**Introduction to Angular**

**What is Angular?**

Angular is the most popular web development framework for developing mobile apps as well as 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 an open source framework written and maintained by the angular team at *Google*and the Father of Angular is '*Misko Hevery'*. The bootstrapping process creates the components listed in the bootstrap array and inserts each one into the browser (DOM).

It was written in TypeScript, Angular implements core and additional functionality as a set of TypeScript libraries. These are imported to Angular apps. which is completely based on components.

**Why Angular?**

There are many front-end JavaScript frameworks to choose from today, each one is having their own benefits. Angular made it faster, more scalable and modern and was written in Typescript, a superset of JavaScript.

Let’s have a look at the main benefits that makes Angular stand out.

**TypeScript**: As mentioned earlier Angular was written in Typescript, which is basically a super set of JavaScript. It fully compiles to JavaScript.

**Data Binding**: If there is one feature that Angular made us go “Wow”, then it was probably its two-way data binding system.

**Routing:** It Provides a service that lets you define a Navigation path among the different application states and view hierarchies in your app. The router maps URL-like paths to views instead of pages.

**Component-based architecture:** Components can be thought of as small pieces of user interface, like a section of the application. They allow you to control the UI of your app with Reusability, Readability, Maintainability are the benefits.

**Integration:** It helps you to integrate the third-party features into the application you are building.

**High Performance:** Multiple factors can help in making your application faster. The main boost is ensured by hierarchical dependency injection and Angular Universal support.

**History:**

**Angular1.0/Angular JS:**

The first version of Angular was Angular1.0 (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.

**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.

**Angular4:**

Angular 4 version was announced on 13 December 2016. The developers skipped the version 3 due to some confusion. Its final version was released on 23 March 2017.

This version has some additional features:

* This version introduced Http Client, a smaller, easier to use, and more powerful library for making HTTP Requests.
* It provides new router life cycle events for Guards and Resolvers. Four new events: GuardsCheckStart, GuardsCheckEnd, ResolveStart, ResolveEnd join the existing set of life cycle event such as NavigationStart.
* It provides the support of conditionally disable animations.

**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.

**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 Shakable Providers, Animations Performance Improvements, and RxJS v6.

**Angular 7:**

The latest version of Angular is Angular 7. It was released on October 18, 2018. This Version is primarily focused on the Ivy project, which has been going on since past release.

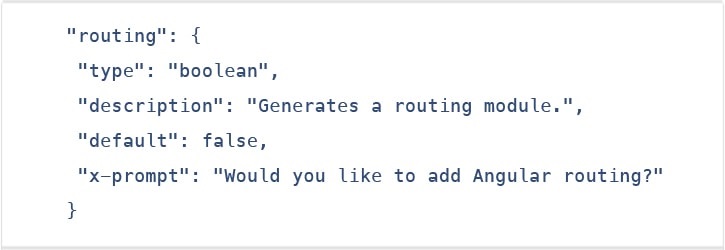
The Ivy project is basically rewriting the Angular compiler and runtime code to make it better, faster, and smaller.

It consists of many extensive features:

**1. CLI prompts**

In Angular 7, the CLI prompts have been updated to v7.0.2 with new features. For instance, it will now prompt users when typing commands like @angular/material, ng-new, and ng-add to help them discover the in-built SCSS support, routing, and more.

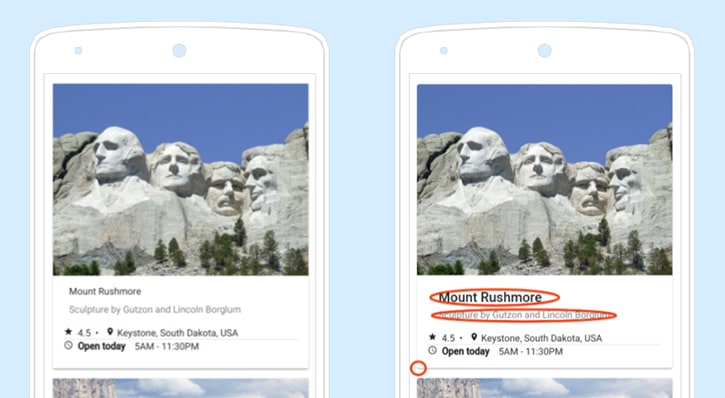
These CLI prompts, in addition, have been added to Schematics, so that all package publishing schematics can now benefit from CLI prompts.



**2. Angular material & component dev kit (CDK)**

The Angular 7 introduced minor visual updates & improvements in Material Design that earlier received a major update this year only.

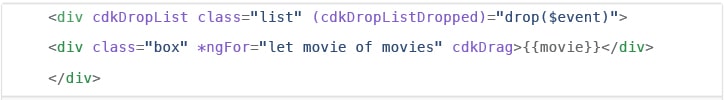
In addition, refresh, virtual scrolling, large lists of data, dynamic loading and unloading of parts of the DOM also were the part of improvements in CDK and Angular Material.



**3. Drag & drop**

The new drag-drop module basically provides a better way to easily create drag & drop interfaces, which is backed by sorting within a list, support for free dragging, animations, custom drag handles, transferring items between lists, previews, and placeholders.

In simple terms, the Drag-and-Drop support has now been implemented in CDK and it also includes automatic rendering as the users relocates items.





**4. Virtual scrolling**

Like mentioned earlier, the new Virtual Scrolling in Angular 7 basically loads and unloads items from the DOM depending upon visible parts of lists, resulting into a much faster experiences for users having huge scrollable lists.

This virtual scrolling package basically provides helpers, which react to all scroll events.



Simply put, it activates a high-performant way by making the height of container element exactly same as the height of total number of remaining elements to be rendered.

This, in turn, then renders the only visible items in view, creating faster experiences for the end-users.

**5. Application performance improvements**

The development team at Google have always focused on the performance improvements, and while doing so, they recently found that most of the developers were using reflect-metadata in their production, which actually was only required in the development.



**6. No Ivy**

No Ivy rendered in the Angular 7 according to the official information. The Angular JS Development Company’s team have said that the Ivy is in the pipeline; however, they haven’t disclosed its final timeline.

The official blog post also mentioned that the backward compatibility validation has begun. And its full beta version is expected to launch with Angular 8 version.

**7. Documentation updates**

Another key improvement introduced in the Angular 7 is of the documentation update. The team has worked hard on improving the reference material and the guidelines for the betterment and convenience of the developers.

The documentation updates for Angular is an important step for the Angular CLI.

**8. Dependency updates**

Documentation are not the only things that have been updated. Even the dependencies have undergone upgradation for the third-party projects.

The support for Node 10, TypeScript3.1, and the RxJS6.3 all are included under this update.

However, you would continue to receive the support if you already have Node 8. Talking about the latest update of TypeScript 3.1, it has now become compulsory for Angular 7 users to bump to TS 3.1.

