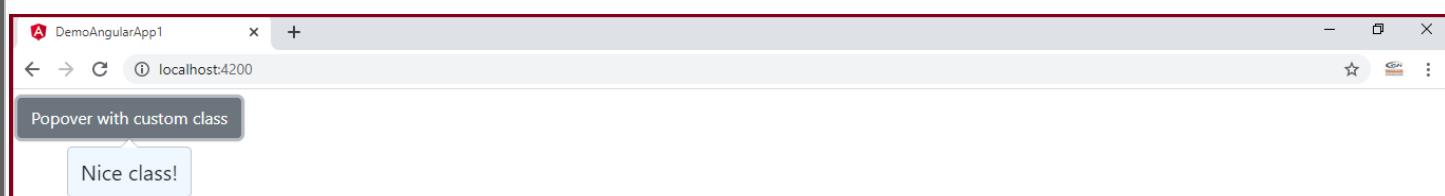
app.component.ts:

```
import { Component, ViewEncapsulation } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  encapsulation: ViewEncapsulation.None,
  styles: [
    .my-custom-class {
      background: aliceblue;
      font-size: 125%;
    }
    .my-custom-class .arrow::after {
      border-top-color: aliceblue;
    }
  ]
})
export class AppComponent {
```

app.component.html:

```
<div class="p-2">
  <button type="button"
    class="btn btn-outline-secondary"
    ngbPopover="Nice class!"
    popoverClass="my-custom-class">
    Popover with custom class
  </button>
</div>
```

Output:



Example 8:

Global configuration of popovers:

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbPopoverConfig } from '@ng-bootstrap/ng-bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  providers: [NgbPopoverConfig] // add NgbPopoverConfig to the component providers
})
export class AppComponent {
  constructor(config: NgbPopoverConfig) {
    // customize default values of popovers used by this component tree
    config.placement = 'right';
    config.triggers = 'hover';
  }
}
```

app.component.html:

```
<div style="padding-left:10px;padding-top:50px;">
  <button type="button"
    class="btn btn-outline-secondary"
    ngbPopover="This popover gets its inputs from the customized configuration"
    popoverTitle="Customized popover">
    Customized popover
  </button>
</div>
```

Output:



10. Nav

Since 5.2.0

Nav includes **NgbNav**, **NgbNavItem**, **NgbNavLink**, **NgbNavContent** directives and the **NgbNavOutlet** component.

These directives are fully based on the bootstrap markup leaving all DOM nodes available for you. They just handle nav selection, accessibility and basic styling for you. You can always add additional classes and behavior on top if necessary.

Nav directives that helps with implementing tabbed navigation components. They're meant to replace existing **Tabset** as a more flexible alternative.

NgbNav – Directive:

Selector: [ngbNav]

Exported as: ngbNav

NgbNavContent – Directive:

This directive must be used to wrap content to be displayed in the nav.

Selector: ng-template[ngbNavContent]

NgbNavItem – Directive:

The directive used to group nav link and related nav content. As well as set nav identifier and some options.

Selector: [ngbNavItem]

Exported as: ngbNavItem

NgbNavLink – Directive:

A directive to put on the nav link.

Selector: a[ngbNavLink]

NgbNavOutlet – Component:

The outlet where currently active nav content will be displayed.

Selector: [ngbNavOutlet]

Example1:

Basic navs:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  active = 1;
}
```



app.component.html:

```
<div class="m-2">
<ul ngbNav #nav="ngbNav" [(activeId)]="active" class="nav-tabs">
  <li [ngbNavItem]="1">
    <a ngbNavLink>One</a>
    <ng-template ngbNavContent>
      <p>Content One</p>
    </ng-template>
  </li>
  <li [ngbNavItem]="2">
    <a ngbNavLink>Two</a>
    <ng-template ngbNavContent>
      <p>Content Two</p>
    </ng-template>
  </li>
  <li [ngbNavItem]="3">
    <a ngbNavLink>Three</a>
    <ng-template ngbNavContent>
      <p>Content Three</p>
    </ng-template>
  </li>
</ul>
<div [ngbNavOutlet]="nav" class="mt-2"></div>
<pre>Active: {{ active }}</pre>
</div>
```

Output:



Example 2: Alternative markup:

You can use alternative markup without **ul li** elements:

app.component.ts:

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

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.component.html:

```
<div class="m-2">
  <nav ngbNav #nav="ngbNav" class="nav-tabs">
    <ng-container ngbNavItem>
      <a ngbNavLink>One</a>
      <ng-template ngbNavContent>
        <p>Content One</p>
      </ng-template>
    </ng-container>
    <ng-container ngbNavItem>
      <a ngbNavLink>Two</a>
      <ng-template ngbNavContent>
        <p>Content Two</p>
      </ng-template>
    </ng-container>
    <ng-container ngbNavItem>
      <a ngbNavLink>Three</a>
      <ng-template ngbNavContent>
        <p>Content Three</p>
      </ng-template>
    </ng-container>
  </nav>
  <div [ngbNavOutlet]="nav" class="mt-2"></div>
</div>
```

Output:



Selecting navs:

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbNavChangeEvent } from '@ng-bootstrap/ng-bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  active;
  disabled = true;
```



```

onNavChange(changeEvent: NgbNavChangeEvent) {
  if (changeEvent.nextId === 3) {
    changeEvent.preventDefault();
  }
}
toggleDisabled() {
  this.disabled = !this.disabled;
  if (this.disabled) {
    this.active = 1;
  }
}

```

app.component.html:

```

<div class="m-2">
  <ul ngbNav #nav="ngbNav" [(activeId)]="active"
    (navChange)="onNavChange($event)" class="nav-tabs">
    <li [ngbNavItem]="1">
      <a ngbNavLink>One</a>
      <ng-template ngbNavContent>
        <p>Content One</p>
      </ng-template>
    </li>
    <li [ngbNavItem]="2">
      <a ngbNavLink>Two</a>
      <ng-template ngbNavContent>
        <p>Content Two</p>
      </ng-template>
    </li>
    <li [ngbNavItem]="3">
      <a ngbNavLink>I can't be selected on click</a>
      <ng-template ngbNavContent>
        <p>Content Three</p>
      </ng-template>
    </li>
    <li [ngbNavItem]="4" [disabled]="disabled">
      <a ngbNavLink>I'm disabled</a>
      <ng-template ngbNavContent>
        <p>Content Four</p>
      </ng-template>
    </li>
  </ul>
  <div [ngbNavOutlet]="nav" class="mt-2"></div>
  <div class="mb-3">
    <button class="btn btn-sm btn-outline-primary"
      [disabled]="active === 2"
      (click)="nav.select(2)">
      Select second tab
    </button>
    <button class="btn btn-sm btn-outline-primary ml-2"
      (click)="toggleDisabled()">
      Toggle last disabled
    </button>
  </div>
  <pre>Active: {{ active }}</pre>
</div>

```

Output:



Example 3: Keep content:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  active = 1;
}
```

app.component.html:

```
<div class="m-2">
  <ul ngbNav #nav="ngbNav" [(activeId)]="active" [destroyOnHide]="false" class="nav-tabs">
    <li [ngbNavItem]="1" [destroyOnHide]="true">
      <a ngbNavLink>One</a>
      <ng-template ngbNavContent>
        <ngb-alert [dismissible]="false" class="d-block mt-3" type="danger">
          This tab content DOM will be destroyed when not active
        </ngb-alert>
        <p>Content One</p>
      </ng-template>
    </li>
    <li [ngbNavItem]="2">
      <a ngbNavLink>Two</a>
      <ng-template ngbNavContent>
        <ngb-alert [dismissible]="false" class="d-block mt-3" type="success">
          This tab content DOM will always stay in DOM
        </ngb-alert>
        <p>Content Two</p>
      </ng-template>
    </li>
    <li [ngbNavItem]="3">
      <a ngbNavLink>Three</a>
      <ng-template ngbNavContent let-active>
        <ngb-alert *ngIf="active" [dismissible]="false" class="d-block mt-3">
          While tab content DOM is never destroyed,
          this alert exists only when current tab is active
        </ngb-alert>
        <p>Content Three</p>
      </ng-template>
    </li>
  </ul>
  <div [ngbNavOutlet]="nav" class="mt-2"></div>
  <pre>Active: {{ active }}</pre>
</div>
```

Output:



Example 4: Dynamic navs:

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: [
    `
      .close {
        font-size: 1.4rem;
        opacity: 0.1;
        transition: opacity 0.3s;
      }
      .nav-link:hover > .close {
        opacity: 0.8;
      }
    `
  ]
})

export class AppComponent {
  tabs = [1, 2, 3, 4, 5];
  counter = this.tabs.length + 1;
  active;

  close(event: MouseEvent, toRemove: number) {
    this.tabs = this.tabs.filter(id => id !== toRemove);
    event.preventDefault();
    event.stopImmediatePropagation();
  }

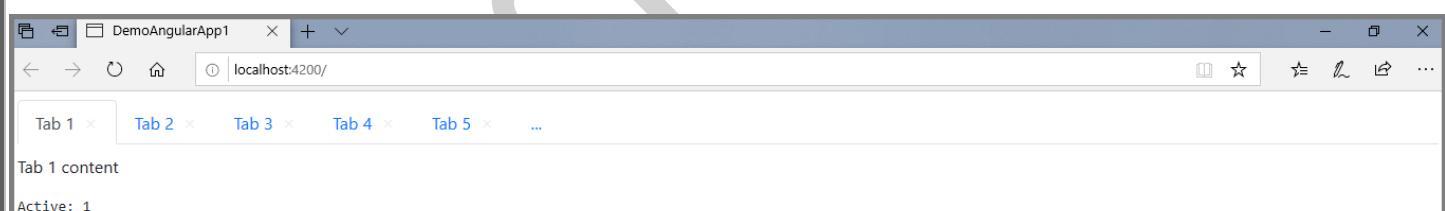
  add(event: MouseEvent) {
    this.tabs.push(this.counter++);
    event.preventDefault();
  }
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.component.html:

```
<div class="mt-2">
  <ul ngbNav #nav="ngbNav" [(activeId)]="active" class="nav-tabs">
    <li *ngFor="let id of tabs" [ngbNavItem]="id">
      <a ngbNavLink>
        Tab {{ id }}
        <span class="close position-relative pl-2 font-weight-light"
              (click)="close($event, id)">
          ×
        </span>
      </a>
      <ng-template ngbNavContent>
        <p>Tab {{ id }} content</p>
      </ng-template>
    </li>
    <li class="nav-item">
      <a class="nav-link" href (click)="add($event)">...</a>
    </li>
  </ul>
  <div [ngbNavOutlet]="nav" class="mt-2"></div>
  <pre>Active: {{ active }}</pre>
</div>
```

Output:



Example 5: Custom style:

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.component.html:

```
<div class="m-2">
  <ul ngbNav #nav="ngbNav" [activeId]="2" class="nav-tabs justify-content-center">
    <li [ngbNavItem]="1">
      <a ngbNavLink>Simple</a>
      <ng-template ngbNavContent>
        <p>Simple Content</p>
      </ng-template>
    </li>
    <li [ngbNavItem]="2">
      <a ngbNavLink><b>Fancy</b> title</a>
      <ng-template ngbNavContent>
        <p>Fancy Content</p>
      </ng-template>
    </li>
    <li ngbDropdown class="nav-item">
      <a href (click)="false" class="nav-link" ngbDropdownToggle>Dropdown</a>
      <div ngbDropdownMenu>
        <button ngbDropdownItem>Action 1</button>
        <button ngbDropdownItem>Action 2</button>
        <button ngbDropdownItem>Action 3</button>
        <div class="dropdown-divider"></div>
        <button ngbDropdownItem>Other</button>
      </div>
    </li>
    <li ngbNavItem [disabled]="true" class="ml-auto">
      <a ngbNavLink>Disabled</a>
      <ng-template ngbNavContent>
        <p>Disabled Content</p>
      </ng-template>
    </li>
  </ul>
  <div [ngbNavOutlet]="nav" class="mt-2"></div>
</div>
```

Output:



Example 6: Global configuration of navs:

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbNavConfig } from '@ng-bootstrap/ng-bootstrap';

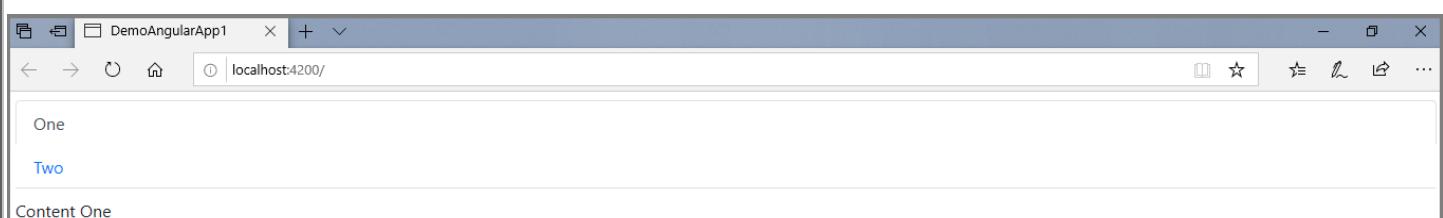
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [NgbNavConfig] // add NgbNavConfig to the component providers
})

export class AppComponent {
  constructor(config: NgbNavConfig) {
    config.destroyOnHide = false;
    config.orientation = "vertical";
  }
}
```

app.component.html

```
<div class="m-2">
  <ul ngbNav #nav="ngbNav" class="nav-tabs">
    <li ngbNavItem>
      <a ngbNavLink>One</a>
      <ng-template ngbNavContent>
        <p>Content One</p>
      </ng-template>
    </li>
    <li ngbNavItem>
      <a ngbNavLink>Two</a>
      <ng-template ngbNavContent>
        <p>Content Two</p>
      </ng-template>
    </li>
  </ul>
  <div [ngbNavOutlet]="nav" class="mt-2"></div>
</div>
```

Output:





11. Pagination

Pagination is a component that only displays page numbers. It will not manipulate your data collection. You will have to split your data collection into pages yourself.

NgbPagination – Component:

A component that displays page numbers and allows to customize them in several ways.

Selector: ngb-pagination

NgbPaginationEllipsis – Directive:

Since 4.1.0

A directive to match the 'ellipsis' link template

Selector: ng-template[ngbPaginationEllipsis]

NgbPaginationFirst – Directive:

Since 4.1.0

A directive to match the 'first' link template

Selector: ng-template[ngbPaginationFirst]

NgbPaginationLast – Directive:

Since 4.1.0

A directive to match the 'last' link template

Selector: ng-template[ngbPaginationLast]

NgbPaginationNext – Directive:

Since 4.1.0

A directive to match the 'next' link template

Selector: ng-template[ngbPaginationNext]

NgbPaginationPrevious – Directive:

Since 4.1.0

A directive to match the 'previous' link template

Selector: ng-template[ngbPaginationPrevious]

NgbPaginationNumber – Directive:

Since 4.1.0

A directive to match the 'number' link template

Selector: ng-template[ngbPaginationNumber]

NgbPaginationLinkContext – Interface:

Since 4.1.0

A context for the

- NgbPaginationFirst
- NgbPaginationPrevious
- NgbPaginationNext
- NgbPaginationLast
- NgbPaginationEllipsis

link templates in case you want to override one.

NgbPaginationConfig – Configuration:

A configuration service for the **NgbPagination** component.

You can inject this service, typically in your root component, and customize the values of its properties in order to provide default values for all the paginations used in the application.

Properties:

boundaryLinks **directionLinks** **disabled** **ellipses** **maxSize** **pageSize** **rotate** **size**

Example 1: Basic pagination:

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  page = 4;
}
```

app.component.html:

```
<div class="p-2">
  <p>Default pagination:</p>
  <ngb-pagination [collectionSize]="70" [(page)]="page" aria-label="Default pagination"></ngb-pagination>

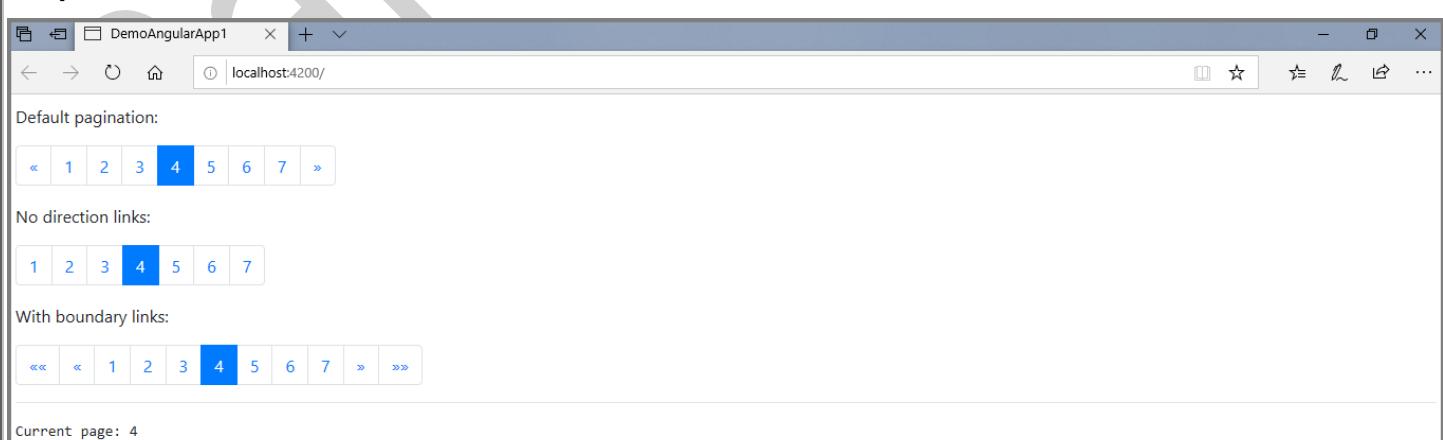
  <p>No direction links:</p>
  <ngb-pagination [collectionSize]="70" [(page)]="page" [directionLinks]="false"></ngb-pagination>

  <p>With boundary links:</p>
  <ngb-pagination [collectionSize]="70" [(page)]="page" [boundaryLinks]="true"></ngb-pagination>

  <hr />

  <pre>Current page: {{page}}</pre>
</div>
```

Output:



The screenshot shows a browser window with four examples of Angular pagination components:

- Default pagination:** A simple numeric range from 1 to 7, with page 4 highlighted.
- No direction links:** A numeric range from 1 to 7, with page 4 highlighted.
- With boundary links:** An extended numeric range from 1 to 7, flanked by '<<' and '>>>' links.
- Current page:** A text field showing 'Current page: 4'.