**Comparison:**

**Comparison between Angular & Angular JS:**

|  |  |
| --- | --- |
| **Angular JS** | **Angular** |
| Released by Google in the year 2010. | Released in Sept 2016. |
| JavaScript-based framework for creating SPA. | Complete re-write of AngularJS version. |
| Still supported but no longer will be developed. | It's updated version regularly released because of Semantic Versioning. |
| The architecture of AngularJS is based on MVC. | The architecture of Angular 2 is based on service/controller. |
| AngularJS was not developed with a mobile base in mind. | Angular 2 is a mobile-oriented framework. |
| AngularJS code can write by using only ES5, ES6, and Dart. | We can use ES5, ES6, Typescript to write an Angular 2 code. |
| Based on controllers whose scope is now over. | Nowadays, the controllers are replaced by components, and Angular two is completely component based. |
| Factory, service, provider, value and constant are used for services | The class is the only method to define services in Angular2 |
| Run on only client-side | Runs on client-side & server-side |
| ng-app and angular bootstrap function are used to initialize | bootstrapmodule() function is used to initialize |

**Comparison between Angular & React:**

React and Angular are two very popular front-end development tools. There are three major elements of concern:

* **HTML:** Used to build the structure of your web page.
* **CSS:** Used to format the appearance of different structural elements.
* **JavaScript:** A programming language used to describe the functionality and handle all the dynamic elements on the web page.

So how do Angular, a framework, and React, a library, compare with each other? Let’s break down the term’s framework and library first.

**Framework:** A software framework (be it front-end or backend) includes standardized, pre-written code, which makes the development of certain functionalities easier and faster. You have less freedom to code, as you have to code as the framework architecture dictates.

**Library:** A library is a collection of functions and functionalities, which you can use to achieve a certain end. You have more freedom to design and construct the system when using a library, but that adds more responsibility on the coder to be able to use it efficiently and find the right library for the right job, because, for projects which need to grow over time and become more serious, this could become significantly riskier and more difficult to manage.

|  |  |  |  |
| --- | --- | --- | --- |
| **Technology** | **React JS** | **Angular JS** | **Angular** |
| **Latest Version** | 16.8 (February 2019) | 1.7.7 (February 2019) | Angular 8 \*(to be released in March/April 2019) |
| **Author** | Facebook Community | Google | Google |
| **Type** | Open Source JS library | Fully-featured MVC framework | Fully-featured MVC framework |
| **Tool Chain** | High | Low | High |
| **Language** | JSX | JavaScript, HTML | TypeScript |
| **Learning Curve** | Low | High | Medium |
| **Packaging** | Strong | Weak | Medium |
| **Rendering** | Server Side | Client Side | Server Side |
| **App Architecture** | None, combined with Flux | MVC | Component-Based |
| **Data Binding** | Uni-Directional | Bi-Directional | Bi-Directional |
| **DOM** | Virtual DOM | Regular DOM | Regular DOM |
| **Native Application Development Support** | Available (React Native) | Available | Available |

Now that you know the major differences in Angular and React, let’s find out what to use and when:

**React** is pretty much ideal for logic less applications or computation-less apps. In cases where you need a custom solution, React will best suit your needs because, with React, you can build almost anything.

Especially if you are building:

* Dynamic Applications
* Single Page Apps
* Native Mobile Apps

**Angular** is best for creating:

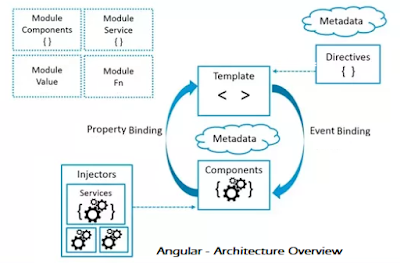
* Cross-platform Mobile Apps
* Enterprise Software
* Progressive web apps and hybrid mobile apps

**EXAMPLE:**

<https://coursetro.com/posts/code/171/Angular-7-Tutorial---Learn-Angular-7-by-Example>

**Architecture**

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.

[](https://1.bp.blogspot.com/-dEK3X5tnvgA/WuLlDS2S_0I/AAAAAAAASZU/_xCvA4_tvEIHT3n3EujoVYphlHKx5QwXgCLcBGAs/s400/Angular-5-4-6-Architecture-Overview.png)

With the help of the above architecture overview, you can identify the seven main building blocks of an Angular Application.

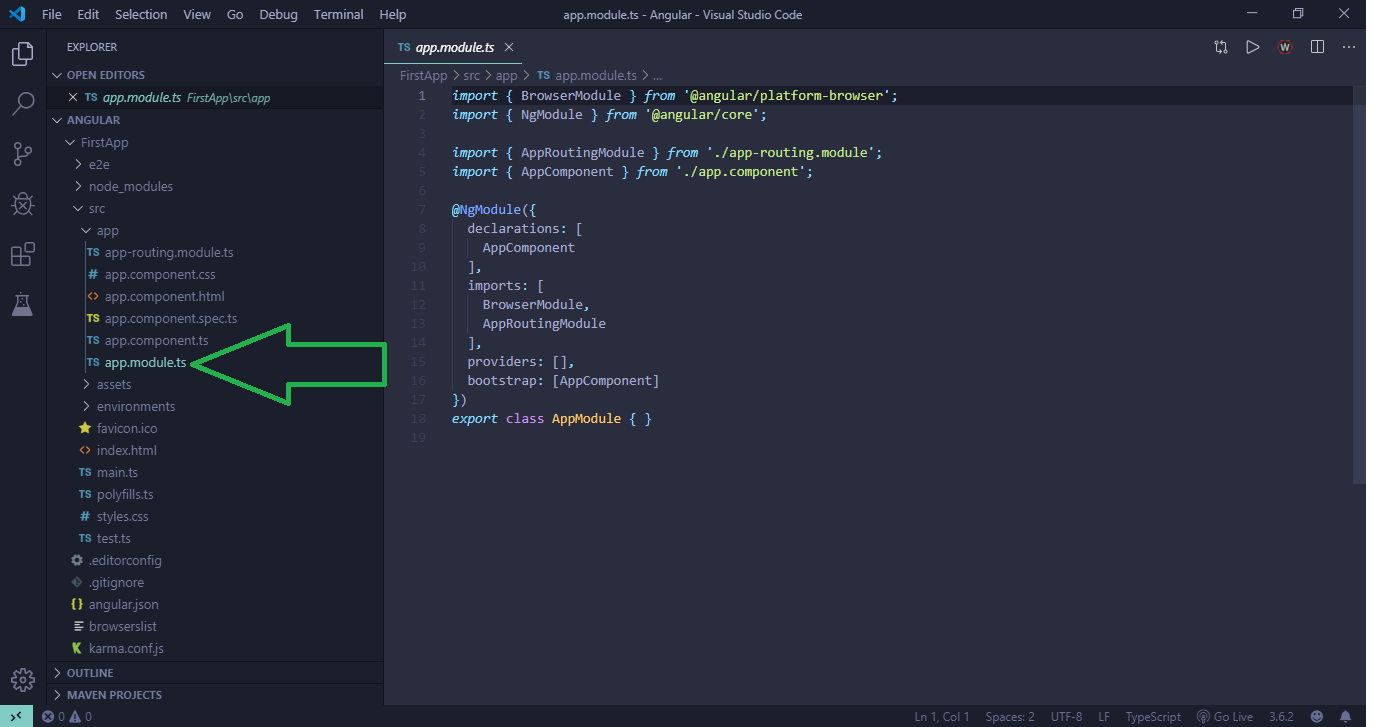
1. Module
2. Component
3. Templates
4. Meta Data
5. Data Binding
6. Directives
7. Services
8. Dependency Injection

**Module:**

The basic building blocks of an Angular application is *NgModules*, which provide a compilation context for components. The angular app is defined by a set of *NgModules*and it always has at least a root module that enables bootstrapping, and many more feature modules.