Example 2: Advanced pagination

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  page = 1;
}
```

app.component.html:

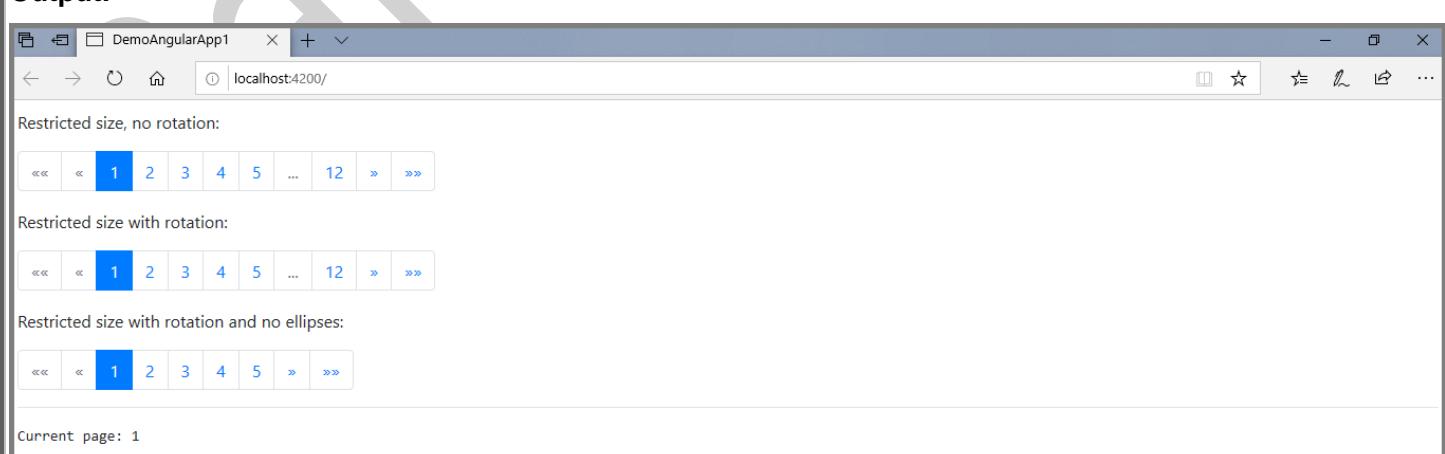
```
<div class="p-2">
  <p>Restricted size, no rotation:</p>
  <ngb-pagination [collectionSize]="120" [(page)]="page" [maxSize]="5"
    [boundaryLinks]="true">
  </ngb-pagination>

  <p>Restricted size with rotation:</p>
  <ngb-pagination [collectionSize]="120" [(page)]="page" [maxSize]="5" [rotate]="true"
    [boundaryLinks]="true">
  </ngb-pagination>

  <p>Restricted size with rotation and no ellipses:</p>
  <ngb-pagination [collectionSize]="120" [(page)]="page" [maxSize]="5" [rotate]="true"
    [ellipses]="false" [boundaryLinks]="true">
  </ngb-pagination>

  <hr>
  <pre>Current page: {{page}}</pre>
</div>
```

Output:



DemoAngularApp1

localhost:4200/

Restricted size, no rotation:

« « 1 2 3 4 5 ... 12 » »»

Restricted size with rotation:

« « 1 2 3 4 5 ... 12 » »»

Restricted size with rotation and no ellipses:

« « 1 2 3 4 5 » »»

Current page: 1

Example 3: Custom links:

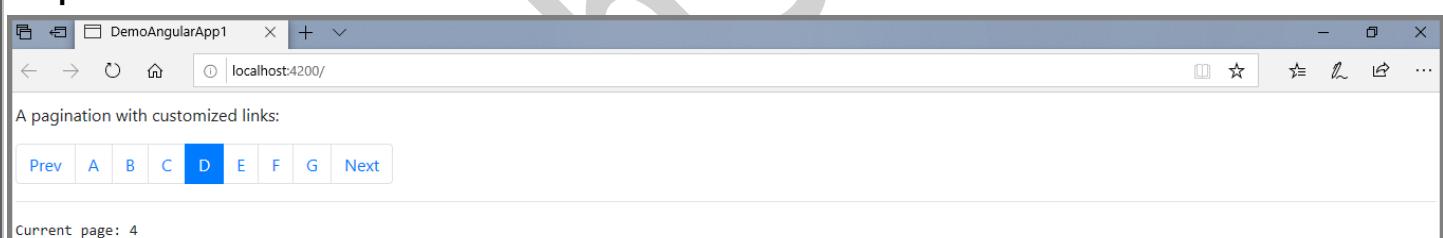
app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  page = 4;
  getPageSymbol(current: number) {
    return ['A', 'B', 'C', 'D', 'E', 'F', 'G'][current - 1];
  }
}
```

app.component.html:

```
<div class="p-2">
  <p>A pagination with customized links:</p>
  <ngb-pagination [collectionSize]="70" [(page)]="page" aria-label="Custom pagination">
    <ng-template ngbPaginationPrevious>Prev</ng-template>
    <ng-template ngbPaginationNext>Next</ng-template>
    <ng-template ngbPaginationNumber let-p>{{ getPageSymbol(p) }}</ng-template>
  </ngb-pagination>
  <hr>
  <pre>Current page: {{page}}</pre>
</div>
```

Output:



Example 4: Pagination size:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  currentPage = 3;
}
```

app.component.html:

```
<ngb-pagination [collectionSize]="50" [(page)]="currentPage" size="lg"></ngb-pagination>
<ngb-pagination [collectionSize]="50" [(page)]="currentPage"></ngb-pagination>
<ngb-pagination [collectionSize]="50" [(page)]="currentPage" size="sm"></ngb-pagination>
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Output:



Example 5: Pagination alignment

Change the alignment of pagination components with flexbox utilities.

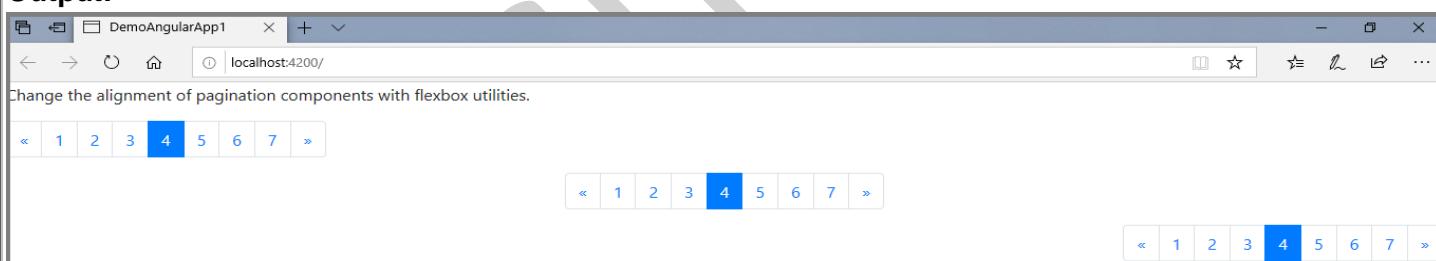
app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  page = 4;
}
```

app.component.html:

```
<ngb-pagination class="d-flex justify-content-start" [collectionSize]="70" [(page)]="page">
</ngb-pagination>
<ngb-pagination class="d-flex justify-content-center" [collectionSize]="70" [(page)]="page">
</ngb-pagination>
<ngb-pagination class="d-flex justify-content-end" [collectionSize]="70" [(page)]="page">
</ngb-pagination>
```

Output:



Example 6: Disabled pagination:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  page = 3;
  isEnabled = true;
  toggleEnabled() {
    this.isEnabled = !this.isEnabled;
  }
}
```

app.component.html:

```
<div class="p-2">
  <ngb-pagination [collectionSize]="70" [(page)]="page" [disabled]="'isDisabled'>
  </ngb-pagination>
  <hr>
  <button class="btn btn-sm btn-outline-primary" (click)="toggleDisabled()">
    Toggle disabled
  </button>
</div>
```

Output:



Example 7: Global configuration

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbPaginationConfig } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [NgbPaginationConfig] // add NgbPaginationConfig to the component providers
})
export class AppComponent {
  page = 4;

  constructor(config: NgbPaginationConfig) {
    // customize default values of paginations used by this component
    config.size = 'sm';
    config.boundaryLinks = true;
  }
}
```

app.component.html:

```
<div class="p-2">
  <ngb-pagination [collectionSize]="70" [(page)]="page"></ngb-pagination>
</div>
```

Output:



There are following properties used in the above examples:

boundaryLinks	If true , the "First" and "Last" page links are shown. Type: boolean Default value: false — initialized from NgbPaginationConfig service
collectionSize	The number of items in your paginated collection. Note, that this is not the number of pages. Page numbers are calculated dynamically based on collectionSize and pageSize . Ex. if you have 100 items in your collection and displaying 20 items per page, you'll end up with 5 pages. Type: number
directionLinks	If true , the "Next" and "Previous" page links are shown. Type: boolean Default value: true — initialized from NgbPaginationConfig service
disabled	If true , pagination links will be disabled. Type: boolean Default value: false — initialized from NgbPaginationConfig service
ellipses	If true , the ellipsis symbols and first/last page numbers will be shown when maxSize > number of pages. Type: boolean Default value: true — initialized from NgbPaginationConfig service
maxSize	The maximum number of pages to display. Type: number Default value: 0 — initialized from NgbPaginationConfig service
page	The current page. Page numbers start with 1 . Type: number Default value: 1
pageSize	The number of items per page. Type: number Default value: 10 — initialized from NgbPaginationConfig service
rotate	Whether to rotate pages when maxSize > number of pages. The current page always stays in the middle if true . Type: boolean Default value: false — initialized from NgbPaginationConfig service
size	The pagination display size. Bootstrap currently supports small and large sizes. Type: 'sm' 'lg' Default value: - — initialized from NgbPaginationConfig service

12. Progressbar

NgbProgressbar – Component:

A directive that provides feedback on the progress of a workflow or an action.

Selector: ngb-progressbar

Example 1: Contextual progress bars

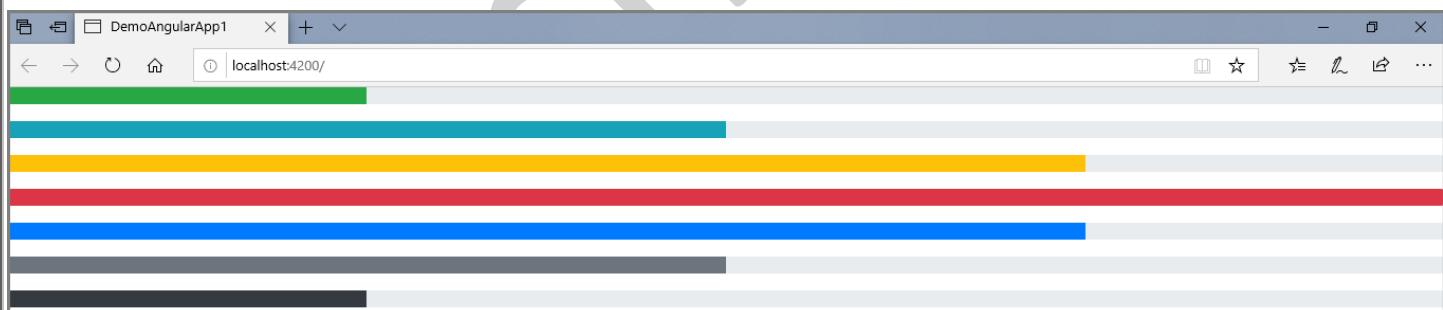
app.component.ts:

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

app.component.html:

```
<p><ngb-progressbar type="success" [value]="25"></ngb-progressbar></p>
<p><ngb-progressbar type="info" [value]="50"></ngb-progressbar></p>
<p><ngb-progressbar type="warning" [value]="75"></ngb-progressbar></p>
<p><ngb-progressbar type="danger" [value]="100"></ngb-progressbar></p>
<p><ngb-progressbar type="primary" [value]="75"></ngb-progressbar></p>
<p><ngb-progressbar type="secondary" [value]="50"></ngb-progressbar></p>
<p><ngb-progressbar type="dark" [value]="25"></ngb-progressbar></p>
```

Output:



Example 2: Contextual text progress bars:

app.component.ts:

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

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

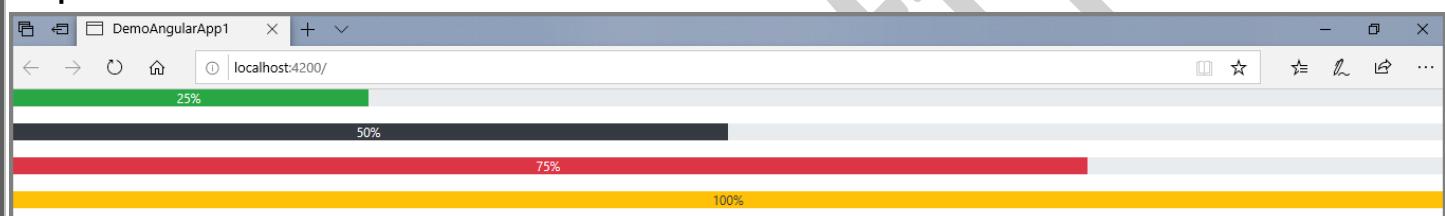
app.component.html:

```

<p>
    <ngb-progressbar type="success" textType="white" [value]="25" showValue="true">
    </ngb-progressbar>
</p>
<p>
    <ngb-progressbar type="dark" textType="white" [value]="50" showValue="true">
    </ngb-progressbar>
</p>
<p>
    <ngb-progressbar type="info" textType="white" [value]="75" showValue="true">
    </ngb-progressbar>
</p>
<p>
    <ngb-progressbar type="warning" textType="dark" [value]="100" showValue="true">
    </ngb-progressbar>
</p>

```

Output:



Example 3: Progress bars with current value labels:

app.component.ts:

```

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

```

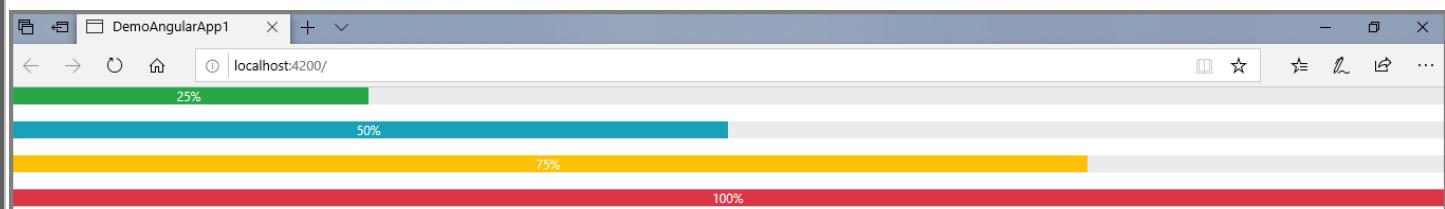
app.component.html:

```

<p>
    <ngb-progressbar showValue="true" type="success" [value]="25">
    </ngb-progressbar>
</p>
<p>
    <ngb-progressbar [showValue]="true" type="info" [value]="50">
    </ngb-progressbar>
</p>
<p>
    <ngb-progressbar showValue="true" type="warning" [value]="150" [max]="200">
    </ngb-progressbar>
</p>
<p>
    <ngb-progressbar [showValue]="true" type="danger" [value]="150" [max]="150">
    </ngb-progressbar>
</p>

```

Output:



Example 4: Striped progress bars:

app.component.ts:

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

app.component.html:

```
<p><ngb-progressbar type="success" [value]="25" [striped]="true"></ngb-progressbar></p>
<p><ngb-progressbar type="info" [value]="50" [striped]="true"></ngb-progressbar></p>
<p><ngb-progressbar type="warning" [value]="75" [striped]="true"></ngb-progressbar></p>
<p><ngb-progressbar type="danger" [value]="100" [striped]="true"></ngb-progressbar></p>
```

Output:



Example 5: Progress bars with custom labels:

app.component.ts:

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

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

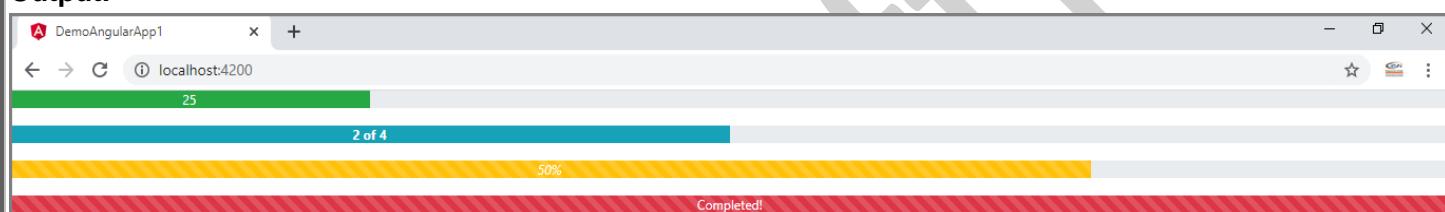
app.component.html:

```

<p>
    <ngb-progressbar type="success" [value]="25">25</ngb-progressbar>
</p>
<p>
    <ngb-progressbar type="info" [value]="50">Copying file <b>2</b> of 4</ngb-progressbar>
</p>
<p>
    <ngb-progressbar type="warning" [value]="75" [striped]="true" [animated]="true">
        <i>50%</i>
    </ngb-progressbar>
</p>
<p>
    <ngb-progressbar type="danger" [value]="100" [striped]="true">
        Completed!
    </ngb-progressbar>
</p>

```

Output:



Example 6: Progress bars with height:

app.component.ts:

```

import { Component } from '@angular/core';
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {
    height = '20px';
}

```

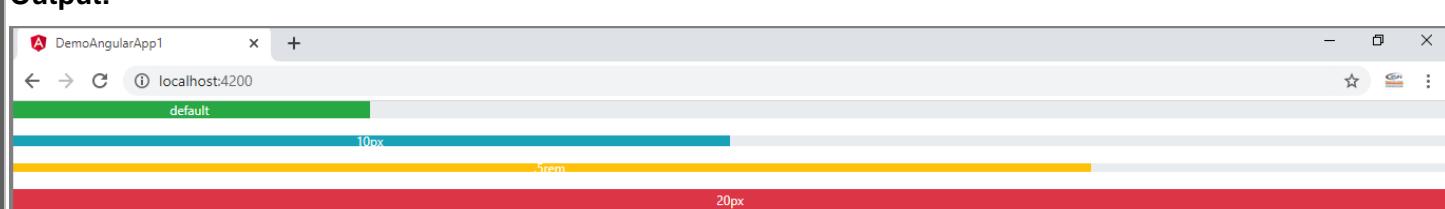
app.component.html:

```

<p><ngb-progressbar type="success" [value]="25">default</ngb-progressbar></p>
<p><ngb-progressbar type="info" [value]="50" height="10px">10px</ngb-progressbar></p>
<p><ngb-progressbar type="warning" [value]="75" height=".5rem">.5rem</ngb-progressbar></p>
<p>
    <ngb-progressbar type="danger" [value]="100" [height]="{{height}}">
        {{height}}
    </ngb-progressbar>
</p>

```

Output:



Example 6: Global configuration of progress bars:

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbProgressbarConfig } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [NgbProgressbarConfig] // add the NgbProgressbarConfig to the component provider
})
export class AppComponent {
  constructor(config: NgbProgressbarConfig) {
    // customize default values of progress bars used by this component
    config.max = 1000;
    config.striped = true;
    config.animated = true;
    config.type = 'success';
    config.height = '20px';
  }
}
```

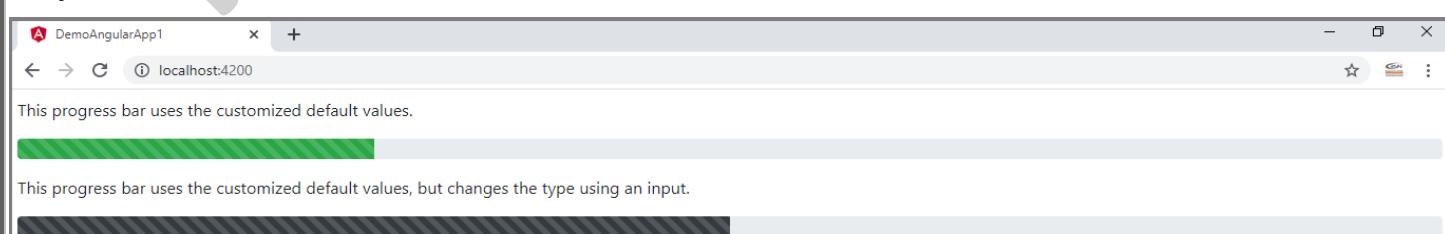
app.component.html:



```
<div class="p-2">
  <p>This progress bar uses the customized default values.</p>
  <p>
    <ngb-progressbar value="250"></ngb-progressbar>
  </p>

  <p>
    This progress bar uses the customized default values, but changes the type using an input.
  </p>
  <p>
    <ngb-progressbar value="500" type="dark"></ngb-progressbar>
  </p>
</div>
```

Output:



There are following properties used in the above examples:

animated	If true , the stripes on the progress bar are animated. Takes effect only for browsers supporting CSS3 animations, and if striped is true . Type: boolean Default value: false — initialized from NgbProgressbarConfig service
height	The height of the progress bar. Accepts any valid CSS height values, ex. "2rem" Type: string Default value: - — initialized from NgbProgressbarConfig service
max	The maximal value to be displayed in the progress bar. Should be a positive number. Will default to 100 otherwise. Type: number Default value: 100 — initialized from NgbProgressbarConfig service
showValue	If true , the current percentage will be shown in the xx% format. Type: boolean Default value: false — initialized from NgbProgressbarConfig service
striped	If true , the progress bars will be displayed as striped. Type: boolean Default value: false — initialized from NgbProgressbarConfig service
textType since 5.2.0	Optional text variant type of the progress bar. Supports types based on Bootstrap background color variants, like: "success", "info", "warning", "danger", "primary", "secondary", "dark" and so on. Type: string Default value: - — initialized from NgbProgressbarConfig service
type	The type of the progress bar. Supports types based on Bootstrap background color variants, like: "success", "info", "warning", "danger", "primary", "secondary", "dark" and so on. Type: string Default value: - — initialized from NgbProgressbarConfig service
value	The current value for the progress bar. Should be in the [0, max] range. Type: number Default value: 0



13. Rating

NgbRating – Component:

A directive that helps visualising and interacting with a star rating bar.

Selector: ngb-rating

StarTemplateContext – Interface:

The context for the custom star display template defined in the starTemplate.

NgbRatingConfig – Configuration:

A configuration service for the **NgbRating** component.

You can inject this service, typically in your root component, and customize the values of its properties in order to provide default values for all the ratings used in the application.

Example 1: Basic demo

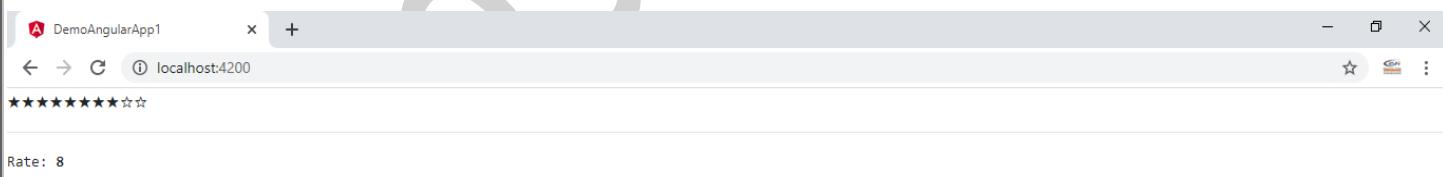
app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  currentRate = 8;
}
```

app.component.html:

```
<ngb-rating [(rate)]="currentRate"></ngb-rating>
<hr>
<pre>Rate: <b>{{currentRate}}</b></pre>
```

Output:



Example 2: Events and readonly ratings:

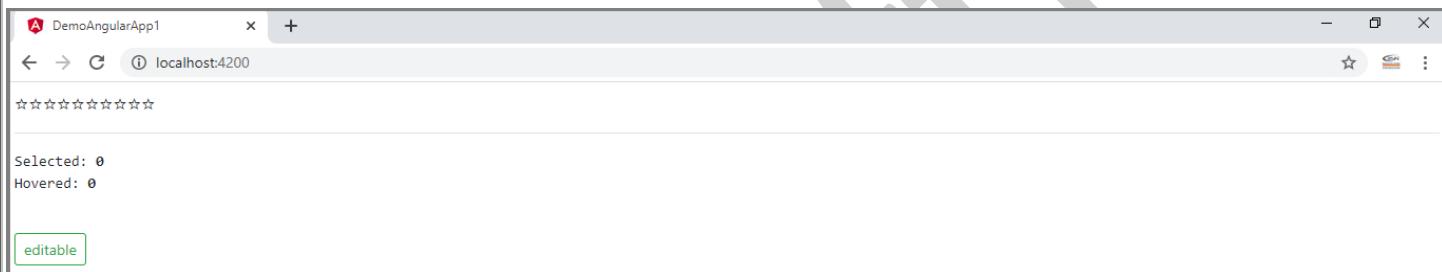
app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  selected = 0;
  hovered = 0;
  readonly = false;
}
```

app.component.html:

```
<div class="p-2">
  <ngb-rating [(rate)]="selected" (hover)="hovered=$event" (leave)="hovered=0"
    [readonly]="readonly">
  </ngb-rating>
  <hr>
  <pre>
    Selected: <b>{{selected}}</b>
    Hovered: <b>{{hovered}}</b>
  </pre>
  <button class="btn btn-sm btn-outline-{{readonly ? 'danger' : 'success'}}"
    (click)="readonly = !readonly">
    {{ readonly ? "readonly" : "editable"}}
  </button>
</div>
```

Output:



Example 3: Custom star template:

app.component.ts:

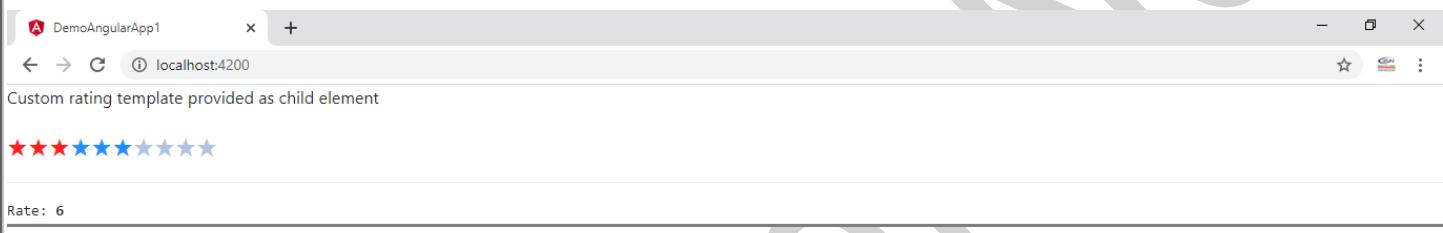
```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: [
    `
      .star {
        font-size: 1.5rem;
        color: #b0c4de;
      }
      .filled {
        color: #1e90ff;
      }
      .bad {
        color: #deb0b0;
      }
      .filled.bad {
        color: #ff1e1e;
      }
    `
  ]
})
export class AppComponent {
  currentRate = 6;
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.component.html:

```
<p>Custom rating template provided as child element</p>
<ngb-rating [(rate)]="currentRate">
  <ng-template let-fill="fill" let-index="index">
    <span class="star" [class.filled]="fill === 100" [class.bad]="index < 3">
      &#9733;
    </span>
  </ng-template>
</ngb-rating>
<hr>
<pre>Rate: <b>{{currentRate}}</b></pre>
```

Output:



Example 4: Custom decimal rating:

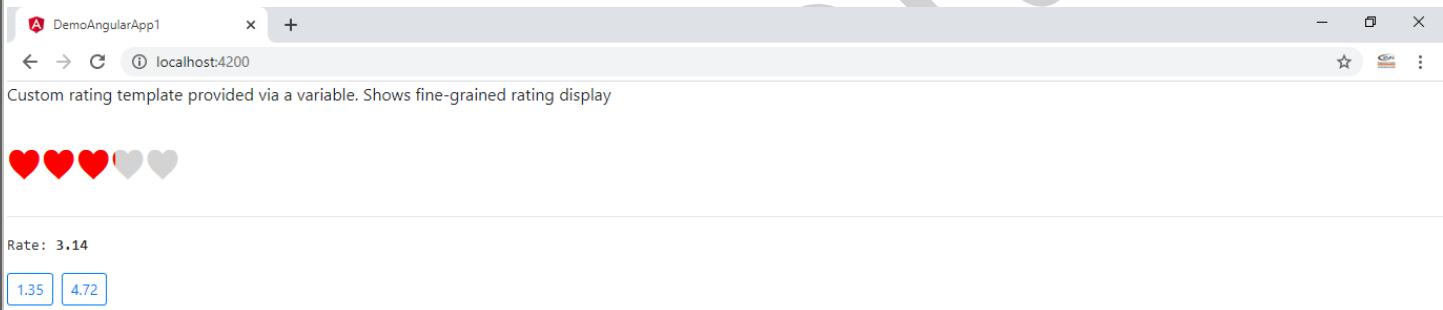
app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: [
    .star {
      position: relative;
      display: inline-block;
      font-size: 3rem;
      color: #d3d3d3;
    }
    .full {
      color: red;
    }
    .half {
      position: absolute;
      display: inline-block;
      overflow: hidden;
      color: red;
    }
  ]
})
export class AppComponent {
  currentRate = 3.14;
}
```

app.component.html:

```
<p>Custom rating template provided via a variable. Shows fine-grained rating display</p>
<ng-template #t let-fill="fill">
  <span class="star" [class.full]="fill === 100">
    <span class="half" [style.width.%]="fill">&hearts;</span>&hearts;
  </span>
</ng-template>
<ngb-rating [(rate)]="currentRate" [starTemplate]="t" [readonly]="true" max="5"></ngb-rating>
<hr>
<pre>Rate: <b>{{currentRate}}</b></pre>
<button class="btn btn-sm btn-outline-primary mr-2" (click)="currentRate = 1.35">
  1.35
</button>
<button class="btn btn-sm btn-outline-primary mr-2" (click)="currentRate = 4.72">
  4.72
</button>
```

Output:



A screenshot of a web browser window titled "DemoAngularApp1". The address bar shows "localhost:4200". The page content is as follows:

Custom rating template provided via a variable. Shows fine-grained rating display

Rate: 3.14



1.35 4.72

Example 5: Form integration:

app.component.ts:

```
import { Component } from '@angular/core';
import { FormControl, Validators } from '@angular/forms';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  ctrl = new FormControl(null, Validators.required);

  toggle() {
    if (this.ctrl.disabled) {
      this.ctrl.enable();
    } else {
      this.ctrl.disable();
    }
  }
}
```

app.component.html:

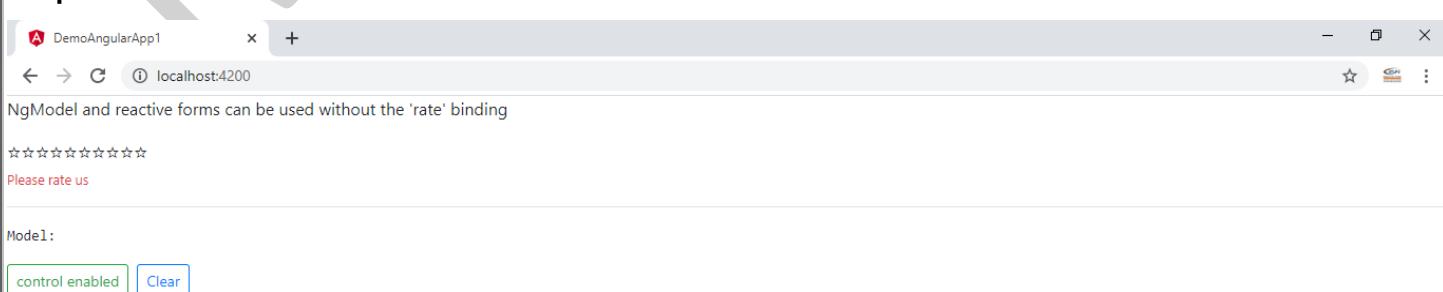
```
<p>NgModel and reactive forms can be used without the 'rate' binding</p>
<div class="form-group">
  <ngb-rating [formControl]="ctrl"></ngb-rating>
  <div class="form-text small">
    <div *ngIf="ctrl.valid" class="text-success">Thanks!</div>
    <div *ngIf="ctrl.invalid" class="text-danger">Please rate us</div>
  </div>
</div>
<hr>
<pre>Model: <b>{{ ctrl.value }}</b></pre>
<button class="btn btn-sm btn-outline-{{ ctrl.disabled ? 'danger' : 'success' }} mr-2"
       (click)="toggle()">
  {{ ctrl.disabled ? "control disabled" : "control enabled" }}
</button>
<button class="btn btn-sm btn-outline-primary mr-2" (click)="ctrl.setValue(null)">
  Clear
</button>
```

app.module.ts: FormControl is exposed as a part of **ReactiveFormsModule** and NOT the **FormsModule** so need to import the **ReactiveFormsModule** in your **@NgModule** which is the **AppModule**.

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

@NgModule({
  imports: [..., ReactiveFormsModule, ...],
  ...
})
export class AppModule {...}
```

Output:



Example 6: Global configuration of ratings:

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbRatingConfig } from '@ng-bootstrap/ng-bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  providers: [NgbRatingConfig] // add NgbRatingConfig to the component providers
})
export class AppComponent {
  constructor(config: NgbRatingConfig) {
    // customize default values of ratings used by this component tree
    config.max = 5;
    config.readonly = true;
  }
}
```

app.component.html:

```
<p>This rating uses customized default values.</p>
<ngb-rating [rate]="3"></ngb-rating>
```

Output:



There are following properties & events used in the above examples:

max	The maximal rating that can be given. Type: number Default value: 10 — initialized from NgbRatingConfig service
rate	The current rating. Could be a decimal value like 3.75. Type: number
readonly	If true , the rating can't be changed. Type: boolean Default value: false — initialized from NgbRatingConfig service
resettable	If true , the rating can be reset to 0 by mouse clicking currently set rating. Type: boolean Default value: false — initialized from NgbRatingConfig service
starTemplate	The template to override the way each star is displayed. Alternatively put an <code><ng-template></code> as the only child of your <code><ngb-rating></code> element Type: TemplateRef<StarTemplateContext>
hover	An event emitted when the user is hovering over a given rating. Event payload equals to the rating being hovered over.
leave	An event emitted when the user stops hovering over a given rating. Event payload equals to the rating of the last item being hovered over.
rateChange	An event emitted when the user selects a new rating. Event payload equals to the newly selected rating.
fill	The star fill percentage, an integer in the [0, 100] range. Type: number
index	Index of the star, starts with 0. Type: number

14. Table

Bootstrap provides the some basic styling for the tables including CSS classes for responsiveness, striping odd/even rows, changing borders and captions, hovering rows, etc. These styles are opt-in and can be used with pure Angular to produce something like this:

#	Country	Area	Population
1	Russia	17,075,200	146,989,754
2	Canada	9,976,140	36,624,199
3	United States	9,629,091	324,459,463

Note:

At the moment ng-bootstrap do not provide a dedicated component like **NgbTable** or **NgbGrid** as a part of ng-bootstrap project.

Example 1: Basic table:

Using the most basic table markup, here's how .table-based tables look in Bootstrap.

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  students = [
    {name: "John", age: 21, gender: "Male"},
    {name: "Alina", age: 23, gender: "Female"},
    {name: "David", age: 25, gender: "Male"},
    {name: "Smith", age: 24, gender: "Male"},
    {name: "Peter", age: 27, gender: "Male"}
  ];
}
```

```
app.component.html:
<table class="table">
  <thead>
    <tr>
      <th scope="col">#</th>
      <th scope="col">Name</th>
      <th scope="col">Age</th>
      <th scope="col">Gender</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let student of students; index as i">
      <th scope="row">{{ i + 1 }}</th>
      <td>{{ student.name }}</td>
      <td>{{ student.age }}</td>
      <td>{{ student.gender }}</td>
    </tr>
  </tbody>
</table>
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Output:

#	Name	Age	Gender
1	John	21	Male
2	Alina	23	Female
3	David	25	Male
4	Smith	24	Male
5	Peter	27	Male

You can also invert the colors—with light text on dark backgrounds—with `.table-dark`.

app.component.html:

```
<table class="table table-dark">
  <thead>
    <tr>
      <th scope="col">#</th>
      <th scope="col">Name</th>
      <th scope="col">Age</th>
      <th scope="col">Gender</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let student of students;index as i">
      <th scope="row">{{ i + 1 }}</th>
      <td>{{ student.name }}</td>
      <td>{{ student.age }}</td>
      <td>{{ student.gender }}</td>
    </tr>
  </tbody>
</table>
```

Output:

#	Name	Age	Gender
1	John	21	Male
2	Alina	23	Female
3	David	25	Male
4	Smith	24	Male
5	Peter	27	Male



Table head options

Similar to tables and dark tables, use the modifier classes `.thead-light` or `.thead-dark` to make `<thead>`s appear light or dark gray.

app.component.html:

```
<table class="table">
  <thead class="thead-dark">
    <tr>
      <th scope="col">#</th>
      <th scope="col">Name</th>
      <th scope="col">Age</th>
      <th scope="col">Gender</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let student of students;index as i">
      <th scope="row">{{ i + 1 }}</th>
      <td>{{ student.name }}</td>
      <td>{{ student.age }}</td>
      <td>{{ student.gender }}</td>
    </tr>
  </tbody>
</table>
```

Output:

#	Name	Age	Gender
1	John	21	Male
2	Alina	23	Female
3	David	25	Male
4	Smith	24	Male
5	Peter	27	Male

```
<table class="table">
  <thead class="thead-light">
    <tr>
      <th scope="col">#</th>
      <th scope="col">Name</th>
      <th scope="col">Age</th>
      <th scope="col">Gender</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let student of students;index as i">
      <th scope="row">{{ i + 1 }}</th>
      <td>{{ student.name }}</td>
      <td>{{ student.age }}</td>
      <td>{{ student.gender }}</td>
    </tr>
  </tbody>
</table>
```

Output:

#	Name	Age	Gender
1	John	21	Male
2	Alina	23	Female
3	David	25	Male
4	Smith	24	Male
5	Peter	27	Male

Striped rows:

Use `.table-striped` to add zebra-striping to any table row within the `<tbody>`.

app.component.html:

```
<table class="table table-striped">
  <thead>
    <tr>
      <th scope="col">#</th>
      <th scope="col">Name</th>
      <th scope="col">Age</th>
      <th scope="col">Gender</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let student of students;index as i">
      <th scope="row">{{ i + 1 }}</th>
      <td>{{ student.name }}</td>
      <td>{{ student.age }}</td>
      <td>{{ student.gender }}</td>
    </tr>
  </tbody>
</table>
```

Output:

#	Name	Age	Gender
1	John	21	Male
2	Alina	23	Female
3	David	25	Male
4	Smith	24	Male
5	Peter	27	Male



table-striped with table-dark:

app.component.html:

```
<table class="table table-striped table-dark">
  <thead>
    <tr>
      <th scope="col">#</th>
      <th scope="col">Name</th>
      <th scope="col">Age</th>
      <th scope="col">Gender</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let student of students;index as i">
      <th scope="row">{{ i + 1 }}</th>
      <td>{{ student.name }}</td>
      <td>{{ student.age }}</td>
      <td>{{ student.gender }}</td>
    </tr>
  </tbody>
</table>
```

Output:



#	Name	Age	Gender
1	John	21	Male
2	Alina	23	Female
3	David	25	Male
4	Smith	24	Male
5	Peter	27	Male

Bordered table:

Add `.table-bordered` for borders on all sides of the table and cells.

app.component.html:

```
<table class="table table-bordered">
  <thead>
    <tr>
      <th scope="col">#</th>
      <th scope="col">Name</th>
      <th scope="col">Age</th>
      <th scope="col">Gender</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let student of students;index as i">
      <th scope="row">{{ i + 1 }}</th>
      <td>{{ student.name }}</td>
      <td>{{ student.age }}</td>
      <td>{{ student.gender }}</td>
    </tr>
  </tbody>
</table>
```

Output:

#	Name	Age	Gender
1	John	21	Male
2	Alina	23	Female
3	David	25	Male
4	Smith	24	Male
5	Peter	27	Male

Borderless table

Add `.table-borderless` for a table without borders.

app.component.html:

```
<table class="table table-borderless">
  <thead>
    <tr>
      <th scope="col">#</th>
      <th scope="col">Name</th>
      <th scope="col">Age</th>
      <th scope="col">Gender</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let student of students;index as i">
      <th scope="row">{{ i + 1 }}</th>
      <td>{{ student.name }}</td>
      <td>{{ student.age }}</td>
      <td>{{ student.gender }}</td>
    </tr>
  </tbody>
</table>
```

Output:

#	Name	Age	Gender
1	John	21	Male
2	Alina	23	Female
3	David	25	Male
4	Smith	24	Male
5	Peter	27	Male

.table-borderless can also be used on dark tables.

app.component.html:

```
<table class="table table-borderless table-dark">
  <thead>
    <tr>
      <th scope="col">#</th>
      <th scope="col">Name</th>
      <th scope="col">Age</th>
      <th scope="col">Gender</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let student of students;index as i">
      <th scope="row">{{ i + 1 }}</th>
      <td>{{ student.name }}</td>
      <td>{{ student.age }}</td>
      <td>{{ student.gender }}</td>
    </tr>
  </tbody>
</table>
```

Output:



#	Name	Age	Gender
1	John	21	Male
2	Alina	23	Female
3	David	25	Male
4	Smith	24	Male
5	Peter	27	Male

Hoverable rows

Add .table-hover to enable a hover state on table rows within a <tbody>.

app.component.html:

```
<table class="table table-hover">
  <thead>
    <tr>
      <th scope="col">#</th>
      <th scope="col">Name</th>
      <th scope="col">Age</th>
      <th scope="col">Gender</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let student of students;index as i">
      <th scope="row">{{ i + 1 }}</th>
      <td>{{ student.name }}</td>
      <td>{{ student.age }}</td>
      <td>{{ student.gender }}</td>
    </tr>
  </tbody>
</table>
```

Output:

#	Name	Age	Gender
1	John	21	Male
2	Alina	23	Female
3	David	25	Male
4	Smith	24	Male
5	Peter	27	Male

Small table:

Add **.table-sm** to make tables more compact by cutting cell padding in half.

app.component.html:

```
<table class="table table-sm">
  <thead>
    <tr>
      <th scope="col">#</th>
      <th scope="col">Name</th>
      <th scope="col">Age</th>
      <th scope="col">Gender</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let student of students;index as i">
      <th scope="row">{{ i + 1 }}</th>
      <td>{{ student.name }}</td>
      <td>{{ student.age }}</td>
      <td>{{ student.gender }}</td>
    </tr>
  </tbody>
</table>
```

Output:

#	Name	Age	Gender
1	John	21	Male
2	Alina	23	Female
3	David	25	Male
4	Smith	24	Male
5	Peter	27	Male



Contextual classes:

Use contextual classes to color table rows or individual cells.