Some important features of Angular 7 Modules:

* Angular 7 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.

Organizing your code into distinct functional modules helps in managing development of complex applications, and in designing for reusability. In addition, this technique lets you take advantage of lazy-loading i.e. loading modules on demand to minimize the amount of code that needs to be loaded at startup.

The purpose of a NgModule is to declare each thing you create in Angular, and group them together (like Java packages or PHP / C# namespaces).Simply it is a

EX:

@NgModule({  
 declarations:[ AppComponent, NewComponentComponent ],  
 imports: [],  
 providers: [],

bootstrap: [AppComponent]  
})

**Declarations:** Angular CLI registers components in the declarations array by default.

**Imports:** Register all the modules in the imports array.

**Providers:** To make your service globally available register in the providers array.

**Modules to import each time you need them:** If the module is imported for components, you’ll need to import it in each module needing them

* Common Module.
* Forms Module / Reactive Forms Module.
* MatX Module and other UI modules.
* any other module giving components, directives or pipes.

**Modules to import only once:** if the module is imported for services, you’ll need to import it only once, in the first app module

* Http Client Module.
* Browser Animations Module or Noop Animations Module.
* any other module providing services.

**Components:**

Components are the most building block of angular app. It forms a tree structure using these components. The main tasks of component are,

* Binding data between the view and the model
* Displaying a specific page/section and its data with the support of Interpolation, Directives, and Pipes

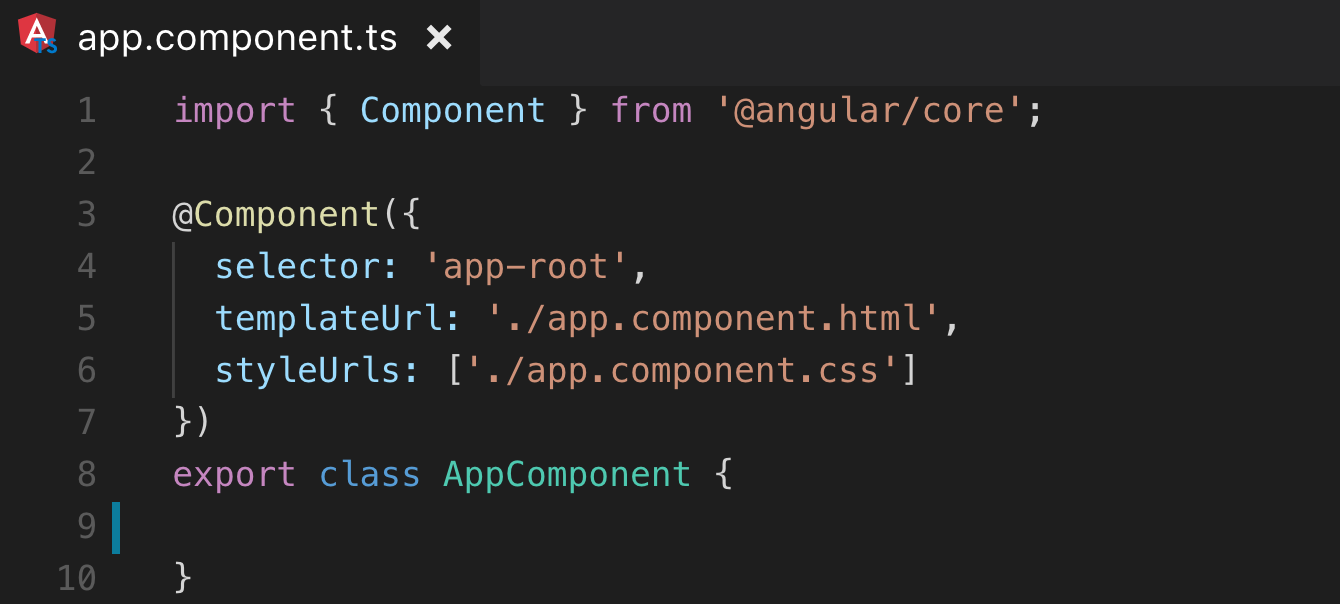
Each component is divided into 3 parts:

Template File - HTML View

Type Script File - Model

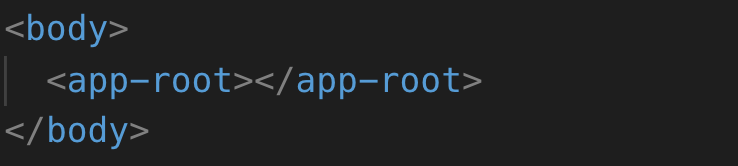
Style File - CSS file

Ex:



**Working:**

* First of all, we import Component from the @angular/core library, so we can create an Angular Component
* The @Component decorator marks the TS class as a Component and allows us to add the following metadata
* The selector is for calling the component inside other HTML files of the project, as an HTML tag: <app-root> </app-root>
* TemplateUrl is where the HTML View of the component is.
* style URLs (can be more than 1) is where the style CSS of the component is.
* Finally, we export the class(component) so that we can call it inside the app.module or other places in the project.

After creating the component, we must save it inside the App Module, under declarations. Finally, we can call the component with its selectorname**,** and then we should be able to see it on the webpage.

Every component has a lifecycle. The ngOnInit() method is called immediately after Angular finishes setting up the component. In other words, Angular is telling that “Component is ready”.

What we can do inside the ngOnInit() method is that some operations like fetching data, or something we want to see immediately on page load. It is not recommended to do these kinds of operations inside the constructor (Only responsible for DI), so we have ngOnInit instead.

**Template, Directives and Data Binding:**

In Angular 7, 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:

* **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.
* **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.

**Directives:**

They specify how to place your components and business logic in the Angular.

Directives are JS class and declared as @directive. There are 3 directives in Angular.

* Component Directives
* Structural Directives
* Attribute Directives

**Component Directives:** Component directives are used in main class. They contain the detail of how the component should be processed, instantiated and used at runtime.

**Structural Directives:** Structural directives start with a \* sign. These directives are used to manipulate and change the structure of the DOM elements. For example, \*ngIf and \*ngFor.

**Attribute Directives:** Attribute directives are used to change the look and behavior of the DOM elements. For example: ngClass, ngStyle etc.

**Custom Directives:**

Custom directives are created by us and are not standard.

To create directive: **ng g directive DirectiveName (or) ng generate directive DirectiveName**

Ex: ng g directive ChangeText

**App.module.ts:**

After directive created, change-text.directive.spec.ts and change-text.directive.ts get created and the app.module.ts file is updated.

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

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

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

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

import { ChangeTextDirective } from './change-text.directive';

@NgModule({

  declarations: [

    AppComponent,

    ChangeTextDirective

  ],

 imports: [ BrowserModule, AppRoutingModule ],

providers: [],

 bootstrap: [AppComponent]

})

export class AppModule {

}

**Change-text.directive.ts:**

Directive and it also has a selector property. Whatever we define in the selector, the same has to match in the view, where we assign the custom directive.

import { Directive, ElementRef} from '@angular/core';

@Directive({

   selector: '[changeText]'

})

export class ChangeTextDirective {

   constructor(Element: ElementRef) {

      console.log(Element);

      Element.nativeElement.innerText = "Text is changed by changeText Directive.";

   }

}

In the above file, there is a class called Change Text Directive and a constructor, which takes the element of type ElementRef, which is mandatory. The element has all the details to which the Change Text directive is applied.

We have added the console.log element. The output of the same can be seen in the browser console. The text of the element is also changed as shown above.

**App.component.html:** In the app.component.html view, let us add the directive as follows

<div style = "text-align:center">

  <h1> Welcome to {{title}}. </h1>

</div>

<div style = "text-align:center">

  <span changeText >Welcome to {{title}}.</span> </div>

**Services and Dependency Injection:**

Services in allow one to define code that's accessible and reusable throughout multiple components. A common use case for services is when you need to communicate with a backend of some sort to send and receive data.

To Create Service: ng generate service ServiceName

Ex: ng generate service data

Then data.service.ts and data.service.spec.ts created.

**In data.service.ts:**

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

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

**@Injectable({**

**providedIn: 'root'**

**})**

**export class DataService {**

**constructor(private http: HttpClient) { }**

**getUsers() {**

**return this.http.get('https://reqres.in/api/users')**

**}**

**}**

**In app.component.ts:**

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

import { DataService } from './data.service';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.scss']

})

export class AppComponent implements OnInit {

  users: Object;

  constructor(private data: DataService) { }

  ngOnInit() {

    this.data.getUsers().subscribe(data => {

        this.users = data

        console.log(this.users);

      }

    );

  }

}

here are 3 things happening here:

* We're first importing the DataService at the top.
* We're creating an instance of it through dependency injection within the constructor () function.
* Then we call the method with this.data.getusers() when the user has to utilize the data (here we call it in ngOnInit() ).

DI is wired into the Angular framework and used everywhere to provide new components with the services or other things they need. Components consume services; that is, you can *inject* a service into a component, giving the component access to that service class.

To define a class as a service in Angular, use the @[Injectable](https://angular.io/api/core/Injectable)() decorator to provide the metadata that allows Angular to inject it into a component as a *dependency*. Similarly, use the @[Injectable](https://angular.io/api/core/Injectable)() decorator to indicate that a component or other class (such as another service, a pipe, or an NgModule) *has* a dependency.

* The *injector* is the main mechanism. Angular creates an application-wide injector for you during the bootstrap process, and additional injectors as needed. You don't have to create injectors.
* An injector creates dependencies, and maintains a *container* of dependency instances that it reuses if possible.
* A *provider* is an object that tells an injector how to obtain or create a dependency.

For any dependency that you need in your app, you must register a provider with the app's injector, so that the injector can use the provider to create new instances. For a service, the provider is typically the service class itself.

A dependency doesn't have to be a service—it could be a function, for example, or a value.

When Angular creates a new instance of a component class, it determines which services or other dependencies that component needs by looking at the constructor parameter types. For example, the constructor of  App.component needs DataService.

**src/app/app.component.ts (constructor):**

constructor(private data: DataService)

**Displaying Dynamic Data**

**Types of Directives:** There are three types of directives in Angular,

* Components directives
* Structural directives
* Attribute directives

**Components directives:**

It is mainly used to specify the HTML templates. It is the most commonly-used directive in an Angular project. It is decorated with the @component decorator. This directive is a class. The component directive is used to specify the template/HTML for the Dom Layout. Its built-in is @component.

A root component is the first Angular component that gets bootstrapped when the application runs. Two things are special about this component:

First, if you open the application module file **src/app/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';

@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

In the above, it's added to the bootstrap array of the module definition.

Second, if you open the src/index.html file (the first file that gets rendered when you visit the application URL) of the application:

<!doctype html>

<html lang="en">

<head>

  <meta charset="utf-8">

  <title>Angular6Project</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>

  <Satya-App>Loading AppComponent, Please wait....</Satya-App>

</body>

</html>

**Code Ref. of app.component.ts:** It provides metadata that determines how the component should be processed, instantiated and used at runtime. The modified code is,

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

@Component({

  selector: 'Satya-App',

  template: '<h1>{{name}}</h1>'

})

export class AppComponent {

  name: string = "Angular 7 By Poojitha"

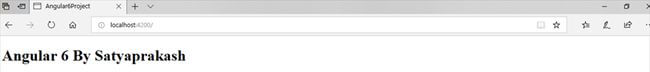
}

**Output:**

Initially the page will look like this:

C:\Users\Pujitha.t\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\AEAF1CD5.tmp

Then the result will be shown like this.



**Attribute Directive:**

Attribute directive is used to change/modify the behavior of the HTML element in the Dom Layout. Its built-in type is NgStyle, NgClass. it is used to change the attributes of the existing HTML element. NgClass, NgStyle are the most used attribute directives.

**NgStyle:**

NgStyle directive is similar to one of the data binding techniques called style binding in Angular. The main point is NgStyle is used to set multiple inline styles for the HTML element.

**NgClass:**

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

**Code Ref. of app.component.ts for NgStyle:**

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

import { Attribute } from '@angular/compiler';

@Component({

    selector: 'Satya-App',

    template: `<button style='color:green' [ngStyle]="MyStyle()">Poojitha Ramana Thotakura</button>`

})

export class AppComponent {

    isBold: boolean = true;

    fontSize: number = 30;

    isItalic: boolean = true;

    MyStyle() {

        let mystyles = {

            'font-weight': this.isBold ? 'bold' : 'normal',

            'font-style': this.isItalic ? 'italic' : 'normal',

            'font-size.px': this.fontSize

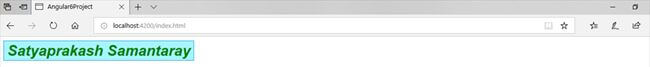
        };

        return mystyles;

    }

}

**Output:**



**Code Ref. of app.component.ts for NgClass:**

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

@Component({

    selector: 'Satya-App',

    template: `<button class='colorClass' [ngClass]='applyClasses()'>Satyaprakash Samantaray</button>`,

    styles: [`.boldClass{

                font-weight:bold;

                  font-size : 30px;

            }

            .italicsClass{

                font-style:italic;

            }

            .colorClass{

                color:Red;

 }`]

})

export class AppComponent {

    applyBoldClass: boolean = true;

    applyItalicsClass: boolean = true;

    applyClasses() {

        let classes = {

            boldClass: this.applyBoldClass,

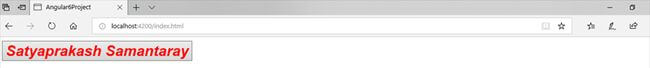
            italicsClass: this.applyItalicsClass

        };

        return classes;

    }

}

**Output:**

**Structural Directive:**

The structural directive is used to add or remove the HTML Element in the Dom Layout, typically by adding, removing, or manipulating elements... Its built-in types are \*NgIf,\*NgFor,\*NgSwitch. Structural directives are easy to recognize by using an asterisk (\*). The asterisk is "syntactic sugar" for something a bit more complicated. Internally, Angular translates the \*ngIf attribute into a <ng-template> element, wrapped around the host element. The \*ngIf directive moved to the <ng-template> element and it became a property binding,[ngIf].

**Difference between the structural directive and attribute directive:**

The structural directive is used to add or remove the Dom Element itself in the Dom Layout, whereas attribute directives are used to just change the attribute or appearance of the Dom element.