Class	Heading	Heading
Active	Cell	Cell
Default	Cell	Cell
Primary	Cell	Cell
Secondary	Cell	Cell
Success	Cell	Cell
Danger	Cell	Cell
Warning	Cell	Cell
Info	Cell	Cell
Light	Cell	Cell
Dark	Cell	Cell

```
<!-- On rows -->
<tr class="table-active">...</tr>

<tr class="table-primary">...</tr>
<tr class="table-secondary">...</tr>
<tr class="table-success">...</tr>
<tr class="table-danger">...</tr>
<tr class="table-warning">...</tr>
<tr class="table-info">...</tr>
<tr class="table-light">...</tr>
<tr class="table-dark">...</tr>

<!-- On cells (`td` or `th`) -->
<tr>
  <td class="table-active">...</td>

  <td class="table-primary">...</td>
  <td class="table-secondary">...</td>
  <td class="table-success">...</td>
  <td class="table-danger">...</td>
  <td class="table-warning">...</td>
  <td class="table-info">...</td>
  <td class="table-light">...</td>
  <td class="table-dark">...</td>
</tr>
```

Regular table background variants are not available with the dark table, however, you may use **text or background utilities** to achieve similar styles.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



#	Heading	Heading
1	Cell	Cell
2	Cell	Cell
3	Cell	Cell
4	Cell	Cell
5	Cell	Cell
6	Cell	Cell
7	Cell	Cell
8	Cell	Cell
9	Cell	Cell

<!-- On rows -->

```
<tr class="bg-primary">...</tr>
<tr class="bg-success">...</tr>
<tr class="bg-warning">...</tr>
<tr class="bg-danger">...</tr>
<tr class="bg-info">...</tr>
```

<!-- On cells (`td` or `th`) -->

```
<tr>
  <td class="bg-primary">...</td>
  <td class="bg-success">...</td>
  <td class="bg-warning">...</td>
  <td class="bg-danger">...</td>
  <td class="bg-info">...</td>
</tr>
```

Colors:

Convey meaning through color with a handful of color utility classes. Includes support for styling links with hover states, too.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Color

.text-primary

.text-secondary

.text-success

.text-danger

.text-warning

.text-info

.text-light

.text-dark

.text-body

.text-muted

.text-white

.text-black-50

.text-white-50

```
<p class="text-primary">.text-primary</p>
<p class="text-secondary">.text-secondary</p>
<p class="text-success">.text-success</p>
<p class="text-danger">.text-danger</p>
<p class="text-warning">.text-warning</p>
<p class="text-info">.text-info</p>
<p class="text-light bg-dark">.text-light</p>
<p class="text-dark">.text-dark</p>
<p class="text-body">.text-body</p>
<p class="text-muted">.text-muted</p>
<p class="text-white bg-dark">.text-white</p>
<p class="text-black-50">.text-black-50</p>
<p class="text-white-50 bg-dark">.text-white-50</p>
```

Contextual text classes also work well on anchors with the provided hover and focus states. **Note that the .text-white and .text-muted class has no additional link styling beyond underline.**

Primary link

Secondary link

Success link

Danger link

Warning link

Info link

Light link

Dark link

Muted link

White link

```
<p><a href="#" class="text-primary">Primary link</a></p>
<p><a href="#" class="text-secondary">Secondary link</a></p>
<p><a href="#" class="text-success">Success link</a></p>
<p><a href="#" class="text-danger">Danger link</a></p>
<p><a href="#" class="text-warning">Warning link</a></p>
<p><a href="#" class="text-info">Info link</a></p>
<p><a href="#" class="text-light bg-dark">Light link</a></p>
<p><a href="#" class="text-dark">Dark link</a></p>
<p><a href="#" class="text-muted">Muted link</a></p>
<p><a href="#" class="text-white bg-dark">White link</a></p>
```



Background color:

Similar to the contextual text color classes, easily set the background of an element to any contextual class. Anchor components will darken on hover, just like the text classes. Background utilities **do not set color**, so in some cases you'll want to use `.text-*` utilities.



```
<div class="p-3 mb-2 bg-primary text-white">.bg-primary</div>
<div class="p-3 mb-2 bg-secondary text-white">.bg-secondary</div>
<div class="p-3 mb-2 bg-success text-white">.bg-success</div>
<div class="p-3 mb-2 bg-danger text-white">.bg-danger</div>
<div class="p-3 mb-2 bg-warning text-dark">.bg-warning</div>
<div class="p-3 mb-2 bg-info text-white">.bg-info</div>
<div class="p-3 mb-2 bg-light text-dark">.bg-light</div>
<div class="p-3 mb-2 bg-dark text-white">.bg-dark</div>
<div class="p-3 mb-2 bg-white text-dark">.bg-white</div>
<div class="p-3 mb-2 bg-transparent text-dark">.bg-transparent</div>
```

Captions:

A `<caption>` functions like a heading for a table. It helps users with screen readers to find a table and understand what it's about and decide if they want to read it.

app.component.html:

```
<table class="table">
  <caption>List of users</caption>
  <thead>
    <tr>
      <th scope="col">#</th>
      <th scope="col">Name</th>
      <th scope="col">Age</th>
      <th scope="col">Gender</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let student of students;index as i">
      <th scope="row">{{ i + 1 }}</th>
      <td>{{ student.name }}</td>
      <td>{{ student.age }}</td>
      <td>{{ student.gender }}</td>
    </tr>
  </tbody>
</table>
```

Output:



#	Name	Age	Gender
1	John	21	Male
2	Alina	23	Female
3	David	25	Male
4	Smith	24	Male
5	Peter	27	Male

List of users

Responsive tables

Responsive tables allow tables to be scrolled horizontally with ease. Make any table responsive across all viewports by wrapping a `.table` with `.table-responsive`. Or, pick a maximum breakpoint with which to have a responsive table up to by using `.table-responsive{-sm|-md|-lg|-xl}`.

Vertical clipping/truncation

Responsive tables make use of `overflow-y: hidden`, which clips off any content that goes beyond the bottom or top edges of the table. In particular, this can clip off dropdown menus and other third-party widgets.

Always responsive

Across every breakpoint, use `.table-responsive` for horizontally scrolling tables.

| # | Heading |
|---|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| 1 | Cell |
| 2 | Cell |
| 3 | Cell |

```
<div class="table-responsive">
  <table class="table">
    ...
  </table>
</div>
```

Breakpoint specific

Use `.table-responsive{-sm|-md|-lg|-xl}` as needed to create responsive tables up to a particular breakpoint. From that breakpoint and up, the table will behave normally and not scroll horizontally.

These tables may appear broken until their responsive styles apply at specific viewport widths.

```
<div class="table-responsive-sm">
  <table class="table">
    ...
  </table>
</div>
```

```
<div class="table-responsive-md">
  <table class="table">
    ...
  </table>
</div>
```

```
<div class="table-responsive-lg">
  <table class="table">
    ...
  </table>
</div>
```

```
<div class="table-responsive-xl">
  <table class="table">
    ...
  </table>
</div>
```

15. Tabset

Since version **5.2.0** please consider using **Nav directives** as a more flexible alternative. Tabset will not be supported anymore and will be deprecated.

- **NgbTab: Directive**

A directive representing an individual tab.

Selector: ngb-tab

- **NgbTabContent: Directive**

A directive to wrap content to be displayed in a tab.

Selector: ng-template[ngbTabContent]

- **NgbTabTitle: Directive**

A directive to wrap tab titles that need to contain HTML markup or other directives.

Alternatively you could use the NgbTab.title input for string titles.

Selector: ng-template[ngbTabTitle]

- **NgbTabset: Component**

A component that makes it easy to create tabbed interface.

Selector: ngb-tabset

Exported as: ngbTabset

- **NgbTabChangeEvent**

The payload of the change event fired right before the tab change.

- **NgbTabsetConfig: Configuration**

A configuration service for the NgbTabset component.

You can inject this service, typically in your root component, and customize the values of its properties in order to provide default values for all the tabs used in the application.

Examples:

Example 1: Tabset

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```



app.component.html:

```
<ngb-tabset [destroyOnHide]="false">
  <ngb-tab title="Tab1">
    <ng-template ngbTabContent>
      <p>Tab1 Content !!!</p>
    </ng-template>
  </ngb-tab>
  <ngb-tab>
    <ng-template ngbTabTitle><b>Tab2</b></ng-template>
    <ng-template ngbTabContent>
      <p>Tab2 Content !!!</p>
    </ng-template>
  </ngb-tab>
  <ngb-tab title="Disabled Tab" [disabled]="true">
    <ng-template ngbTabContent>
      <p>Disabled Tab Content</p>
    </ng-template>
  </ngb-tab>
</ngb-tabset>
```

Output:



Example 2: Pills

app.component.ts:

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

app.component.ts:

```
<div class="p-2">
  <ngb-tabset type="pills">
    <ngb-tab title="Tab1">
      <ng-template ngbTabContent>
        <p>Tab1 Content !!!</p>
      </ng-template>
    </ngb-tab>
    <ngb-tab>
      <ng-template ngbTabTitle><b>Tab2</b></ng-template>
      <ng-template ngbTabContent>
        <p>Tab2 Content !!!</p>
      </ng-template>
    </ngb-tab>
    <ngb-tab title="Tab3">
      <ng-template ngbTabContent>
        <p>Tab3 Content !!!</p>
      </ng-template>
    </ngb-tab>
  </ngb-tabset>
</div>
```



Output:



Example 3: Select an active tab by id

app.component.ts:

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

app.component.html:

```
<div class="p-2">
  <ngb-tabset #t="ngbTabset">
    <ngb-tab id="tab-1" title="Tab1">
      <ng-template ngbTabContent>
        <p>Tab1 Content !!!</p>
      </ng-template>
    </ngb-tab>
    <ngb-tab id="tab-2">
      <ng-template ngbTabTitle><b>Tab2</b></ng-template>
      <ng-template ngbTabContent>
        <p>Tab2 Content !!!</p>
      </ng-template>
    </ngb-tab>
    <ngb-tab id="tab-3" title="Tab3">
      <ng-template ngbTabContent>
        <p>Tab3 Content !!!</p>
      </ng-template>
    </ngb-tab>
  </ngb-tabset>
  <p>
    <button class="btn btn-outline-primary" (click)="t.select('tab-2')">
      Selected tab with "tab-2" id
    </button>
  </p>
</div>
```

Output:



Example 3: Prevent tab change

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbTabChangeEvent } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  public beforeChange($event: NgbTabChangeEvent) {
    if ($event.nextId === 'tab-2') {
      $event.preventDefault();
    }
  }
}
```

app.component.html:

```
<div class="p-2">
  <ngb-tabset (tabChange)="beforeChange($event)">
    <ngb-tab id="tab-1" title="Tab1">
      <ng-template ngbTabContent>
        <p>Tab1 Content !!!</p>
      </ng-template>
    </ngb-tab>
    <ngb-tab id="tab-2">
      <ng-template ngbTabTitle><b>Tab 2 [can't be selected...]</b></ng-template>
      <ng-template ngbTabContent>
        <p>Tab2 Content !!!</p>
      </ng-template>
    </ngb-tab>
    <ngb-tab id="tab-3" title="Tab3">
      <ng-template ngbTabContent>
        <p>Tab3 Content !!!</p>
      </ng-template>
    </ngb-tab>
  </ngb-tabset>
</div>
```

Output:



Example 4: Nav justification

app.component.ts:

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

app.component.html:

```
<div class="p-2">
  <ngb-tabset [justify]="currentJustify">
    <ngb-tab id="tab-1" title="Tab1">
      <ng-template ngbTabContent>
        <p>Tab1 Content !!!</p>
      </ng-template>
    </ngb-tab>
    <ngb-tab id="tab-2">
      <ng-template ngbTabTitle><b>Tab 2</b></ng-template>
      <ng-template ngbTabContent>
        <p>Tab2 Content !!!</p>
      </ng-template>
    </ngb-tab>
    <ngb-tab id="tab-3" title="Tab3 Page Title">
      <ng-template ngbTabContent>
        <p>Tab3 Content !!!</p>
      </ng-template>
    </ngb-tab>
  </ngb-tabset>
  <div class="btn-group btn-group-toggle" ngbRadioGroup [(ngModel)]="currentJustify">
    <label ngbButtonLabel class="btn-outline-primary btn-sm">
      <input ngbButton type="radio" value="start">Start
    </label>
    <label ngbButtonLabel class="btn-outline-primary btn-sm">
      <input ngbButton type="radio" value="center">Center
    </label>
    <label ngbButtonLabel class="btn-outline-primary btn-sm">
      <input ngbButton type="radio" value="end">End
    </label>
    <label ngbButtonLabel class="btn-outline-primary btn-sm">
      <input ngbButton type="radio" value="fill">Fill
    </label>
    <label ngbButtonLabel class="btn-outline-primary btn-sm">
      <input ngbButton type="radio" value="justified">Justified
    </label>
  </div>
</div>
```

Output:



Example 5: Nav orientation

app.component.ts:

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

app.component.html:

```
<div class="p-2">
  <ngb-tabset type="pills" [orientation]="currentOrientation">
    <ngb-tab id="tab-1" title="Tab1">
      <ng-template ngbTabContent>
        <p>Tab1 Content !!!</p>
      </ng-template>
    </ngb-tab>
    <ngb-tab id="tab-2">
      <ng-template ngbTabTitle><b>Tab 2</b></ng-template>
      <ng-template ngbTabContent>
        <p>Tab2 Content !!!</p>
      </ng-template>
    </ngb-tab>
    <ngb-tab id="tab-3" title="Tab3">
      <ng-template ngbTabContent>
        <p>Tab3 Content !!!</p>
      </ng-template>
    </ngb-tab>
  </ngb-tabset>
  <div class="btn-group btn-group-toggle" ngbRadioGroup [(ngModel)]="currentOrientation">
    <label ngbButtonLabel class="btn-outline-primary btn-sm">
      <input ngbButton type="radio" value="horizontal">Horizontal
    </label>
    <label ngbButtonLabel class="btn-outline-primary btn-sm">
      <input ngbButton type="radio" value="vertical">Vertical
    </label>
  </div>
</div>
```

Output:



Example 6: Global configuration of tabs

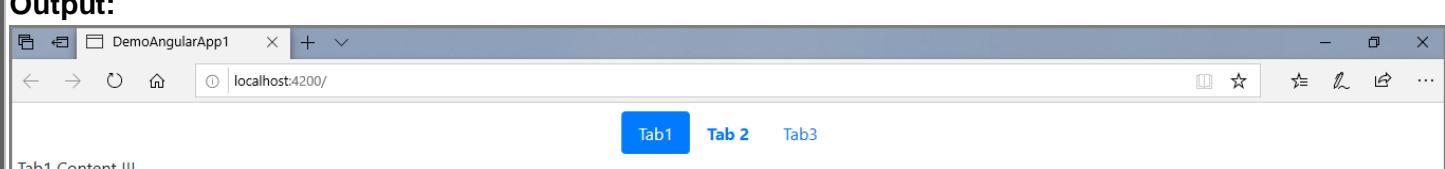
app.component.ts:

```
import { Component } from '@angular/core';
import { NgbTabsetConfig } from '@ng-bootstrap/ng-bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [NgbTabsetConfig] // add NgbTabsetConfig to the component providers
})
export class AppComponent {
  constructor(config: NgbTabsetConfig) {
    // customize default values of tabs used by this component tree
    config.justify = 'center';
    config.type = 'pills';
  }
}
```

app.component.html:

```
<div class="p-2">
  <ngb-tabset>
    <ngb-tab id="tab-1" title="Tab1">
      <ng-template ngbTabContent>
        <p>Tab1 Content !!!</p>
      </ng-template>
    </ngb-tab>
    <ngb-tab id="tab-2">
      <ng-template ngbTabTitle><b>Tab 2</b></ng-template>
      <ng-template ngbTabContent>
        <p>Tab2 Content !!!</p>
      </ng-template>
    </ngb-tab>
    <ngb-tab id="tab-3" title="Tab3">
      <ng-template ngbTabContent>
        <p>Tab3 Content !!!</p>
      </ng-template>
    </ngb-tab>
  </ngb-tabset>
</div>
```

Output:



In the above examples, these are the following properties & events used:

disabled	If <code>true</code> , the current tab is disabled and can't be toggled.
	Type: <code>boolean</code>
	Default value: <code>false</code>
id	The tab identifier. Must be unique for the entire document for proper accessibility support.
	Type: <code>string</code>
title	The tab title. Use the <code>NgbTabTitle</code> directive for non-string titles.
	Type: <code>string</code>
activelid	The identifier of the tab that should be opened initially . For subsequent tab switches use the <code>.select()</code> method and the <code>(tabChange)</code> event.
	Type: <code>string</code>
destroyOnHide	If <code>true</code> , non-visible tabs content will be removed from DOM. Otherwise it will just be hidden.
	Type: <code>boolean</code>
	Default value: <code>true</code>
justify	The horizontal alignment of the tabs with flexbox utilities.
	Type: <code>"start" "center" "end" "fill" "justified"</code>
	Default value: <code>start</code> — initialized from <code>NgbTabsetConfig</code> service
orientation	The orientation of the tabset.
	Type: <code>'horizontal' 'vertical'</code>
	Default value: <code>horizontal</code> — initialized from <code>NgbTabsetConfig</code> service
type	Type of navigation to be used for tabs. Currently Bootstrap supports only <code>"tabs"</code> and <code>"pills"</code> . Since <code>3.0.0</code> can also be an arbitrary string (ex. for custom themes).
	Type: <code>'tabs' 'pills' string</code>
	Default value: <code>tabs</code> — initialized from <code>NgbTabsetConfig</code> service
tabChange	A tab change event emitted right before the tab change happens.
select	<code>select(tabId: string) => void</code> Selects the tab with the given id and shows its associated content panel. Any other tab that was previously selected becomes unselected and its associated pane is removed from DOM or hidden depending on the <code>destroyOnHide</code> value.
nextId	The id of the newly selected tab.
	Type: <code>string</code>
preventDefault	Calling this function will prevent tab switching.
	Type: <code>() => void</code>

16. Timepicker

NgbTimepicker: Component

A directive that helps with picking hours, minutes and seconds.

Selector: ngb-timepicker

hourStep	The number of hours to add/subtract when clicking hour spinners. Type: number Default value: 1 — initialized from NgbTimepickerConfig service
meridian	Whether to display 12H or 24H mode. Type: boolean Default value: false — initialized from NgbTimepickerConfig service
minuteStep	The number of minutes to add/subtract when clicking minute spinners. Type: number Default value: 1 — initialized from NgbTimepickerConfig service
readonlyInputs	If true, the timepicker is readonly and can't be changed. Type: boolean Default value: false — initialized from NgbTimepickerConfig service
seconds	If true, it is possible to select seconds. Type: boolean Default value: false — initialized from NgbTimepickerConfig service
secondStep	The number of seconds to add/subtract when clicking second spinners. Type: number Default value: 1 — initialized from NgbTimepickerConfig service
size	The size of inputs and buttons. Type: 'small' 'medium' 'large' Default value: medium — initialized from NgbTimepickerConfig service
spinners	If true, the spinners above and below inputs are visible. Type: boolean Default value: true — initialized from NgbTimepickerConfig service

NgbTimeStruct: Interface

An interface for the time model used by the timepicker.

hour	The hour in the [0, 23] range. Type: number
minute	The minute in the [0, 59] range. Type: number
second	The second in the [0, 59] range. Type: number

NgbTimepickerConfig: Configuration

A configuration service for the NgbTimepicker component.

You can inject this service, typically in your root component, and customize the values of its properties in order to provide default values for all the timepickers used in the application.

Properties

disabled hourStep meridian minuteStep readonlyInputs seconds secondStep size spinners

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Examples:

Example 1: Timepicker

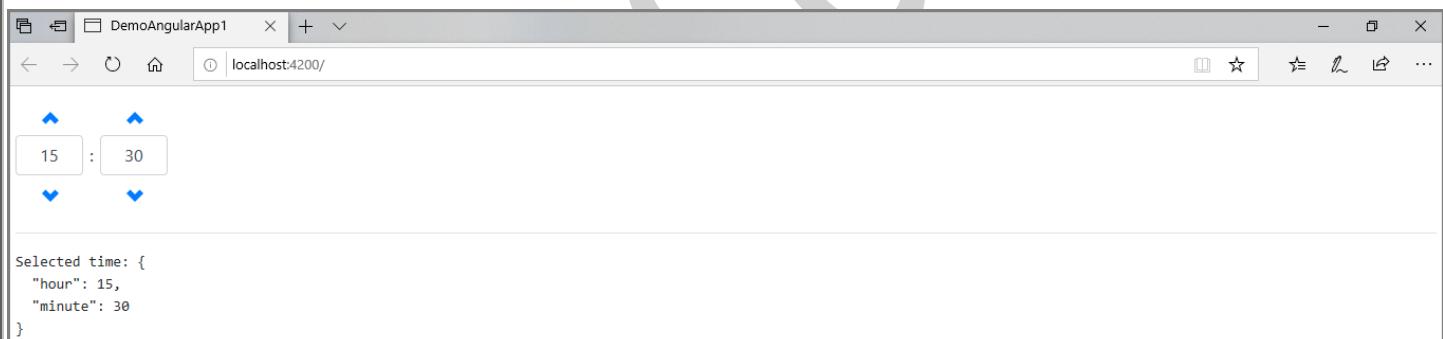
app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  time = {hour: 15, minute: 30};
}
```

app.component.html:

```
<div class="p-2">
  <ngb-timepicker [(ngModel)]="time"></ngb-timepicker>
  <hr>
  <pre>Selected time: {{time | json}}</pre>
</div>
```

Output:



Example 2: Meridian

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  time = {hour: 15, minute: 30};
  meridian = true;

  toggleMeridian() {
    this.meridian = !this.meridian;
  }
}
```

app.component.html:

```

<div class="p-2">
  <ngb-timepicker [(ngModel)]="time" [meridian]="meridian"></ngb-timepicker>
  <button class="btn btn-sm btn-outline-{{meridian ? 'success' : 'danger'}}"
    (click)="toggleMeridian()">
    Meridian - {{meridian ? "ON" : "OFF"}}
  </button>
  <hr>
  <pre>Selected time: {{time | json}}</pre>
</div>

```

Output:



Example 3: Seconds

app.component.ts:

```

import { Component } from '@angular/core';
import { NgbTimeStruct } from '@ng-bootstrap/ng-bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  time: NgbTimeStruct = {hour: 15, minute: 30, second: 30};
  seconds = true;

  toggleSeconds() {
    this.seconds = !this.seconds;
  }
}

```

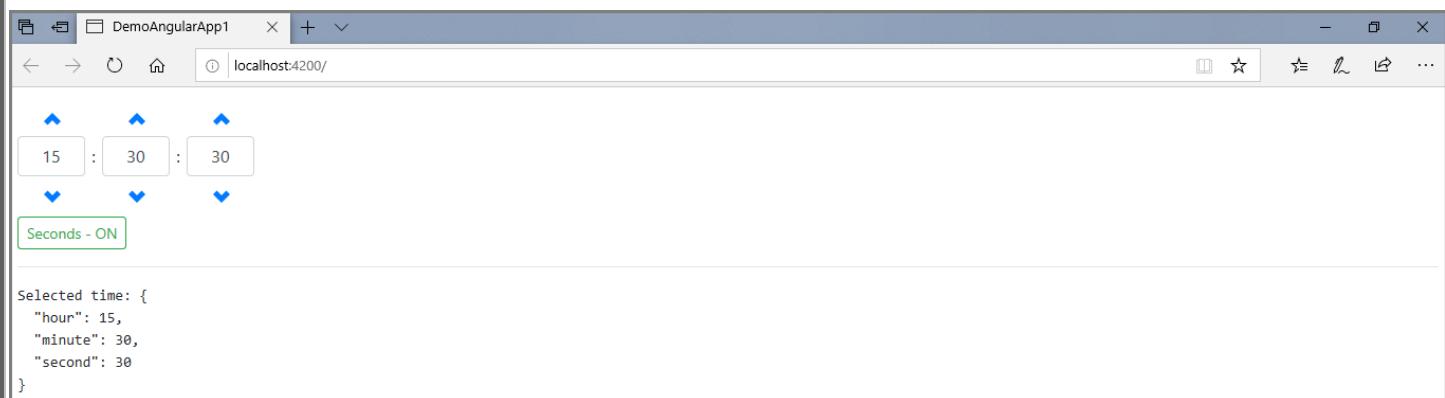
app.component.html:

```

<div class="p-2">
  <ngb-timepicker [(ngModel)]="time" [seconds]="seconds"></ngb-timepicker>
  <button class="btn btn-sm btn-outline-{{seconds ? 'success' : 'danger'}}"
    (click)="toggleSeconds()">
    Seconds - {{seconds ? "ON" : "OFF"}}
  </button>
  <hr>
  <pre>Selected time: {{time | json}}</pre>
</div>

```

Output:



DemoAngularApp1 localhost:4200/

Selected time: {
 "hour": 15,
 "minute": 30,
 "second": 30
}

Example 4: Spinners

app.component.ts:

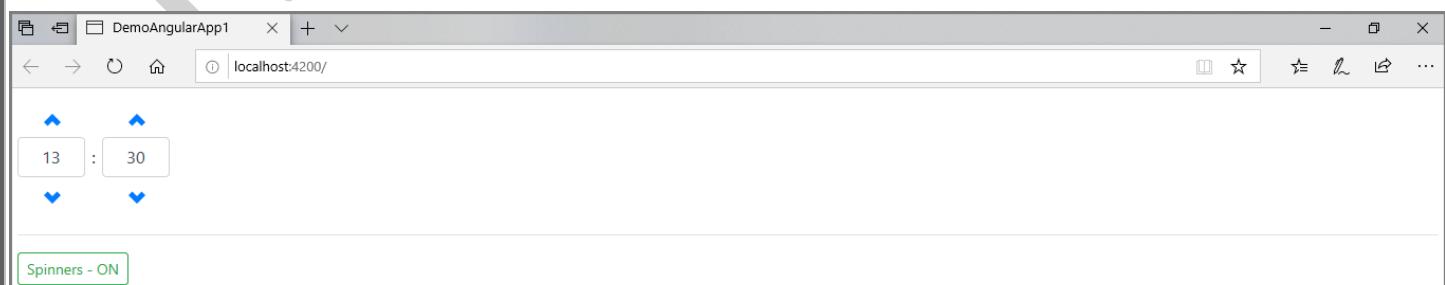
```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  time = {hour: 13, minute: 30};
  spinners = true;

  toggleSpinners() {
    this.spinners = !this.spinners;
  }
}
```

app.component.html:

```
<div class="p-2">
  <ngb-timepicker [(ngModel)]="time" [spinners]="spinners"></ngb-timepicker>
  <hr />
  <button class="m-t-1 btn btn-sm btn-outline-{{spinners ? 'success' : 'danger'}}"
    (click)="toggleSpinners()">
    Spinners - {{spinners ? "ON" : "OFF"}}
  </button>
</div>
```

Output:



DemoAngularApp1 localhost:4200/

Spinners - ON

Example 5: Custom steps

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbTimeStruct } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  time: NgbTimeStruct = {hour: 15, minute: 30, second: 0};
  hourStep = 1;
  minuteStep = 15;
  secondStep = 30;
}
```

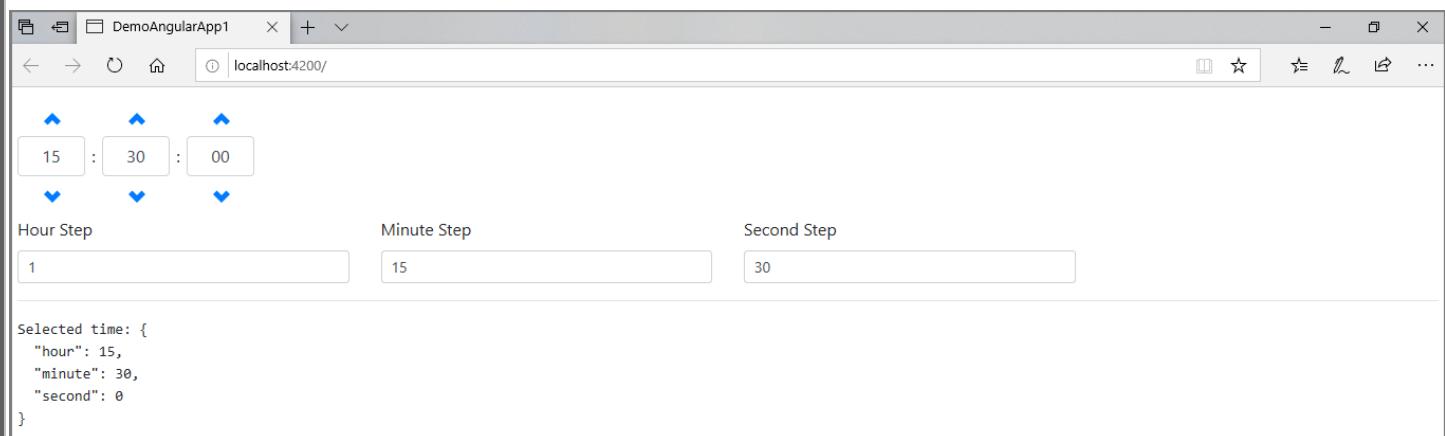
app.component.html:

```
<div class="p-2">
  <ngb-timepicker [(ngModel)]="time"
    [seconds]="true"
    [hourStep]="hourStep"
    [minuteStep]="minuteStep"
    [secondStep]="secondStep">
  </ngb-timepicker>

  <div class="row">
    <div class="col-sm-3">
      <label for="changeHourStep">Hour Step</label>
      <input id="changeHourStep" type="number" class="form-control form-control-sm"
        [(ngModel)]="hourStep" />
    </div>
    <div class="col-sm-3">
      <label for="changeMinuteStep">Minute Step</label>
      <input id="changeMinuteStep" type="number" class="form-control form-control-sm"
        [(ngModel)]="minuteStep" />
    </div>
    <div class="col-sm-3">
      <label for="changeSecondStep">Second Step</label>
      <input id="changeSecondStep" type="number" class="form-control form-control-sm"
        [(ngModel)]="secondStep" />
    </div>
  </div>
  <hr>
  <pre>Selected time: {{time | json}}</pre>
</div>
```



Output:



DemoAngularApp1 localhost:4200/

Hour Step Minute Step Second Step

15 : 30 : 00

Selected time: {
 "hour": 15,
 "minute": 30,
 "second": 0
}

Example 6: Custom validation

Illustrates custom validation, you have to select time between 12:00 and 13:59

app.component.ts:

```

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  ctrl = new FormControl('', (control: FormControl) => {
    const value = control.value;

    if (!value) {
      return null;
    }

    if (value.hour < 12) {
      return {tooEarly: true};
    }
    if (value.hour > 13) {
      return {tooLate: true};
    }

    return null;
  });
}

```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



app.component.html:

```
<div class="p-2">
  <p>Select time between 12:00 and 13:59</p>
  <div class="form-group">
    <ngb-timepicker [FormControl]="ctrl" required></ngb-timepicker>
    <div *ngIf="ctrl.valid" class="small form-text text-success">Great choice</div>
    <div class="small form-text text-danger" *ngIf="!ctrl.valid">
      <div *ngIf="ctrl.errors['required']">Select some time during lunchtime</div>
      <div *ngIf="ctrl.errors['tooLate']">Oh no, it's way too late</div>
      <div *ngIf="ctrl.errors['tooEarly']">It's a bit too early</div>
    </div>
  </div>
  <hr>
  <pre>Selected time: {{ctrl.value | json}}</pre>
</div>
```

Output:

Example 7: Global configuration of timepickers

This timepicker uses customized default values.

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbTimepickerConfig } from '@ng-bootstrap/ng-bootstrap';
import { NgbTimeStruct } from '@ng-bootstrap/ng-bootstrap';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [NgbTimepickerConfig] // add NgbTimepickerConfig to the component providers
})
export class AppComponent {
  time: NgbTimeStruct = {hour: 15, minute: 30, second: 0};

  constructor(config: NgbTimepickerConfig) {
    // customize default values of ratings used by this component tree
    config.seconds = true;
    config.spinners = false;
  }
}
```

app.component.html:

```
<div class="p-2">
  <ngb-timepicker [(ngModel)]="time"></ngb-timepicker>
</div>
```

Output:



17. Toast

NgbToast: Component

Since 5.0.0

Toasts provide feedback messages as notifications to the user. Goal is to mimic the push notifications available both on mobile and desktop operating systems.

Selector: ngb-toast

Exported as: ngbToast

NgbToastHeader: Directive

Since 5.0.0

This directive allows the usage of HTML markup or other directives inside of the toast's header.

Selector: [ngbToastHeader]

NgbToastOptions: Interface

Since 5.0.0

Interface used to type all toast config options.

NgbToastConfig: Configuration

Since 5.0.0

Configuration service for the **NgbToast** component. You can inject this service, typically in your root component, and customize the values of its properties in order to provide default values for all the toasts used in the application.

Examples:

Example 1: Declarative inline usage

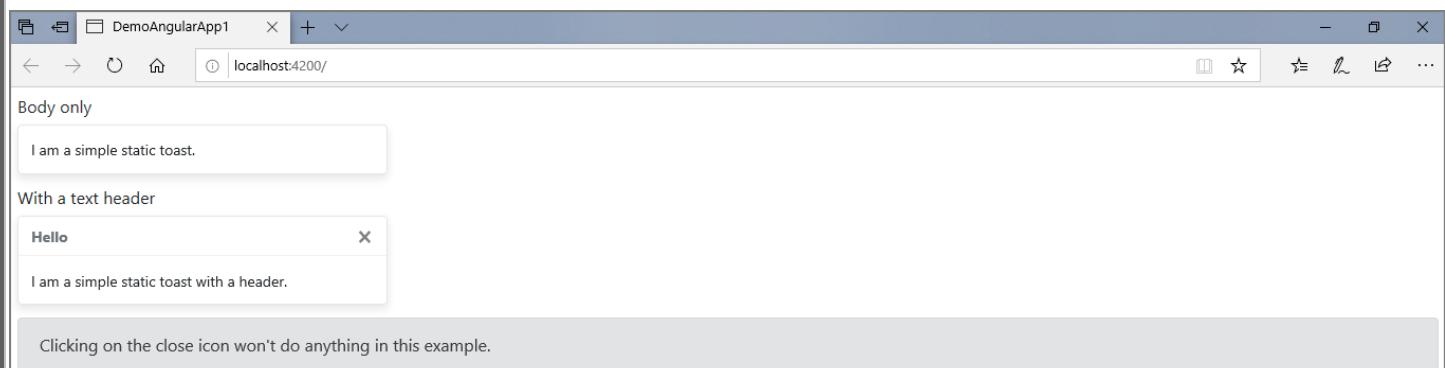
app.component.ts:

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

app.component.html:

```
<div class="p-2">
  <h6>Body only</h6>
  <ngb-toast>
    I am a simple static toast.
  </ngb-toast>
  <h6>With a text header</h6>
  <ngb-toast header="Hello">
    I am a simple static toast with a header.
  </ngb-toast>
  <ngb-alert type="secondary" [dismissible]="false">
    Clicking on the close icon won't do anything in this example.
  </ngb-alert>
</div>
```

Output:



Body only

I am a simple static toast.

With a text header

Hello

I am a simple static toast with a header.

Clicking on the close icon won't do anything in this example.

Example 2: Using a Template as header

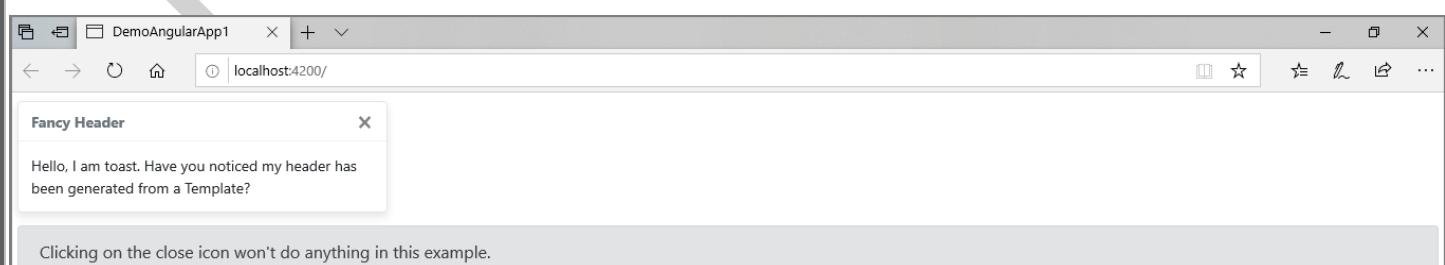
app.component.ts:

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

app.component.html:

```
<div class="p-2">
  <ngb-toast>
    <ng-template ngbToastHeader>
      <strong>Fancy Header</strong>
    </ng-template>
    Hello, I am toast. Have you noticed my header has been generated from a Template?
  </ngb-toast>
  <ngb-alert type="secondary" [dismissible]="false">
    Clicking on the close icon won't do anything in this example.
  </ngb-alert>
</div>
```

Output:



Fancy Header

Hello, I am toast. Have you noticed my header has been generated from a Template?

Clicking on the close icon won't do anything in this example.

Example 3: Closeable toast

app.component.ts:

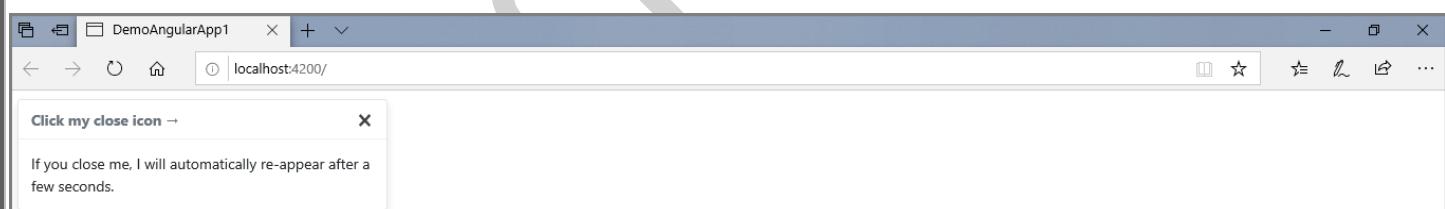
```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  show = true;

  close() {
    this.show = false;
    setTimeout(() => this.show = true, 5000);
  }
}
```

app.component.html:

```
<div class="p-2">
  <ngb-toast *ngIf="show" header="Click my close icon →"
    [autohide]="false" (hide)="close()">
    If you close me, I will automatically re-appear after a few seconds.
  </ngb-toast>
  <p *ngIf="!show">I'll be back!</p>
</div>
```

Output:



Example 4: Prevent autohide on mouseover

In this example, we can show a toast by clicking the button below. It will hide itself after a 5 seconds delay unless you simply hover it with your mouse.

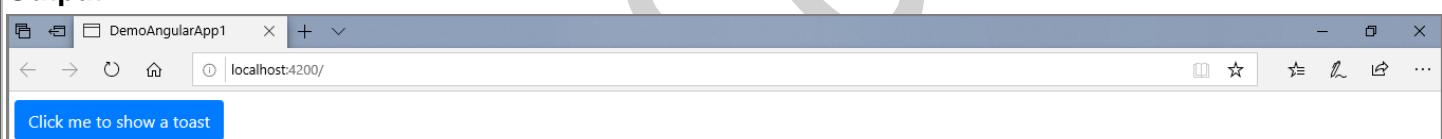
app.component.ts:

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

app.component.html:

```
<div class="p-2">
  <button class="btn btn-primary" (click)="show = true">
    Click me to show a toast
  </button>
  <hr *ngIf="show" />
  <ngb-toast *ngIf="show" header="Autohide can be cancelled"
    [delay]="5000" [autohide]="autohide"
    (mouseenter)="autohide = false"
    (mouseleave)="autohide = true"
    (hide)="show = false; autohide = true"
    [class.bg-warning]="!autohide">
    <div *ngIf="autohide">
      Try to mouse hover me.
    </div>
    <div *ngIf="!autohide">
      I will remain visible until you leave again.
    </div>
  </ngb-toast>
</div>
```

Output:



In the above examples, these are the following properties & events used:

autohide	Auto hide the toast after a delay in ms. default: <code>true</code> (inherited from <code>NgbToastConfig</code>) <code>Type: boolean</code> <code>Default value: true</code> — initialized from <code>NgbToastConfig</code> service
delay	Delay after which the toast will hide (ms). default: <code>500</code> (ms) (inherited from <code>NgbToastConfig</code>) <code>Type: number</code> <code>Default value: 500</code> — initialized from <code>NgbToastConfig</code> service
header	Text to be used as toast's header. Ignored if a <code>ContentChild</code> template is specified at the same time. <code>Type: string</code>
hide	An event fired immediately when toast's <code>hide()</code> method has been called. It can only occur in 2 different scenarios: <ul style="list-style-type: none"> • <code>autohide</code> timeout fires • user clicks on a closing cross (&times;) Additionally this output is purely informative. The toast won't disappear. It's up to the user to take care of that.

18. Tooltip

NgbTooltip: Directive

A lightweight and extensible directive for fancy tooltip creation.

Selector: [ngbTooltip]

Exported as: ngbTooltip

Properties, Methods & Events:

autoClose since 3.0.0	Indicates whether the tooltip should be closed on <code>Escape</code> key and inside/outside clicks: <ul style="list-style-type: none"> <code>true</code> - closes on both outside and inside clicks as well as <code>Escape</code> presses <code>false</code> - disables the <code>autoClose</code> feature (NB: triggers still apply) <code>"inside"</code> - closes on inside clicks as well as <code>Escape</code> presses <code>"outside"</code> - closes on outside clicks (sometimes also achievable through triggers) as well as <code>Escape</code> presses Type: boolean 'inside' 'outside' Default value: true — initialized from NgbTooltipConfig service
closeDelay since 4.1.0	The closing delay in ms. Works only for "non-manual" opening triggers defined by the <code>triggers</code> input. Type: number Default value: 0 — initialized from NgbTooltipConfig service
container	A selector specifying the element the tooltip should be appended to. Currently only supports " <code>body</code> ". Type: string Default value: - — initialized from NgbTooltipConfig service
disableTooltip since 1.1.0	If <code>true</code> , tooltip is disabled and won't be displayed. Type: boolean Default value: false — initialized from NgbTooltipConfig service
ngbTooltip	The string content or a <code>TemplateRef</code> for the content to be displayed in the tooltip. If the content is falsy, the tooltip won't open. Type: string TemplateRef<any>
openDelay since 4.1.0	The opening delay in ms. Works only for "non-manual" opening triggers defined by the <code>triggers</code> input. Type: number Default value: 0 — initialized from NgbTooltipConfig service
placement	The preferred placement of the tooltip. Possible values are <code>"top"</code> , <code>"top-left"</code> , <code>"top-right"</code> , <code>"bottom"</code> , <code>"bottom-left"</code> , <code>"bottom-right"</code> , <code>"left"</code> , <code>"left-top"</code> , <code>"left-bottom"</code> , <code>"right"</code> , <code>"right-top"</code> , <code>"right-bottom"</code> Accepts an array of strings or a string with space separated possible values. The default order of preference is <code>"auto"</code> (same as the sequence above). Type: PlacementArray Default value: auto — initialized from NgbTooltipConfig service
tooltipClass since 3.2.0	An optional class applied to the tooltip window element. Type: string Default value: - — initialized from NgbTooltipConfig service
triggers	Specifies events that should trigger the tooltip. Supports a space separated list of event names. Type: string Default value: hover focus — initialized from NgbTooltipConfig service

hidden	An event emitted when the popover is hidden. Contains no payload.
shown	An event emitted when the tooltip is shown. Contains no payload.
open	<code>open(context: any) => void</code> Opens the tooltip. This is considered to be a "manual" triggering. The context is an optional value to be injected into the tooltip template when it is created.
close	<code>close() => void</code> Closes the tooltip. This is considered to be a "manual" triggering of the tooltip.
toggle	<code>toggle() => void</code> Toggles the tooltip. This is considered to be a "manual" triggering of the tooltip.
isOpen	<code>isOpen() => boolean</code> Returns true, if the popover is currently shown.

NgbTooltipConfig: Configuration

A configuration service for the **NgbTooltip** component.

You can inject this service, typically in your root component, and customize the values of its properties in order to provide default values for all the tooltips used in the application.

Properties:

autoClose closeDelay container disableTooltip openDelay placement tooltipClass triggers

Examples:

Example 1: Quick and easy tooltips

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
}
app.component.html:
<div class="p-2" style="margin-top: 30px;">
  <button type="button" class="btn btn-outline-secondary mr-2"
    placement="top" ngbTooltip="Tooltip on top">
    Tooltip on top
  </button>
  <button type="button" class="btn btn-outline-secondary mr-2"
    placement="right" ngbTooltip="Tooltip on right">
    Tooltip on right
  </button>
  <button type="button" class="btn btn-outline-secondary mr-2"
    placement="bottom" ngbTooltip="Tooltip on bottom">
    Tooltip on bottom
  </button>
  <button type="button" class="btn btn-outline-secondary mr-2"
    placement="left" ngbTooltip="Tooltip on left">
    Tooltip on left
  </button>
</div>
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Output:



Example 2: HTML and bindings in tooltips

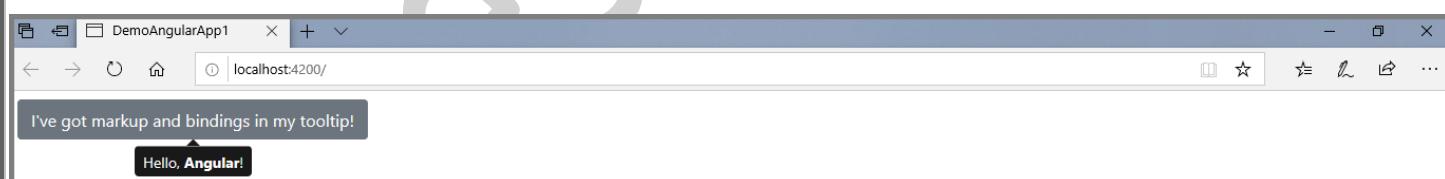
app.component.ts:

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

app.component.html:

```
<div class="p-2">
  <ng-template #tipContent>Hello, <b>{{name}}</b>!</ng-template>
  <button type="button" class="btn btn-outline-secondary" [ngbTooltip]="tipContent">
    I've got markup and bindings in my tooltip!
  </button>
</div>
```

Output:



Example 3: Custom and manual triggers

app.component.ts:

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



app.component.html:

```
<div class="p-2">
  <p>
    You can easily override open and close triggers by specifying event names (separated
    by <code>:</code>) in the
    <code>triggers</code> property.
  </p>
  <button type="button" class="btn btn-outline-secondary"
    ngbTooltip="You see, I show up on click!"
    triggers="click:blur">
    Click me!
  </button>
  <hr />
  <p>
    Alternatively you can take full manual control over tooltip opening / closing events.
  </p>
  <button type="button" class="btn btn-outline-secondary mr-2"
    ngbTooltip="What a great tip!" [autoClose]="false"
    triggers="manual" #t="ngbTooltip" (click)="t.open()">
    Click me to open a tooltip
  </button>
  <button type="button" class="btn btn-outline-secondary mr-2" (click)="t.close()">
    Click me to close a tooltip
  </button>
</div>
```

Output:

The screenshot shows a browser window titled "DemoAngularApp1" with the URL "localhost:4200/" in the address bar. The page content is as follows:

You can easily override open and close triggers by specifying event names (separated by :) in the triggers property.

Alternatively you can take full manual control over tooltip opening / closing events.

Example 4: Automatic closing with keyboard and mouse

app.component.ts:

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

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.component.html:

```
<div class="p-2">
  <p>As for some other popup-based widgets,  

    you can set the tooltip to close automatically upon some events.</p>
  <p>In the following examples, they will all close on <code>Escape</code> as well as:</p>
  <ul>
    <li class="mb-2">
      click inside:  

      <button type="button" class="btn btn-outline-secondary" triggers="click"  

          [autoClose]="'inside'"  

          ngbTooltip="Click inside or press Escape to close">  

        Click to toggle
      </button>
    </li>
    <li class="mb-2">
      click outside:  

      <button type="button" class="btn btn-outline-secondary" triggers="click"  

          [autoClose]="'outside'"  

          ngbTooltip="Click outside or press Escape to close">  

        Click to toggle
      </button>
    </li>
    <li class="mb-2">
      all clicks:  

      <button type="button" class="btn btn-outline-secondary" triggers="click"  

          [autoClose]="true"  

          ngbTooltip="Click anywhere or press Escape to close (try the toggling element too)"  

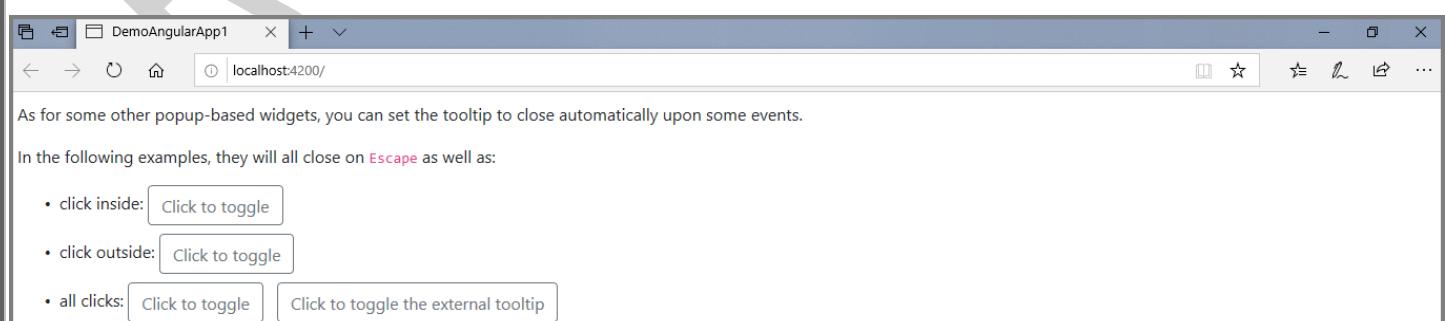
          #tooltip3="ngbTooltip">  

        Click to toggle
      </button>
      &nbsp;
      <button type="button" class="btn btn-outline-secondary mr-2"  

          (click)="tooltip3.toggle()">  

        Click to toggle the external tooltip
      </button>
    </li>
  </ul>
</div>
```

Output:



As for some other popup-based widgets, you can set the tooltip to close automatically upon some events.

In the following examples, they will all close on `Escape` as well as:

- click inside: Click to toggle
- click outside: Click to toggle
- all clicks: Click to toggle Click to toggle the external tooltip

Example 5: Context and manual triggers

You can optionally pass in a context when manually triggering a tooltip.

app.component.ts:

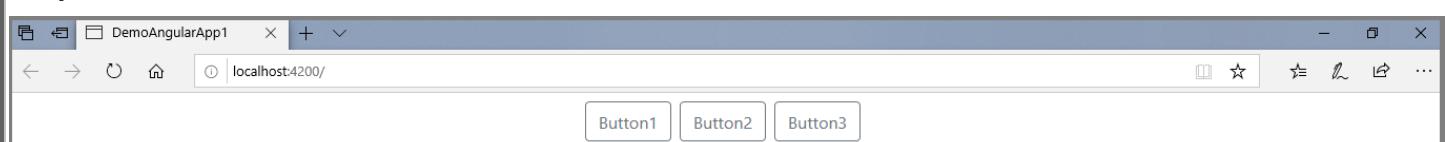
```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  name = 'Angular';

  toggleWithGreeting(tooltip, greeting: string) {
    if (tooltip.isOpen()) {
      tooltip.close();
    } else {
      tooltip.open({greeting});
    }
  }
}
```

app.component.html:

```
<div class="p-2" style="text-align: center;">
  <ng-template #tipContent let-greeting="greeting">{{greeting}}, <b>{{name}}</b>!</ng-
template>
  <button type="button" class="btn btn-outline-secondary mr-2"
    [ngbTooltip]="tipContent" triggers="manual"
    #t1="ngbTooltip" (click)="toggleWithGreeting(t1, 'Button1')">
    Button1
  </button>
  <button type="button" class="btn btn-outline-secondary mr-2"
    [ngbTooltip]="tipContent" triggers="manual"
    #t2="ngbTooltip" (click)="toggleWithGreeting(t2, 'Button2')">
    Button2
  </button>
  <button type="button" class="btn btn-outline-secondary mr-2"
    [ngbTooltip]="tipContent" triggers="manual"
    #t3="ngbTooltip" (click)="toggleWithGreeting(t3, 'Button3')">
    Button3
  </button>
</div>
```

Output:



Example 6: Open and close delays

When using non-manual triggers, you can control the delay to open and close the tooltip through the `openDelay` and `closeDelay` properties. Note that the `autoClose` feature does not use the close delay, it closes the tooltip immediately.

app.component.ts:

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

app.component.html:

```
<div class="p-2">
  <button type="button" class="btn btn-outline-secondary mr-2"
    ngbTooltip="You see, I show up after 300ms and disappear after 500ms!"
    [openDelay]="300" [closeDelay]="500">
    Hover 300ms here
  </button>
  <button type="button" class="btn btn-outline-secondary mr-2"
    ngbTooltip="You see, I show up after 1s and disappear after 2s!"
    [openDelay]="1000" [closeDelay]="2000">
    Hover 1s here
  </button>
</div>
```

Output:



Example 7: Tooltip with custom class

You can optionally pass in a custom class via `tooltipClass`

app.component.ts:

```
import { Component, ViewEncapsulation } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  encapsulation: ViewEncapsulation.None,
  styles: [
    '.my-custom-class .tooltip-inner {
      background-color: darkgreen;
      font-size: 125%;

    }
    .my-custom-class .arrow::before {
      border-top-color: darkgreen;
    }
  ]
})
export class AppComponent {}
```

app.component.html:

```
<div class="p-2">
  <button type="button" class="btn btn-outline-secondary"
    ngbTooltip="Nice class!" tooltipClass="my-custom-class">
    Tooltip with custom class
  </button>
</div>
```

Output:



Example 8: Global configuration of tooltips

app.component.ts:

```
import { Component } from '@angular/core';
import { NgbTooltipConfig } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [NgbTooltipConfig] // add NgbTooltipConfig to the component providers
})
export class AppComponent {
  constructor(config: NgbTooltipConfig) {
    // customize default values of tooltips used by this component tree
    config.placement = 'right';
    config.triggers = 'click';
  }
}
```

app.component.html:

```
<div class="p-2">
  <button type="button" class="btn btn-outline-secondary"
    ngbTooltip="This tooltip gets its inputs from the customized configuration">
    Customized tooltip
  </button>
</div>
```

Output:



19. Typeahead

NgbHighlight: Component

A component that helps with text highlighting.

If splits the result text into parts that contain the searched term and generates the HTML markup to simplify highlighting:

Ex. result="Alaska" and term="as" will produce Alaska.

Selector: nbg-highlight



highlightClass	The CSS class for <code></code> elements wrapping the <code>term</code> inside the <code>result</code> .
	Type: string
	Default value: nbg-highlight
result	<p>The text highlighting is added to.</p> <p>If the <code>term</code> is found inside this text, it will be highlighted. If the <code>term</code> contains array then all the items from it will be highlighted inside the text.</p>
	Type: string
term	<p>The term or array of terms to be highlighted. Since version v4.2.0 term could be a <code>string[]</code></p>
	Type: string readonly string[]

NgbTypeahead: Directive

A directive providing a simple way of creating powerful typeheads from any text input.

Selector: input[nbgTypeahead]

Exported as: nbgTypeahead

autocomplete since 2.1.0	<p>The value for the <code>autocomplete</code> attribute for the <code><input></code> element.</p> <p>Defaults to "off" to disable the native browser autocomplete, but you can override it if necessary.</p>
	Type: string
	Default value: off
container	<p>A selector specifying the element the typeahead popup will be appended to.</p> <p>Currently only supports "body".</p>
	Type: string
	Default value: - — initialized from NgbTypeaheadConfig service
editable	<p>If <code>true</code>, model values will not be restricted only to items selected from the popup.</p>
	Type: boolean
	Default value: true — initialized from NgbTypeaheadConfig service
focusFirst	<p>If <code>true</code>, the first item in the result list will always stay focused while typing.</p>
	Type: boolean
	Default value: true — initialized from NgbTypeaheadConfig service

inputFormatter

The function that converts an item from the result list to a `string` to display in the `<input>` field.

It is called when the user selects something in the popup or the model value changes, so the input needs to be updated.

Type: `(item: any) => string`

ngbTypeahead

The function that converts a stream of text values from the `<input>` element to the stream of the array of items to display in the typeahead popup.

If the resulting observable emits a non-empty array - the popup will be shown. If it emits an empty array - the popup will be closed.

Note that the `this` argument is `undefined` so you need to explicitly bind it to a desired "this" target.

Type: `(text: Observable<string>) => Observable<readonly any[]>`

placement

The preferred placement of the typeahead.

Possible values are `"top"`, `"top-left"`, `"top-right"`, `"bottom"`, `"bottom-left"`, `"bottom-right"`, `"left"`, `"left-top"`, `"left-bottom"`, `"right"`, `"right-top"`, `"right-bottom"`

Accepts an array of strings or a string with space separated possible values.

The default order of preference is `"bottom-left bottom-right top-left top-right"`

Type: `PlacementArray`

Default value: - — initialized from `NgbTypeaheadConfig` service

resultFormatter

The function that converts an item from the result list to a `string` to display in the popup.

Must be provided, if your `ngbTypeahead` returns something other than `Observable<string[]>`.

Alternatively for more complex markup in the popup you should use `resultTemplate`.

Type: `(item: any) => string`

resultTemplate

The template to override the way resulting items are displayed in the popup.

Type: `TemplateRef<ResultTemplateContext>`

showHint

If `true`, will show the hint in the `<input>` when an item in the result list matches.

Type: `boolean`

Default value: `false` — initialized from `NgbTypeaheadConfig` service

selectItem

An event emitted right before an item is selected from the result list.

Event payload is of type `NgbTypeaheadSelectItemEvent`.

dismissPopup

`dismissPopup() => void`

Dismisses typeahead popup window

isPopupOpen

`isPopupOpen() => void`

Returns true if the typeahead popup window is displayed



NgbTypeaheadSelectItemEvent: Interface

An event emitted right before an item is selected from the result list.

item The item from the result list about to be selected.

Type: any

preventDefault Calling this function will prevent item selection from happening.

Type: () => void

NgbTypeaheadConfig: Configuration

A configuration service for the **NgbTypeahead** component.

You can inject this service, typically in your root component, and customize the values of its properties in order to provide default values for all the typeheads used in the application.

Properties

container editable focusFirst placement showHint

Examples:

Example 1: Simple Typeahead

A typeahead example that gets values from a static string[]

- debounceTime operator
- kicks in only if 2+ characters typed
- limits to 10 results

app.component.ts:

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';
import { debounceTime, distinctUntilChanged, map } from 'rxjs/operators';

const states = ['Alabama', 'Alaska', 'American Samoa', 'Arizona', 'Arkansas', 'California',
  'Colorado', 'Connecticut', 'Delaware', 'District Of Columbia',
  'Federated States Of Micronesia', 'Florida', 'Georgia', 'Guam', 'Hawaii',
  'Idaho', 'Illinois', 'Indiana', 'Iowa', 'Kansas', 'Kentucky', 'Louisiana',
  'Maine', 'Marshall Islands', 'Maryland', 'Massachusetts', 'Michigan',
  'Minnesota', 'Mississippi', 'Missouri', 'Montana', 'Nebraska', 'Nevada',
  'New Hampshire', 'New Jersey', 'New Mexico', 'New York', 'North Carolina',
  'North Dakota', 'Northern Mariana Islands', 'Ohio', 'Oklahoma', 'Oregon',
  'Palau', 'Pennsylvania', 'Puerto Rico', 'Rhode Island', 'South Carolina',
  'South Dakota', 'Tennessee', 'Texas', 'Utah', 'Vermont', 'Virgin Islands',
  'Virginia', 'Washington', 'West Virginia', 'Wisconsin', 'Wyoming'
];


```

```
@Component({
  selector: "app-root",
  templateUrl: "./app.component.html",
  styles: [`.form-control { width: 300px; }`]
})
```

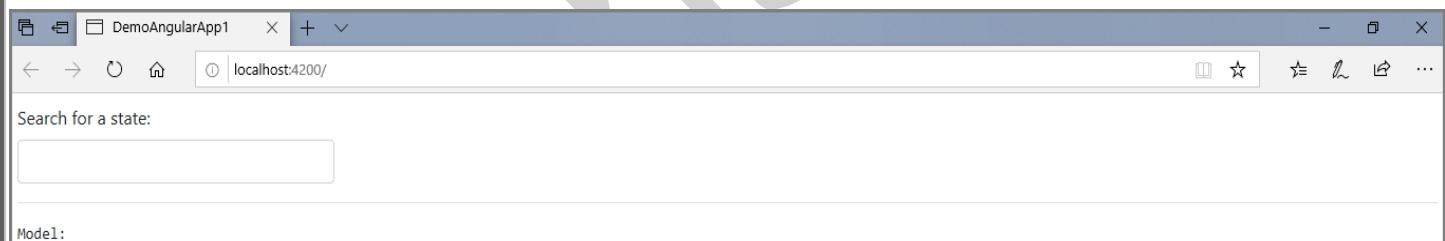
```
export class AppComponent {
  public model: any;

  search = (text$: Observable<string>) =>
    text$.pipe(
      debounceTime(200),
      distinctUntilChanged(),
      map(term => term.length < 2 ? []
        : states.filter(v => v.toLowerCase().indexOf(term.toLowerCase()) > -1).slice(0, 10))
    )
}
```

app.component.html:

```
<div class="p-2">
  <label for="typeahead-basic">Search for a state:</label>
  <input id="typeahead-basic" type="text" class="form-control"
    [(ngModel)]="model" [ngbTypeahead]="search" />
  <hr />
  <pre>Model: {{ model | json }}</pre>
</div>
```

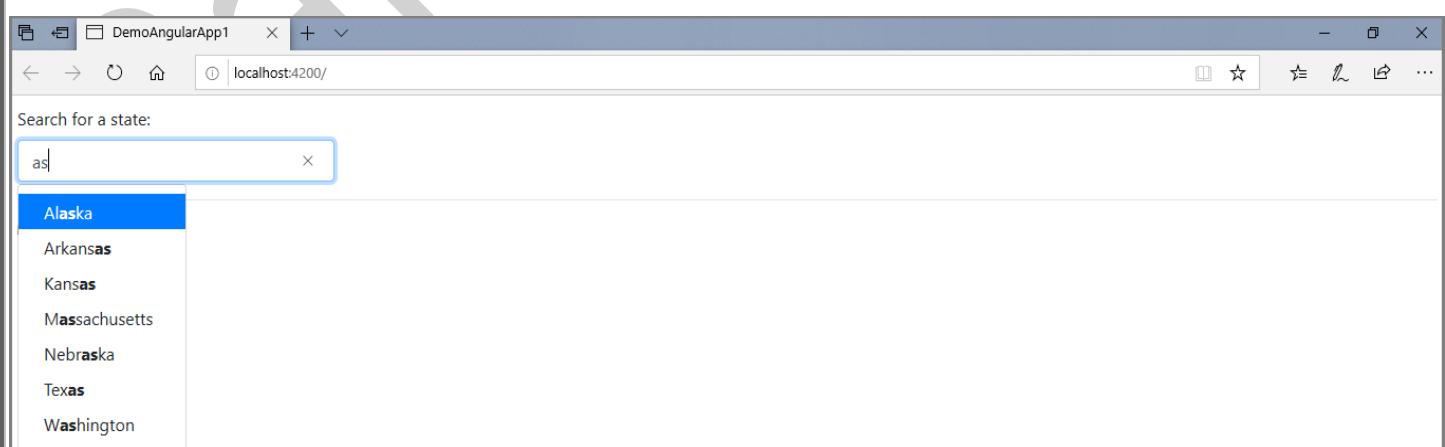
Output:



Search for a state:

Model:

Now type any 2+ characters in a given search text box to show the list of matching states along with highlighted typed characters:



Search for a state:

- Alaska
- Arkansas
- Kansas
- Massachusetts
- Nebraska
- Texas
- Washington



Example 2: Open on focus

It is possible to get the focus events with the current input value to emit results on focus with a great flexibility. In this simple example, a search is done no matter the content of the input:

- on empty input all options will be taken
- otherwise options will be filtered against the search term
- it will limit the display to 10 results in all cases

app.component.ts:

```
import {Component, ViewChild} from '@angular/core';
import {NgbTypeahead} from '@ng-bootstrap/ng-bootstrap';
import {Observable, Subject, merge} from 'rxjs';
import {debounceTime, distinctUntilChanged, filter, map} from 'rxjs/operators';

const states = ['Alabama', 'Alaska', 'American Samoa', 'Arizona', 'Arkansas', 'California',
    'Colorado', 'Connecticut', 'Delaware', 'District Of Columbia',
    'Federated States Of Micronesia', 'Florida', 'Georgia', 'Guam', 'Hawaii',
    'Idaho', 'Illinois', 'Indiana', 'Iowa', 'Kansas', 'Kentucky', 'Louisiana',
    'Maine', 'Marshall Islands', 'Maryland', 'Massachusetts', 'Michigan',
    'Minnesota', 'Mississippi', 'Missouri', 'Montana', 'Nebraska', 'Nevada',
    'New Hampshire', 'New Jersey', 'New Mexico', 'New York', 'North Carolina',
    'North Dakota', 'Northern Mariana Islands', 'Ohio', 'Oklahoma', 'Oregon',
    'Palau', 'Pennsylvania', 'Puerto Rico', 'Rhode Island', 'South Carolina',
    'South Dakota', 'Tennessee', 'Texas', 'Utah', 'Vermont', 'Virgin Islands',
    'Virginia', 'Washington', 'West Virginia', 'Wisconsin', 'Wyoming'
];

@Component({
  selector: "app-root",
  templateUrl: "./app.component.html",
  styles: [`.form-control { width: 300px; }`]
})
export class AppComponent {
  model: any;

  @ViewChild('instance', {static: true}) instance: NgbTypeahead;
  focus$ = new Subject<string>();
  click$ = new Subject<string>();

  search = (text$: Observable<string>) => {
    const debouncedText$ = text$.pipe(debounceTime(200), distinctUntilChanged());
    const clicksWithClosedPopup$ = this.click$.pipe(filter(() => !this.instance.isPopupOpen()));
    const inputFocus$ = this.focus$;

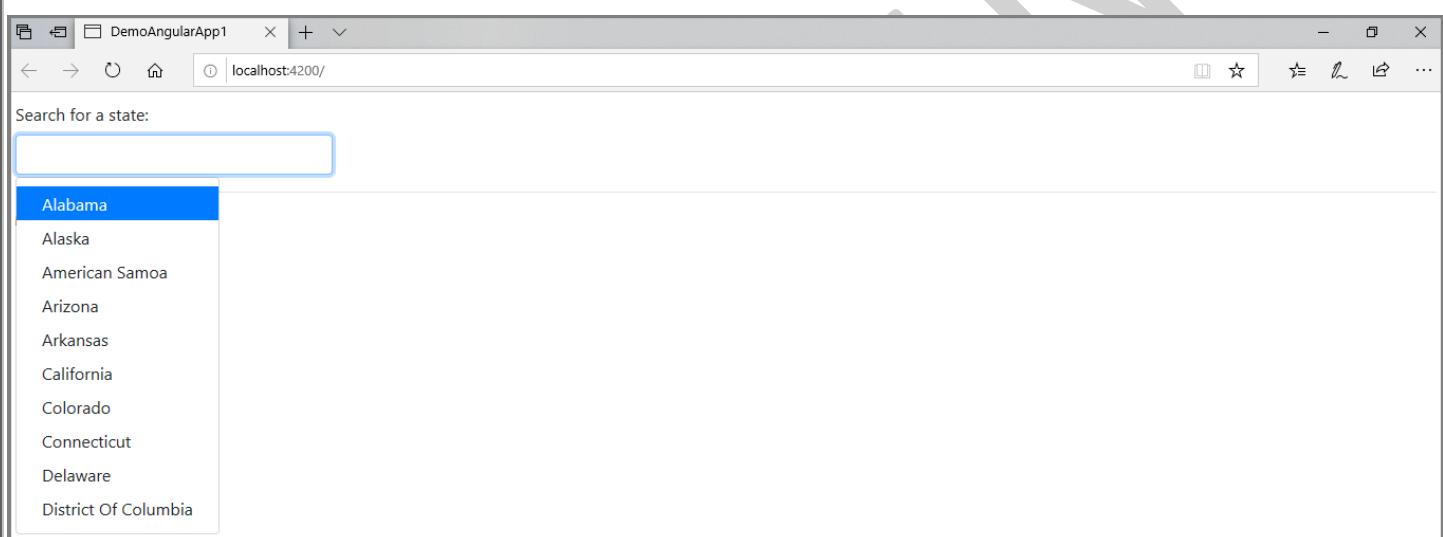
    return merge(debouncedText$, inputFocus$, clicksWithClosedPopup$).pipe(
      map(term => (term === '' ? states
        : states.filter(v => v.toLowerCase().indexOf(term.toLowerCase()) > -1)).slice(0, 10))
    );
  }
}
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

app.component.html:

```
<div class="p-2">
  <label for="typeahead-focus">Search for a state:</label>
  <input id="typeahead-focus" type="text" class="form-control"
    [(ngModel)]="model"
    [ngbTypeahead]="search"
    (focus)="focus$.next($event.target.value)"
    (click)="click$.next($event.target.value)"
    #instance="ngbTypeahead" />
  <hr>
  <pre>Model: {{ model | json }}</pre>
</div>
```

Output:



Example 3: Formatted results

A typeahead example that uses a formatter function for string results

app.component.ts:

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';
import { debounceTime, distinctUntilChanged, map } from 'rxjs/operators';

const states = ['Alabama', 'Alaska', 'American Samoa', 'Arizona', 'Arkansas', 'California',
  'Colorado', 'Connecticut', 'Delaware', 'District Of Columbia',
  'Federated States Of Micronesia', 'Florida', 'Georgia', 'Guam', 'Hawaii',
  'Idaho', 'Illinois', 'Indiana', 'Iowa', 'Kansas', 'Kentucky', 'Louisiana',
  'Maine', 'Marshall Islands', 'Maryland', 'Massachusetts', 'Michigan',
  'Minnesota', 'Mississippi', 'Missouri', 'Montana', 'Nebraska', 'Nevada',
  'New Hampshire', 'New Jersey', 'New Mexico', 'New York', 'North Carolina',
  'North Dakota', 'Northern Mariana Islands', 'Ohio', 'Oklahoma', 'Oregon',
  'Palau', 'Pennsylvania', 'Puerto Rico', 'Rhode Island', 'South Carolina',
  'South Dakota', 'Tennessee', 'Texas', 'Utah', 'Vermont', 'Virgin Islands',
  'Virginia', 'Washington', 'West Virginia', 'Wisconsin', 'Wyoming'
];
```

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

```

@Component({
  selector: "app-root",
  templateUrl: "./app.component.html",
  styles: [`.form-control { width: 300px; }`]
})
export class AppComponent {
  public model: any;

  formatter = (result: string) => result.toUpperCase();

  search = (text$: Observable<string>) =>
    text$.pipe(
      debounceTime(200),
      distinctUntilChanged(),
      map(term => term === '' ? []
        : states.filter(v => v.toLowerCase().indexOf(term.toLowerCase()) > -1).slice(0, 10))
    )
}

```

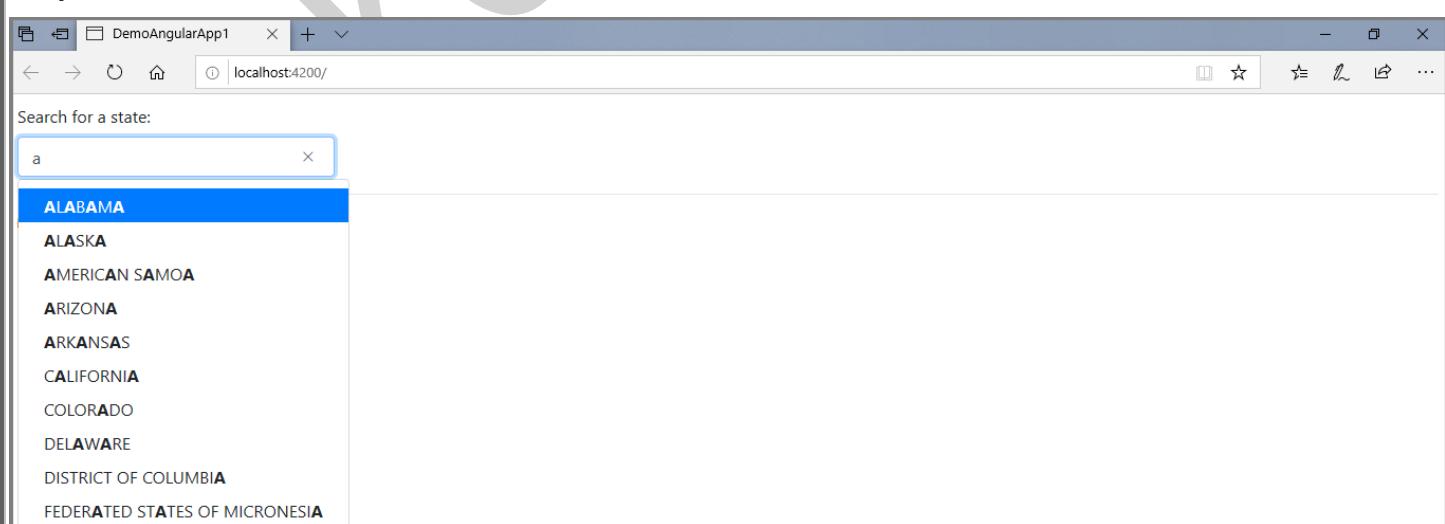
app.component.html:

```

<div class="p-2">
  <label for="typeahead-format">Search for a state:</label>
  <input id="typeahead-format" type="text" class="form-control"
    [(ngModel)]="model"
    [ngbTypeahead]="search"
    [resultFormatter]="formatter" />
  <hr>
  <pre>Model: {{ model | json }}</pre>
</div>

```

Output:



DemoAngularApp1

localhost:4200/

Search for a state:

a

- ALABAMA
- ALASKA
- AMERICAN SAMOA
- ARIZONA
- ARKANSAS
- CALIFORNIA
- COLORADO
- DELAWARE
- DISTRICT OF COLUMBIA
- FEDERATED STATES OF MICRONESIA

Working with ngx-bootstrap:

Working with Bootstrap 4 Components with Angular using ngx-bootstrap. It contains all the core Bootstrap components powered by Angular. So you don't need to include original JS components, but we are using markup and CSS provided by Bootstrap.

Installation of ngx bootstrap

Using npm to install ngx bootstrap:

Open the VS Code terminal and write the below command to install ngx-bootstrap.

➤ **npm install ngx-bootstrap --save**

Now, ngx-bootstrap (ngx-bootstrap v5.3.2) is installed successfully.

Once installed you need to import ngx-bootstrap's required module from the package 'ngx-bootstrap' in root module.

Add the necessary packages required to NgModule (RootModule) imports:

Let's add our ngx-bootstrap components and that will be added to the root module. We first import the specific module for a specific component and then declare in the @NgModule.

For Example: Tooltip Component

AppModule (RootModule):

app.module.ts:

```
import { TooltipModule } from 'ngx-bootstrap/tooltip';
/* ... */

@NgModule({
  declarations: [/* ... */],
  imports: [/* ... */, TooltipModule.forRoot()],
  /* ... */
})

export class AppModule {}
```

Note: You need to import the module for each component you want to use in the same way. ngx-bootstrap provides each bootstrap component in each own module so you only import the components you need. In this way your app will be smaller since it bundles only the components you are actually using.

You can find all the available components that you can use from the <https://valor-software.com/ngx-bootstrap/>

Accordion	Alerts
Buttons	Carousel
Collapse	Datepicker
Dropdowns	Modals
Pagination	Popover
Progressbar	Rating
Sortable	Tabs
Timepicker	Tooltip
Typeahead	



Please note that **ngx-bootstrap** requires the Bootstrap 4 CSS file to be present.

You can add it in the **styles** array of the **angular.json** file like that:

```
"styles": [
    "src/styles.css",
    "node_modules/bootstrap/dist/css/bootstrap.min.css"
]
```

Or you can also add directly Bootstrap 4 CSS file local reference in **<head>** section of **index.html** as following:

index.html:

```
<head>
  <link rel="stylesheet" href="assets/bootstrap/dist/css/bootstrap.css"/>
</head>
```

Note: Make sure bootstrap should be located in **assets** folder to use local reference path in **index.html**

Or Adding Bootstrap 4 CSS file to Angular Using styles.css:

We can also use the **styles.css** file to add the CSS file of Bootstrap to our project.

Open the **src/styles.css** file of your Angular project and import the **bootstrap.css** file as follows:

```
@import "~bootstrap/dist/css/bootstrap.css";
```

OR

```
@import url(~bootstrap/dist/css/bootstrap.css);
```

This replaces the previous method(s) so you don't need to add the file to the **styles** array of the **angular.json** file or to the **index.html** file.

Or Adding Bootstrap 4 CSS file to Angular Using CDN:

You can also add directly Bootstrap 4 CSS file reference using **CDN** in **<head>** section of **index.html** as following:

index.html:

```
<head>
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css" />
</head>
```

Now we're ready to make use of **ngx-bootstrap** component in one of our component templates. Let's try it out by opening file **src/app/app.component.html** and insert the following HTML template code:

app.component.html: (Example of Tooltip)

```
<div class="p-2">
  <button type="button" class="btn btn-primary"
    tooltip="This is a Button Element"
    placement="right">
    Simple Demo
  </button>
</div>
```

Output:



Method 2

Use the Angular CLI **ng add** command for updating your Angular project

- **ng add ngx-bootstrap**

Or use **ng add** to add needed component (for example tooltip)

- **ng add ngx-bootstrap --component tooltip**

The most important difference between **ng-bootstrap** vs. **ngx-bootstrap** is that **ngx-bootstrap** uses separate modules for components to reduce the final app size.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.



Animations in Angular:

Introduction to Angular Animations:

Animation is defined as the transition from an initial state to a final state. It is an integral part of any modern web application. Animation not only helps us to create a great UI but it also makes the application interesting and fun to use. A well-structured animation keeps the user glued to the application and enhances the user experience.

Angular allows us to create animations, which provides us with native-like performance. Here, we will learn how we can create animations using Angular 8.

Understanding Angular Animation States:

Animation involves transition from one state of an element to another state. Angular defines three different states for an element:

- Void state:** The void state represents the state of an element which is not a part of the DOM. This state occurs when an element is created but not yet placed in the DOM or the element is removed from the DOM. This state is useful when we want to create animations while adding or removing an element from our DOM. To define this state in our code we use the keyword `void`.
- The wildcard state:** This is also known as the default state of the element. The styles defined for this state is applicable to the element regardless of its current animation state. To define this state in our code we use the `*` symbol.
- Custom state:** This is the custom state of the element and it needs to be defined explicitly in the code. To define this state in our code we can use any custom name of our choice.

Animation Transition Timing:

To show the animation transition from one state to another we define animation transition timing in our application.

Angular provides us the following three timing properties: **animate ('duration delay easing')**

1. Duration

This property represents the time our animation takes to complete from start (initial state) to finish (final state). We can define the duration of the animation in the following three ways:

- Using an integer value, which represents the time in milliseconds, e.g., 500
- Using a string value to represent the time in milliseconds, e.g., '500ms'
- Using a string value to represent the time in seconds. e.g.. '0.5s'

2. Delay

This property represent the time duration between the animation trigger and the beginning of the actual transition. This property also follows the same syntax as duration. To define the delay, we need to add the delay value after the duration value in a string format — ' Duration Delay'. Delay is an optional property, e.g., '0.3s 500ms'. This means the transition will wait for 500ms and then run for 0.3s.

3. Easing

This property represents how the animation accelerates or decelerates during its execution. We can define the easing by adding it as the third variable in the string after duration and delay. If the delay value is not present, then easing will be the second value. This is also an optional property. For example:

- '0.3s 500ms ease-in'. This means the transition will wait for 500ms and then run for 0.3s (300ms) with the ease-in effect.
- '300ms ease-out'. This means the transition will run for 300ms (0.3s) with the ease-out effect.
- '200ms ease-in-out'. This means the transition will run for 200ms (0.2s) with the ease-in-out effect.

Configure Animation in Angular Application:

Find the steps to configure animation in our Angular application.

Step-1: Make sure that **package.json** contains **@angular/animations** in **dependencies** block. If not available then either upgrade Angular CLI or configure **@angular/animations** in **dependencies** block and run **npm install**.

Step-2: Configure **BrowserAnimationsModule** in application module to support animations.

```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

@NgModule({
  imports: [
    ...
    BrowserAnimationsModule
  ]
...
})
export class AppModule { }
```

Understanding the Angular Animation Syntax

We will write our animation code inside the component's metadata. The syntax for animation is as shown below:

```
@Component({
  // other component properties.
  animations: [
    trigger('triggerName'), [
      state('stateName', style())
      transition('stateChangeExpression', [Animation Steps])
    ]
  ]
})
```

Here we will use a property called **animations**. This property will take an array as input. The array contains one or more "trigger." Each trigger has a unique name and an implementation. The state and transitions for our animation need to be defined in the trigger implementation.

Each state function has a **stateName** defined in order to uniquely identify the state and a style function to show the style of an element in that state.

Each transition function has a **stateChangeExpression** defined to show the change of state of an element and the corresponding array of animation steps to show how the transition will take place. We can include multiple trigger functions inside the animation property as comma separated values.

These functions (**trigger**, **state**, and **transition**) are defined in the **@angular/animations** module. Hence, we need to import this module in our component.

To apply animation on an element we need to include the trigger name in the element definition. Include the trigger name followed by the **@** symbol in the element tag. Refer to the sample code below:

```
<div @changeSize></div>
```

This will apply the trigger **changeSize** to the **<div>** element.

Let us create a few animations to get a better understanding of the Angular animation concepts.

Change Size Animation:

We will create an animation to change the size of a `<div>` element on a button click.

Open the `animationdemo.component.ts` file and add the following import definition.

```
import { trigger, state, style, animate, transition } from '@angular/animations';
```

Add the following animation property definition in the component metadata.

```
animations: [
  trigger('changeDivSize', [
    state('initial', style({
      backgroundColor: 'green',
      width: '100px',
      height: '100px'
    })),
    state('final', style({
      backgroundColor: 'red',
      width: '200px',
      height: '200px'
    })),
    transition('initial=>final', animate('1500ms')),
    transition('final=>initial', animate('1000ms'))
  ])
]
```

Here we have defined a trigger, `changeDivSize`, and two state functions inside the trigger. The element will be green in color in the "initial" state and will be red in color with an increased width and height in the "final" state.

We have defined transitions for the state change. The transition from "initial" state to "final" will take 1500ms and from "final" state to "initial" will take 1000ms.

To change the state of our element we will define a function in the class definition of our component. Include the following code in the `AnimationdemoComponent` class:

```
currentState = 'initial';
changeState() {
  this.currentState = this.currentState === 'initial' ? 'final' : 'initial';
}
```

Here we have defined a `changeState` method which will switch the state of the element.

Open `animationdemo.component.html` file and add the following code:

```
<h3>Change the div size</h3>
<button (click)="changeState()">Change Size</button>
<br /><br />
<div [@changeDivSize]=currentState></div>
<br />
```

We have defined a button which will invoke the **changeState** function when clicked. We have defined a **<div>** element and applied the **changeDivSize** animation trigger to it. When we click on the button it will flip the state of the **<div>** element and its **size** will change with a **transition effect**.