**Types of built-in structural directive:**

* NgIf
* NgFor
* NgSwitch

**NgIf** : It is used to create or remove a part of the DOM tree depending on a condition.

**NgFor**: It is used to customize data display. It is mainly used for displaying a list of items using repetitive loops.

**NgSwitch**: It 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.

**Code Ref. of app.component.ts for implementation of NgIf:**

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

@Component({

    selector: 'Satya-App',

    template: `<div style='color:blue' \*ngIf="true">You can See Text....Using ngIf directive</div>`

})

export class AppComponent {

}

**Output:**

C:\Users\Pujitha.t\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\4D0460CD.tmp

**Code Ref. of app.component.ts for implementation of NgFor:**

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

@Component({

    selector: 'Satya-App',

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

})

export class AppComponent {

    employees: any[] = [

        {

            code: '1001', name: 'Satya', gender: 'Male',

            annualSalary: 5500, dateOfBirth: '25/6/1988'

        },

        {

            code: '1002', name: 'Mohit', gender: 'Male',

            annualSalary: 5700.95, dateOfBirth: '9/6/1982'

        },

        {

            code: '1003', name: 'Rohit', gender: 'Male',

            annualSalary: 5900, dateOfBirth: '12/8/1979'

        },

        {

            code: '1004', name: 'Satyaprakash Samantaray', gender: 'Female',

            annualSalary: 6500.826, dateOfBirth: '14/10/1980'

        },

    ];

}

**Code ref. of app.component.html for implementation of NgFor:**

 <!DOCTYPE html>

<html>

<head>

    <title></title>

    <meta charset="utf-8" />

    <style>

      table {

          font-family: arial, sans-serif;

          border-collapse: collapse;

          width: 100%;

      }

      td, th {

          border: 1px solid #dddddd;

          text-align: left;

          padding: 8px;

      }

      tr:nth-child(even) {

          background-color: #dddddd;

      }

  </style>

</head>

<body>

    <table align="center" border="1" cellpadding="4" cellspacing="4">

        <thead>

            <tr>

                <th style="background-color: Yellow;color: blue">Code</th>

                <th style="background-color: Yellow;color: blue">Name</th>

                <th style="background-color: Yellow;color: blue">Gender</th>

                <th style="background-color: Yellow;color: blue">Annual Salary</th>

                <th style="background-color: Yellow;color: blue">Date of Birth</th>

            </tr>

        </thead>

        <tbody>

            <tr \*ngFor='let employee of employees'>

                <td>{{employee.code}}</td>

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

                <td>{{employee.gender}}</td>

                <td>{{employee.annualSalary}}</td>

                <td>{{employee.dateOfBirth}}</td>

            </tr>

            <tr \*ngIf="!employees || employees.length==0">

                <td colspan="5">

                    No employees to show in the page....

                </td>

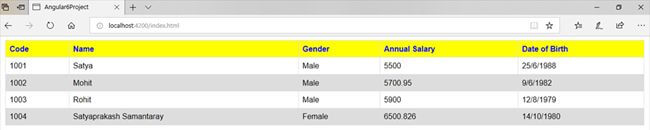
            </tr>

        </tbody>

    </table>

</body>

</html>

**Output:**

**Code Ref. of app.component.ts for implementation of NgSwitch:**

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

@Component({

    selector: 'Satya-App',

    template: `<h2>{{title}}</h2>

<p \*ngIf="showElement">Show Element</p>

<div [ngSwitch]="inpvalue">

<p style='color:blue'  \*ngSwitchCase="1">You have selected M S Dhoni</p>

<p style='color:blue'  \*ngSwitchCase="2">You have selected Sachin Tendulkar</p>

<p style='color:blue'  \*ngSwitchCase="3">You have selected any option using swith case..</p>

<p style='color:red'  \*ngSwitchDefault>Sorry! Invalid selection....</p>

</div>`

})

export class AppComponent {

    inpvalue: number = 3;

}

**Output:**

For valid selection:

For Invalid selection:

C:\Users\Pujitha.t\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\1079635F.tmp

**\* vs <Template>:**

The HTML <template> element is a mechanism for holding client-side content that is not to be rendered when a page is loaded but may subsequently be instantiated during runtime using Java script.

* ng-template is a virtual element and its contents are displayed only when needed (based on conditions).
* It 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.
* ng-template never meant to be used like other HTML elements. It’s an internal implementation of Angular’s structural directives.
* 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.
* 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.
* 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.
* Angular will evaluate the ng-template element to convert it into a comment section in HTML DOM.

Example:

<div \*ngIf="display" class="ng-template-example">

  ng-template example

</div>

<ng-template [ngIf]="display">

  <div class="ng-template-example">ng-template example</div>

</ng-template>

One more example in brief:

<ng-template>

  You will never see me :(

</ng-template>

<ng-template \*ngIf="showContents">

  You still won’t see me (even though I’m using a structural directive, there’s nothing that's telling Angular to use me)

</ng-template>

<ng-template \*ngFor="let item of items">

  No dice (same reason as \*ngIf above)

</ng-template>

<div \*ngIf="!showTemplate else template">

  Hello

</div>

<ng-template #template>

  Finally I am seen! (When showTemplate is true)

</ng-template>

**When to use <ng-container>:**

<ng-container> is an Angular grouping element it doesn’t represent a DOM element. The difference is that it will always be rendered, whereas an <ng-template> will only be rendered if it is explicitly requested. <ng-container>s are useful anywhere you need an extra container for some template elements, but don’t want to (or can’t) create a container such as a div to hold them with due to syntax or style constraints.

<!-- Can't do this -->

<div \*ngIf="todos" \*ngFor="let todo of todos">

  {{ todo.content }}

</div>

You can’t use more than one template binding on an element. So you can’t combine structural directives like this. Instead what we usually do is use a wrapper like this.

<div \*ngIf="todos">

  <div \*ngFor="let todo of todos">

    {{ todo.content }}

  </div>

</div>

This is all well and good, but it adds a useless div element to the DOM. This is where the ng-container element becomes useful. The following behaves exactly the same, but without adding any extra element to the DOM at runtime.

<ng-container \*ngIf="todos">

  <div \*ngFor="let todo of todos">

    {{ todo.content }}

  </div>

</ng-container>

## Angular Components Deep Dive

## What are Components?

Components are the key features of Angular. The whole application is built by using different components. An Angular application is a tree of Angular components. Angular components are a subset of directives. Unlike directives, components always have a template and only one component can be instantiated per an element in a template. 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 part of the presentational layers of our applications i. e, User interaction and presentation. Components sit on the outermost layer of our overall application architecture and they are the method of presenting information to our users and allow them to interact with the application.

**Create a component using CLI:**

ng generate component component name

Or

ng g c component name

The following files are created by default when we created new project using the angular-cli command.

* 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

Components are defined using the @component decorator. A component has a selector, template, style and other properties, using which it specifies the metadata required to process the component.

## Components Life Cycle Hooks:

Every Angular component and Angular directive have a lifecycle which is managed internally by Angular. Angular creates it, renders it, creates and renders its children, checks it when it’s data-bound properties change, and destroys it before removing it from the DOM.

That is a very simple description of the sequence of events that an Angular component’s life experiences. These events are called “Lifecycle Hooks”.  Developers can use these lifecycle hooks to do something (run some code) whenever one of these events occur. There are eight different lifecycle hooks that a developer can tap into in any component or directive. To do so, a developer just needs to add one of the eight function calls that correspond to the lifecycle event.

**8 Lifecycle Hooks:**

* **ngOnChanges()**
  + Used in pretty much any component that has an input.
  + Called whenever an input value change.
  + Is called the first time before ngOnInit.
* **ngOnInit()**
  + Used to initialize data in a component.
  + Called after input values are set when a component is initialized.
  + Added to every component by default by the Angular CLI.
  + Called only once.
* **ngDoCheck()**
  + Called during all change detection runs.
  + A run through the view by Angular to update/detect changes.
* **ngAfterContentInit()**
  + Called only once after first ngDoCheck().
  + Called after the first run through of initializing content.
* **ngAfterContentChecked()**
  + Called after every ngDoCheck().
  + Waits till after ngAfterContentInit() on first run through.
* **ngAfterViewInit()**
  + Called after Angular initializes component and child component content.
  + Called only once after view is initialized.
* **ngAfterViewChecked()**
  + Called after all the content is initialized and checked. (Component and child components).
  + First call is after ngAfterViewInit().
  + Called after every ngAfterContentChecked() call is completed.
* **ngOnDestroy()**
  + Used to clean up any necessary code when a component is removed from the DOM.
  + Fairly often used to unsubscribe from things like services.
  + Called only once just before component is removed from the DOM.

**Dynamic Components:**

Dynamic means, that the components location in the application is not defined at build time. That means, that it is not used in any angular template. Instead, the component is instantiated and placed in the application at runtime.

Imagine we have a modal window component. If we want to encapsulate the idea of a popup message that can be reused in many different situations. For this we will create a 'dynamically generated component' that also supports content projection.

For Example,

Create a angular app and generate a dynamic component with name ‘dynamic’ by using the following CLI command,

ng generate component dynamic (or) ng g c dynamic

Inside dynamic.component.html

<p>Dynamic Component {{ index }}</p>

This text should be dynamically added. Then render this dynamic component inside app.component.html as follows,

<button (click)="add()">Dynamically Add Component</button>

<!-- This is where we will insert the dynamic components -->

<div #container></div>

The functionality behind the dynamic component should be write inside app.component.ts i.e,

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

import { DynamicComponent } from './dynamic/dynamic.component';

@Component({

  selector: 'my-app',

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

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

})

export class AppComponent  {

  @ViewChild('container', { read: ViewContainerRef }) container: ViewContainerRef;

  private \_counter = 1;

  constructor(private componentFactoryResolver: ComponentFactoryResolver) { }

  add(): void {

    // create the component factory

    const componentFactory = this.componentFactoryResolver.resolveComponentFactory(DynamicComponent);

    // add the component to the view

    const componentRef = this.container.createComponent(componentFactory);

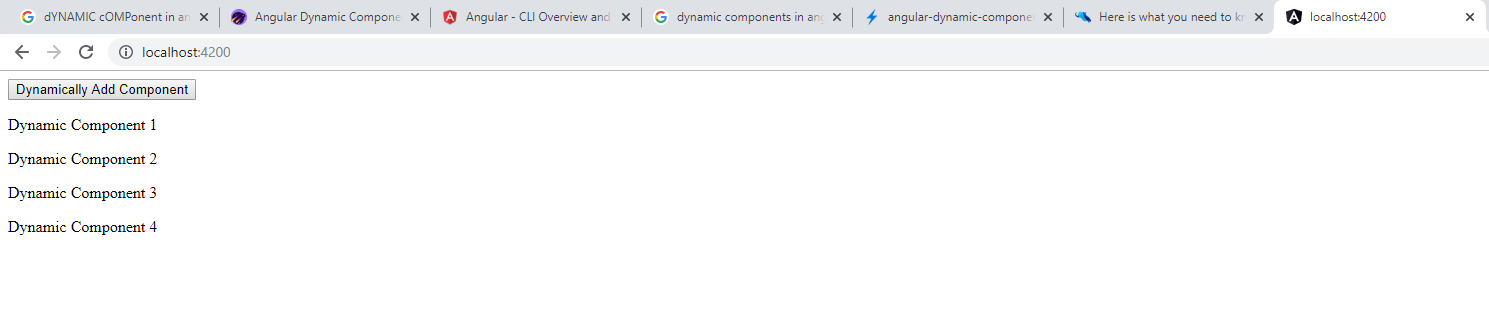
    // pass some data to the component

    componentRef.instance.index = this.\_counter++;

  }

}

Here we create the dynamic component after resolving it. Then output should be like as follows,



**Working with Model Class:**

Model are mainly used to store the data and makes the developer live significantly easier. The modelin an MVC-based application is generally responsible for modeling the data used in the view and handling user interactions such as clicking on buttons, scrolling, or causing other changes in the view. Let explain it with an example,

Let's assume we've got an API which returns our users,

GET /api/user

{

"status": "success",

"response": [

{

"id": 1,

"name": "John",

"car": {

"brand": "BMW",

"year": 2015

}

},

{

"id": 2,

"name": Bob",

"car": {

"brand": "Koenigsegg",

"year": 2014

}

}

]

}

We're going to create two very simple models; one for User and one for Car.

Our user model and another car model:

// src/app/shared/models/user.model.ts

import {Car} from "./car.model";

export class User {

id: number;

name: string;

car: Car;

}

// src/app/shared/models/car.model.ts

export class Car {

brand: string;

year: number;

}

These two objects will hold our data from the API. We're going to extend these models later, first let's create a service for getting our users:

// src/app/core/service/user.service.ts

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

import {Http, Response} from "@angular/http";

import 'rxjs/add/operator/map';

import {User} from "../../shared/models/user.model";

@Injectable()

export class UserService {

constructor(private http: Http) {}

getUser() {

return this.http.get('/api/user')

.map((res: Response) => res.json().response);

}

}

Calling getUser() now results in:

(2) [Object, Object]

[

{ id: 1, name: "John", car: Object },

{ id: 2, name: "Bob", car: Object }

]

But that's not exactly what we wanted. We want to get an array of User objects from our service. Let's do this.

We want to deserialize our JSON to our objects. Let's create an interface which provides an API for deserialization:

// src/app/shared/models/deserializable.model.ts

export interface Deserializable {

deserialize(input: any): this;

}

Now we can extend our models and implement our interface. Let's start with the User model:

// src/app/shared/models/user.model.ts

import {Car} from "./car.model";

import {Deserializable} from "./deserializable.model";

export class User implements Deserializable {

id: number;

name: string;

car: Car;

deserialize(input: any) {

Object.assign(this, input);

return this;

}

}

The interesting part here is the deserialize method. Basically, we're just assigning the input object to this - or, in other words, we're merging the input object with the User object.