Before executing the application, we need to include the reference to our **Animationdemo** component inside the **app.component.html** file.

Open the **app.component.html** file. You can see we have some default HTML code in this file. Delete all the code and put in the selector of our component as shown below:

```
<app-animationdemo></app-animationdemo>
```

Complete Example Code:

animationdemo.component.ts:

```
import { Component } from '@angular/core';
import { trigger, state, style, animate, transition } from '@angular/animations';
@Component({
  selector: 'app-animationdemo',
  templateUrl: './animationdemo.component.html',
  animations: [
    trigger('changeDivSize', [
      state('initial', style({
        backgroundColor: 'green',
        width: '100px',
        height: '100px'
      })),
      state('final', style({
        backgroundColor: 'red',
        width: '200px',
        height: '200px'
      })),
      transition('initial=>final', animate('1500ms')),
      transition('final=>initial', animate('1000ms'))
    ])
  ]
})
export class AnimationdemoComponent {
  currentState = 'initial';
  changeState() {
    this.currentState = this.currentState === 'initial' ? 'final' : 'initial';
  }
}
```

animationdemo.component.html:

```
<h3>Change the div size</h3>
<button (click)="changeState()">Change Size</button>
<br /><br />
<div [@changeDivSize]=currentState></div>
<br />
```

app.component.ts:

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

app.component.html:

```
<div class="p-2">
  <app-animationdemo></app-animationdemo>
</div>
```

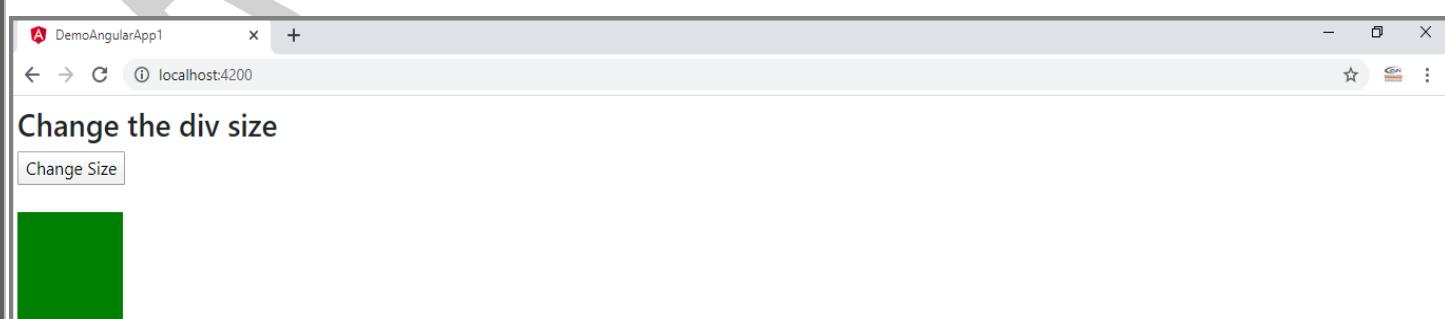
app.module.ts:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { AnimationdemoComponent } from './animationdemo.component';
```

```
@NgModule({
  declarations: [
    AppComponent, AnimationdemoComponent
  ],
  imports: [
    BrowserModule, AppRoutingModule, BrowserAnimationsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

To execute the code run the ng serve --o command in the VS code terminal window. After running this command, it will ask you to open <http://localhost:4200> in the browser. So, open any browser on your machine and navigate to this URL. You can see a webpage as shown below. Click on the button to see the animation.

Output:



Balloon Effect Animation:

In the previous animation, the transition happened in two directions. In this section, we will learn how to change the size from all direction. It will be similar to inflating and deflating a balloon hence the name, balloon effect animation.

Add the following trigger definition in the animation property of **AnimationdemoComponent** (**animationdemo.component.ts**).

```
import { Component } from '@angular/core';
import { trigger, state, style, animate, transition } from '@angular/animations';
@Component({
  selector: 'app-animationdemo',
  templateUrl: './animationdemo.component.html',
  animations: [
    trigger('balloonEffect', [
      state('initial', style({
        backgroundColor: 'green',
        transform: 'scale(1)'
      })),
      state('final', style({
        backgroundColor: 'red',
        transform: 'scale(1.5)'
      })),
      transition('final=>initial', animate('1000ms')),
      transition('initial=>final', animate('1500ms'))
    ])
])
export class AnimationdemoComponent {
  currentState = 'initial';
  changeState() {
    this.currentState = this.currentState === 'initial' ? 'final' : 'initial';
  }
}
```

Here, instead of defining the width and height property, we are using the transform property to change the size from all directions. The transition will occur when the state of the element is changed.

Add the following HTML code in **animationdemo.component.html** file.

```
<h3>Balloon Effect</h3>
<div (click)="changeState()" style="width:100px; height:100px; border-radius: 100%; margin: 3rem; background-color: green" [@balloonEffect]=currentState>
</div>
```

Here we have defined a div and applied the CSS style to make it a circle. Clicking on a div will invoke the **changeState** method to switch the element's state.

Open the browser to see the animation in action as shown below:



Fade In and Fade Out Animation

Sometimes we want to show an animation while adding or removing an element on the DOM. We will see how to animate the adding and removing of an item to a list with fade-in and fade-out effect.

Add the following code inside the **AnimationdemoComponent** class definition for adding and removing the element in a list.

```
listItem = [];
list_order: number = 1;
addItem() {
  var listitem = "ListItem " + this.list_order;
  this.list_order++;
  this.listItem.push(listitem);
}
removeItem() {
  this.listItem.length -= 1;
}
```

Add the following trigger definition in the animation property.

```
animations: [
  trigger('fadeInOut', [
    state('void', style({
      opacity: 0
    })),
    transition('void <=> *', animate(1000)),
  ])
]
```

Here we have defined the trigger **fadeInOut**. When the element is added to the DOM it is a transition from void to wildcard (*) state. This is denoted using **void => ***. When the element is removed from DOM it is a transition from wildcard (*) to void state. This is denoted using *** => void**.

When we use the same animation timing for both directions of the animation we use a shorthand syntax, **<=>**. As defined in this trigger, the animation from **void => *** and *** => void** will take 1000ms to complete.

Add the following HTML code in the **animationdemo.component.html** file.

```
<h3>Fade-In and Fade-Out animation</h3>
<button (click)="addItem()">Add List</button>
<button (click)="removeItem()">Remove List</button>
<div style="width:200px; margin-left: 20px">
  <ul>
    <li *ngFor="let list of listItem" [@fadeInOut]>
      {{list}}
    </li>
  </ul>
</div>
```

Here we are defining two buttons to add and remove an item. We are binding the **fadeInOut** trigger to the **** element, which will show a **fade-in** and **fade-out** effect while being added and removed from the DOM.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Complete Code of **AnimationdemoComponent** Class & html template:

animation.demo.component.ts:

```
import { Component } from '@angular/core';
import { trigger, state, style, animate, transition } from '@angular/animations';
@Component({
  selector: 'app-animationdemo',
  templateUrl: './animationdemo.component.html',
  animations: [
    trigger('fadeInOut', [
      state('void', style({
        opacity: 0
      })),
      transition('void <=> *', animate(1000)),
    ])
  ]
})
export class AnimationdemoComponent {
  listItem = [];
  list_order: number = 1;
  addItem() {
    var listitem = "ListItem " + this.list_order;
    this.list_order++;
    this.listItem.push(listitem);
  }
  removeItem() {
    this.listItem.length -= 1;
  }
}
```

animation.demo.component.html:

```
<h3>Fade-In and Fade-Out animation</h3>
<button (click)="addItem()">Add List</button>
<button (click)="removeItem()">Remove List</button>
<div style="width:200px; margin-left: 20px">
  <ul>
    <li *ngFor="let list of listItem" [@fadeInOut]>
      {{list}}
    </li>
  </ul>
</div>
```

Open the browser to see the animation in action as shown below:

Output:



Enter and Leave Animation:

While adding to the DOM, the element will enter the screen from the left and while deleting the element will leave the screen from the right.

The transition from **void => *** and *** => void** is very common. Therefore, Angular provides aliases for these animations for **void => *** we can use '**:enter**'

for *** => void** we can use '**:leave**'

The aliases make these transitions more readable and easier to understand.

Add the following trigger definition in the animation property.



```
animations: [
  trigger('EnterLeave', [
    state('flyIn', style({ transform: 'translateX(0)' })),
    transition(':enter', [
      style({ transform: 'translateX(-100%)' }),
      animate('0.5s 300ms ease-in')
    ]),
    transition(':leave', [
      animate('0.3s ease-out', style({ transform: 'translateX(100%)' }))
    ])
  ])
]
```

Here we have defined the trigger **EnterLeave**. The '**:enter**' transition will wait for 300ms and then run for 0.5s with an ease-in effect. Whereas the '**:leave**' transition will run for 0.3s with an ease-out effect.

Add the following HTML code in the **animationdemo.component.html** file.

```
<h3>Enter and Leave animation</h3>
<button (click)="addItem()">Add List</button>
<button (click)="removeItem()">Remove List</button>
<div style="width:200px; margin-left: 20px">
  <ul>
    <li *ngFor="let listItem of list" [@EnterLeave]="'flyIn'">
      {{listItem}}
    </li>
  </ul>
</div>
```

Here we are defining two buttons to add and remove an item to the list. We are binding the **EnterLeave** trigger to the **** element that will show enter and leave effect while being added and removed from the DOM.

Complete Code of **AnimationdemoComponent** Class & html template:

animation.demo.component.ts:

```
import { Component } from '@angular/core';
import { trigger, state, style, animate, transition } from '@angular/animations';
@Component({
  selector: 'app-animationdemo',
  templateUrl: './animationdemo.component.html',
})
```

```

        animations: [
            trigger('EnterLeave', [
                state('flyIn', style({ transform: 'translateX(0)' })),
                transition(':enter', [
                    style({ transform: 'translateX(-100%)' }),
                    animate('0.5s 300ms ease-in')
                ]),
                transition(':leave', [
                    animate('0.3s ease-out', style({ transform: 'translateX(100%)' }))
                ])
            ])
        ]
    })
}

export class AnimationdemoComponent {
    listItem = [];
    list_order: number = 1;
    addItem() {
        var listitem = "ListItem " + this.list_order;
        this.list_order++;
        this.listItem.push(listitem);
    }
    removeItem() {
        this.listItem.length -= 1;
    }
}

```

animation.demo.component.html:

```

<h3>Enter and Leave animation</h3>
<button (click)="addItem()">Add List</button>
<button (click)="removeItem()">Remove List</button>
<div style="width:200px; margin-left: 20px">
    <ul>
        <li *ngFor="let list of listItem" [@EnterLeave]="'flyIn'">
            {{list}}
        </li>
    </ul>
</div>

```

Open the browser to see the animation in action as shown below:

Output:



Open & Close Box Animation:

animationdemo.component.ts:

```
import { Component } from '@angular/core';
import { trigger, state, style, animate, transition } from '@angular/animations';
@Component({
  selector: 'app-animationdemo',
  templateUrl: './animationdemo.component.html',
  animations: [
    trigger('OpenClose', [
      state('Open', style({
        backgroundColor: 'red',
        width: '300px',
        height: '300px'
      })),
      state('Close', style({
        width: '0px',
        height: '0px'
      })),
      transition('Open=>Close', animate('1500ms')),
      transition('Close=>Open', animate('1000ms'))
    ])
  ]
})
export class AnimationdemoComponent {
  isOpen = true;

  toggle() {
    this.isOpen = !this.isOpen;
  }
}
```

animationdemo.component.html:

```
<input id="chkOpenClose" type="checkbox" (click)="toggle()" />
<label for="chkOpenClose">{{isOpen ? 'Close' : 'Open'}}</label>
<br />
<div [@OpenClose]="isOpen ? 'Open' : 'Close'">
</div>
```

app.component.ts:

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

app.component.html:

```
<div class="p-2">
  <app-animationdemo></app-animationdemo>
</div>
```

Open the browser to see the animation in action as shown below:

Output:



Animations API Functions:

The functional API provided by the **@angular/animations** module for creating and controlling animations in Angular applications.

Function name	What it does
trigger()	Kicks off the animation and serves as a container for all other animation function calls. HTML template binds to triggerName. Use the first argument to declare a unique trigger name. Uses array syntax.
style()	Defines one or more CSS styles to use in animations. Controls the visual appearance of HTML elements during animations. Uses object syntax.
state()	Creates a named set of CSS styles that should be applied on successful transition to a given state. The state can then be referenced by name within other animation functions.
animate()	Specifies the timing information for a transition. Optional values for delay and easing. Can contain style() calls within.
transition()	Defines the animation sequence between two named states. Uses array syntax.
keyframes()	Allows a sequential change between styles within a specified time interval. Use within animate(). Can include multiple style() calls within each keyframe(). Uses array syntax.
group()	Specifies a group of animation steps (<i>inner animations</i>) to be run in parallel. Animation continues only after all inner animation steps have completed. Used within sequence() or transition().
query()	Use to find one or more inner HTML elements within the current element.
sequence()	Specifies a list of animation steps that are run sequentially, one by one.
stagger()	Staggers the starting time for animations for multiple elements.
animation()	Produces a reusable animation that can be invoked from elsewhere. Used together with useAnimation().
useAnimation()	Activates a reusable animation. Used with animation().
animateChild()	Allows animations on child components to be run within the same timeframe as the parent.

For detail reference visit angular official guide: <https://angular.io/guide/animations>

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

These are most commonly used animation functions in details:

trigger():

trigger is an animation-specific function that creates a trigger with **state** and **transition** entries. **trigger** function accepts following arguments.

1. Trigger name
2. Array of **state** and **transition**.

Trigger is used as follows.

```
trigger('onOffTrigger', [
  state(...),
  state(...),
  transition(...),
  transition(...)
])
```

Trigger function can have any number of **state** and **transition** entries.

style():

style is an animation-specific function. It is defined with key/value of CSS properties. **style** function is passed as an argument in animation-specific functions such as *state*, *transition*, *animate* and *keyframes*. Animation **style** function is defined as follows.

```
style({
  backgroundColor: '#E5E7E9',
  color: '#1C2833',
  fontSize: '18px',
  transform: 'scale(1)'
})
```

state():

state is an animation-specific function. It is an animation state within the given **trigger**. **state** function accepts following arguments.

1. State name
2. style

Animation state function is defined as follows:

```
state('Open', style({
  backgroundColor: 'red',
  width: '300px',
  height: '300px'
}))
```

animate():

animate is an animation-specific function. It specifies an animation step. We need to pass following arguments to **animate** function.

1. Timing expression
2. style or keyframes function. This argument is optional.

Timing expression is the combination of duration, delay and easing. They are written in following formats.

Timing expression: 'duration delay easing'

duration: It defines how long animation will take place. It can be a plane number or string. If we write plane number then by default the time unit will be milliseconds. If we write 100, it means 100 milliseconds and if we want to specify time unit then write as string. '100ms' means 100 milliseconds and '0.1s' means 0.1 second.

delay: This is the time gap between animation trigger and beginning of transition. It is written in string with time unit in string.

easing: It defines how the animations accelerates and decelerates during its runtime. Easing can be ease, ease-in, ease-out, ease-in-out etc.

Find the examples of animate with timing expression.

1.

```
animate(100)
```

duration: 100

Animation will take place for 100ms just after animation trigger.

2.

```
animate('0.6s 100ms ease-in')
```

duration: 0.6s

delay: 100ms

easing: ease-in

After animation trigger there will be a delay of 100 milliseconds to start animation and then animation will take place for 0.6 seconds with ease-in function.

3.

```
animate('0.7s ease-out')
```

duration: 0.7s

easing: ease-out

Animation will start just after trigger. Animation will take place for 0.7 seconds with ease-out function.

Find the example of **animate** with time expression and **style**.

```
animate('0.6s 100ms ease-in', style({
    fontSize: '20px',
    backgroundColor: 'blue'
}))
```

The given **style** will be active from animation start to animation end.

transition():

transition function is an animation-specific function. **transition** function runs the steps of animation for the given state change expression. State change expression is written as **state1 => state2** or **state2 => state1** or **state1 <=> state2**. **transition** function associates state change expression with animation steps.

Find the arguments of transition function.

1. State change expression such as (state1 => state2)
2. Animation step or array of animation steps.

Find some transition function example.

1.

```
transition('state1 => state2', animate(...))
transition('state2 => state3', animate(...))
transition('state3 => state1', animate(...))
```

In the above **transition** function, animation will take place for the defined state change expression.

2. If for two states **animation(...)** definition is same then we can write **transition** function as follows.

```
transition('state1 <=> state2', animate(...))
```

3. If for more than one state change expression, **animation(...)** definition is same then state change expression will be written comma separated.

```
transition('state1 => state2, state2 => state3', animate(...))
```

4.

```
transition('* => *', animate(...))
```

Animation will take place between any state change.

5.

```
transition('void => *', animate(...))
```

Animation will take place from element page entry to any state. The alias of **void => *** is **:enter** and so above transition can also be written as following.

```
transition(':enter', animate(...))
```

6.

```
transition('* => void', animate(...))
```

Animation will take place from any state to no state(void state) of element. The alias of *** => void** is **:leave** and so above transition can also be written as following.

```
transition(':leave', animate(...))
```

keyframes():

keyframes is an animation-specific function that is used with **animate** function. **keyframes** applies different style at different offset of duration of animation. Find the code snippet.

```
animate('0.6s 100ms ease-in', keyframes([
  style({fontSize: '19px', backgroundColor: 'yellow', offset: 0.1}),
  style({fontSize: '20px', backgroundColor: 'green', offset: 0.3}),
  style({fontSize: '21px', backgroundColor: 'red', offset: 0.5})
]))
```

We will observe that we are using more than one **style** within a **keyframes** using offset. Here we have set more than one style that will change according to their offset within the animation start to end. If we do not use offset then it will automatically be calculated.

Copyright © 2019 <https://www.facebook.com/groups/RakeshSoftNetAngular/> All Rights Reserved.

Example with keyframes:

animationdemo.component.ts:

```
import { Component, OnInit } from '@angular/core';
import { trigger, style, animate, transition, keyframes } from '@angular/animations';

@Component({
  selector: 'app-animationdemo',
  templateUrl: './animationdemo.component.html',
  animations: [
    trigger('keyframes', [
      transition(':enter', [
        animate('7s', keyframes([
          style({ backgroundColor: 'lime', transform: 'translateX(0%)' }),
          style({ backgroundColor: 'yellow', transform: 'translateX(800%)' }),
          style({ backgroundColor: 'green', transform: 'translate(800%, 500%)' }),
          style({ backgroundColor: 'red', transform: 'translateY(500%)' }),
          style({ backgroundColor: 'blue', transform: 'translateX(0%)' })
        ])),
      ])
    ])
  ]
})
export class AnimationdemoComponent implements OnInit {
  show: boolean = true;

  ngOnInit() {
    setInterval(() => {
      this.show = !this.show;
      setTimeout(() => {
        this.show = !this.show;
      }, 1);
    }, 7000);
  }
}
```

animationdemo.component.html:

```
<button *ngIf="show" [@keyframes]>Angular Animation</button>
```

app.component.ts:

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

app.component.html:

```
<div class="p-2">
  <app-animationdemo></app-animationdemo>
</div>
```

sequence():

sequence is an animation-specific function. It is used within a transition or group function. sequence function will contain the array of style and animate functions. They will execute in sequence. Find the code snippet.

```
transition('* => void', sequence([
    style({backgroundColor: '#0D6063'}),
    animate('0.6s ease-out', style({transform: 'rotate(-270deg)', opacity: 0}))
]))
```

Using keyword **sequence** is not necessary. It is default, we can directly use array of **style** and **animate** functions.

```
transition('* => void', [
    style({backgroundColor: '#0D6063'}),
    animate('0.6s ease-out', style({transform: 'rotate(-270deg)', opacity: 0}))
])
```

group():

group is an animation-specific function that can be used within a transition or sequence function. group contains the entries of animations steps that run in parallel.

```
transition('void => *', [
    style({
        backgroundColor: '#E3E8EC',
        transform: 'translateX(300%)',
        opacity: 0
    }),
    group([
        animate('0.5s 0.1s ease-in', style({
            transform: 'translateX(0)'
        })),
        animate('0.3s 0.1s ease', style({
            opacity: 1
        }))
    ])
])
```

In the above code snippet **transition** is using state change expression and sequence. Within the sequence we are using **style** and **group** and **group** is using the entries of **animate** functions. These animate functions will run in parallel for the given durations.

(void => *) alias (:enter) and (* => void) alias (:leave)

Here we will provide examples of **void => *** and *** => void** state change expression. **void => *** has the alias **:enter** and *** => void** has alias **:leave**. **void** state means a state for the element when element is not available in DOM. ***** is the default state. This is a state which has not been declared within the **trigger** function. ***** state is used for dynamic start and ending state in animation.