But there's still one minor issue here: the car member won't be an instance of Car but still be an Object. We need to tell our deserialize method this manually:

deserialize(input: any): User {

Object.assign(this, input);

this.car = new Car().deserialize(input.car);

return this;

}

And, of course, we now need to implement our Deserializable interface for Car too:

// src/app/shared/models/car.model.ts

import {Deserializable} from "./deserializable.model";

export class Car implements Deserializable {

brand: string;

year: number;

deserialize(input: any): this {

Object.assign(this, input);

return this;

}

}

Now we can go back to our service and tell it what we want to get: we want to get an array of User, not just objects:

// src/app/core/service/user.service.ts

getUser(): Observable<User[]> {

return this.http.get('/api/user')

.map((res: Response) => res.json().response.map((user: User) => new User().deserialize(user)));

}

If, Users have a firstName and a lastName. If you're handling the raw JSON you'll have to print out the full name of your user within your templates like this:

<ul>

<li \*ngFor="let user of users">{{ user.firstName }} {{ user.lastName }}</li>

</ul>

One day your customer calls and tells that the order of first- and lastname should be switched. An endless joy which can be done by a (skilled) potato, since all you need to do is to go through every template and switch the expressions.

But if your user is a User object, you can simply implement a function to print the fullname:

getFullName() {

return this.firstName + ' ' + this.lastName;

}

// Just another example, assuming our Car class does implement a `isSportsCar` method

hasSportsCar() {

return this.car.isSportsCar();

}

You can now simply call this function in your template:

<ul>

<li \*ngFor="let user of users">{{ user.getFullName() }}</li>

</ul>

And whenever a change is required you need to change one single line. Simple, but effective.

Another good reason for using models like this is that we're working with **Type**script. We want to know the type of things when we use them and not just define everything as any. In combination with a good IDE this makes life a lot easier.

Of course, this can be used for handling forms too:

form: FormGroup;

createForm() {

this.form = this.fb.group({

id: null,

name: ['', Validators.required],

car: null

});

}

private prepareSave(): User {

return new User().deserialize(this.form.value);

}

onSubmit() {

const user = this.prepareSave(); // `user` is now an instance of "User"

// this.http.post('/api/user', user)...

}

**Nested/Child Components:**

The aspects of web frameworks like Angular is the idea of “componentizing” all the various parts of an application. Angular separates an application into different components can help improve code management and reuse.

Breaking up an application into multiple logical components makes it easier to:

* Architect an application as it grows in complexity.
* Re-use common components in multiple places.

A Component in Angular can have child components. Also, those child components can have their own further child components. Finally, Angular seamlessly supports nested components.

A typical webpage is normally break down into a set of logical components, for example header, footer and perhaps a sidebar. Let's first built a Child component in an example app which is Friend list component.

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

@Component({

selector: 'my-friends',

template: `<div>

<table>

<tr \*ngFor='let friend of friends'>

<td></td>

<td>

<img

src="http://witspry.in/ContentServer/Images/User/user.png"

width="20" />

</td>

</tr>

</table>

</div>`

})

export class MyFriendsComponent{

friends: Array<string> = [

'Friend 1',

'Friend 2',

'Friend 3',

'Friend 4',

'Friend 5',

];

}

Lets build its parent component i.e app component as follows,

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

import {MyFriendsComponent} from './nested-comp/nested-comp.component';

@Component({

    selector: 'app-root',

    template:`<img [src]="imgURL" width="50"/>

              <h3>My Friends</h3>

              <my-friends></my-friends>`

})

export class AppComponent{

  imgURL: string;

  constructor(){

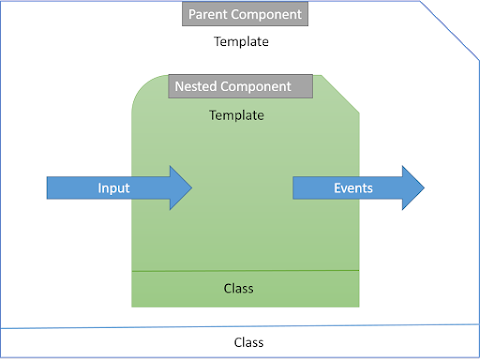
    this.imgURL = 'http://witspry.in/ContentServer/Images/User/user\_circle.png';

  }

}

Here, in template we include the child component <my-friends></my-friends>. Now, in main app import both component and add them in the "declarations" array.

**Passing data Using @Input() decorator in between parent and child component:**



Add a variable name as a following line with @Input() decorator.

export class MyFriendComponent{

@Input() name : string;

}

We can include that in child component as follows,

<my-friend [name]='friend'></my-friend>

Here we are passing a friend name to the "name" property of the child component.

## **Raising an event using @Output() decorator:**

The @Output directive enables a child component to use its properties in the parent component. Let’s modify our MyFriend component as:

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

@Component({

selector: 'my-friend',

template: `<div>

<table>

<tr>

<td></td>

<td>

<img src="http://putiw.xyz/clashroyaleunlimitedgems/img/user.png"

width="20" />

</td>

<td><button (click)='OnClick()'>Ping</button></td>

<td> </td>

</tr>

</table>

</div>`

})

export class MyFriendsComponent{

@Input() name : string;

timesPinged: number = 0;

@Output() pingClicked: EventEmitter<string> = new EventEmitter<string>();

OnClick(){

this.timesPinged++;

this.pingClicked.emit('You pinged ' + this.name + ' ' +

this.timesPinged + (this.timesPinged == 1? ' time' : ' times'));

}

}

Here, we have imported Output and EventEmitter from Angular core library. Also, we have defined an event using @Output() decorator. The OnClick is a local function which is invoked from button's click event. Since the "pingClicked" is decorated with @Output(), it's emit function is able to notify the parent component.

Now, change the parent component to capture the raised event from its child component as below:

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

import {MyFriendsComponent} from './nested-comp/nested-comp.component';

@Component({

    selector: 'app-root',

    template:`

            <img [src]="imgURL" width="50"/>

            <br />

            <br />

            Total Pings today : {{totalPings}}

            <h3>My Friends</h3>

            <div \*ngFor="let friend of friends">

              <my-friend [name]='friend'

                        (pingClicked)='onFriendPingClicked($event)'>

              </my-friend>

            </div>

        `

})

export class AppComponent{

  imgURL: string;

  friends: string[];

  totalPings: number;

  pingMessage: string;

  constructor(){

    this.imgURL =

 'http://witspry.in/ContentServer/Images/User/user\_circle.png';

    this.friends = [

      'Friend A',

      'Friend B',

      'Friend C',

      'Friend D',

      'Friend E'

    ];

    this.totalPings = 0;

    this.pingMessage = '';

  }

  onFriendPingClicked(pingMessage: string): void{

    this.totalPings++;

    this.pingMessage = pingMessage;

  }

}

**Working with Arrays / Collections:**

An array is a user-defined data type. An array is a homogenous collection of similar types of elements that have a contiguous memory location, and which can store multiple values of different data types.

An array is a type of data structure that stores the elements of similar data type and consider it as an object too. We can store only a fixed set of elements and can’t expand its size, once its size is declared.

Features of Array,

* An array declaration allocates sequential memory blocks.
* Arrays are static. This means that an array once initialized cannot be resized.
* Each memory block represents an array element.
* Array elements are identified by a unique integer called as the subscript / index of the element.
* Like variables, arrays too, should be declared before they are used. Use the var keyword to declare an array.
* Array initialization refers to populating the array elements.
* Array element values can be updated or modified but cannot be deleted.

**Advantages**

* Code Optimization: We can retrieve or sort the array data more efficiently.
* Random access: We can randomly access the array data using the location pointer.

**Disadvantages**

* Size Limit: The size of an Array is fixed i.e, static. We cannot increase the array size once it has been declared.

**There are two types of an array:  
1. Single-Dimensional Array  
2. Multi-Dimensional Array**

**Single-Dimensional Array: Syntax for single dimensional array,**

let array\_name[:datatype] = [val1, val2, valn..]

(or)

let array\_name: Array = [val1, val2, valn..]

For Example,

var values: (string | number) [] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];  // here array can be single or multiple data types.

(or)

var values: Array = ['Apple', 2, 'Orange', 3, 4, 'Banana'];

**Multi-Dimensional Array:**  
The data is stored in rows and columns (also known as matrix form) in a Multi-dimensional array.  
**Syntax:**

let arr\_name:datatype[][] = [ [a1, a2, a3], [b1, b2, b3] ];

**Initialization:**

let arr\_name:datatype[initial\_array\_index][referenced\_array\_index] = [ [val1, val2, val 3], [v1, v2, v3]];

**Array Object**:

We can create an Array by using or initializing the Array Object. The Array constructor is used to pass the following arguments to create an Array,

**Syntax:**

let arr\_name:datatype[] = new Array(values);

We can perform different method on arrays like concat(), filter(), foreach(), indexOf(), map(), push(), pop(), splice(), sort(), etc.

**Data Binding**

**INTRO:**

Data binding in Angular works by synchronizing the data in the components with the UI so that it reflects the current value of the data to achieve the synchronization of the View and the Model.

**Binding properties and Interpolation:**

String Interpolation uses template expressions in double curly {{ }} braces to display data from the component, the special syntax {{ }}, also known as moustache syntax. The {{ }} contains JavaScript expression which can be run by Angular and the output will be inserted into the HTML.

Say if we put ***{{ 5 + 5 }}*** in the template ***10*** will be inserted into the HTML

**Helloworld.component.ts**

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

@Component({

    selector: 'app-hello',

    template: `

                <h3>{{welcomeMessage}}</h3>

              `,

    styles: [`

            h3 {

                font-size:28px;

            }`]

})

export class HelloWorldComponent implements OnInit {

    welcomeMessage = 'Welcome to Angular Data Binding Example!!';

  constructor() {

  }

  ngOnInit() {

   }

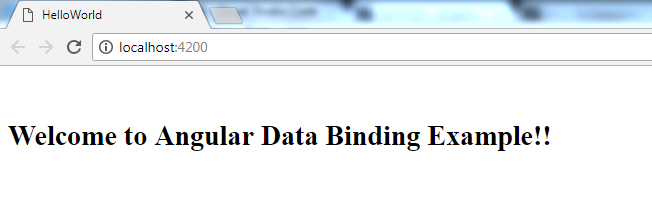
}

In the above code, we have created our HelloWorldComponent and initialized a string “welcomeMessage”, We are binding the “welcomeMessage” string to the DOM using String Interpolation in the template meta-data.

When the HelloWorldComponent is rendered on a page with the above template, it prints the value of “welcomeMessage” on the view. Below goes the output

**app.component.html**

 <app-hello></app-hello>

**Output:**

**One-way Binding / Property Binding:**

The square braces are used to bind data to a property of an element, the trick is to put the property onto the element wrapped in brackets: [property].

The src property of the HTML Element img is bound to the srcURL property of the class. Whenever the srcURL property changes the src property of the img element changes.

Also, property binding could be used to pass data to a component. Let’s say you want to pass data to people component, we will do something like this:

@Component({

selector: "app-root",

template: `

➥ <people [person]="title"></people>

<button (click)="changeTitle()">ChngTitle</button>

`,

styles: []

})

export class AppComponent {

title = "app";

constructor() {}

changeTitle() {

this.title = "Angular app";

}

}

The people element is a Component so Angular will evaluate title and pass it into the people component via person attribute. Angular components can receive values from the outside world and can send values to the outside world. If an Angular component needs to receive values from the outside world, you can bind the producers of these values to the corresponding inputs of the component.

For people component to receive we have to decorate the person property with a @Input decorator.

@Component({

selector:'people',

template: `

{{person}}

`

})

export class PeopleComponent {

➥ @Input()person

}

Now, whenever the title property from the parent component (AppComponent) changes, the changed is communicated to the child component (PeopleComponent) via the person property binding and the changes is reflected on the DOM. This is also called Parent-Child communication which is a one-way event. If it was Child-Parent communication it would be two-way communication.

**Event Binding:**

In Angular, we can register and capture an event from an element by wrapping the event in a () parenthesis.

In event binding, we are binding an event handler for the target event. Whenever we perform some operations, an event will be raised. The event handler then executes the template statement. The template handler will have a receiver, which will perform the operation based on the event received and then respond.

We can bind most of the common events in the DOM:

(focus)="focusCallback()"

(blur)="blurCallback()"(submit)="submitCallback()" (scroll)="scrollCallback()"(cut)="cutCallback()"

(copy)="copyCallback()"

(paste)="pasteCallback()"(keydown)="keydownCallback()"

(keypress)="keypressCallback()"

(keyup)="keyupCallback()"(mouseenter)="mouseenterCallback()"

(mousedown)="mousedownCallback()"

(mouseup)="mouseupCallback()"(click)="clickCallback()"

(dblclick)="dblclickCallback()"(drag)="dragCallback()"

(dragover)="dragoverCallback()"

(drop)="dropCallback()"

@Component({

selector: 'app-root',

template: `<button (click)="clickEvent()">Click Me</button>`

})

export class AppComponent{

clickEvent() {

console.log('You clicked me')

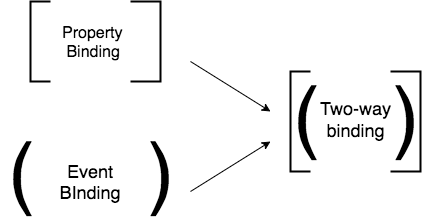
}

}

Click event is attached to the Click Me button element using (). The right-hand side ="expression" specifies the callback function that will be run when the registered event is fired by the element. In our case, we have the clickEvent function attached.

**Two-way Binding:**

With two-way data binding, the framework (angular) is not only watching your variables for changes. It also keeps track of changes that are made by the user (for example with input-elements) and updates the variables accordingly.



**Two-way Binding with NgModel:**

NgModel can be used with form-elements like inputs to implement two-way data binding. To do that, we have to use a pretty special syntax: [(ngModel)]. It’s a combination of the one-way- and the event binding syntax.

NgModel is world-famous “banana in a box” syntax.

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

What’s interesting is that the [()] turns out to be syntactic sugar for the following:

<input [ngModel]="title" (ngModelChange)="title= $event" type="text">

The above code is definitely more verbose(using) but it makes perfect sense when you think about it. A two-way data-binding is really… two one-way data-bindings, right?

Now the real secret that the above example unveils is the use of the *Change* suffix.

The property binding [ngModel] takes care of updating the underlying input DOM element.

The event binding (ngModelChange) notifies the outside world when there was a change in the DOM.

**Attribute Binding:**

Attribute binding is used to bind an attribute property of a view element.

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>

  <tr>

    <td colspan="{{4}}"> </td>

  </tr>

</table>

This will throw an error "Template parse errors: Can't bind to 'colspan' since it isn't a known native property".

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 attar, followed by a dot (.), and the name of the attribute. You then bind the attribute value using an expression that resolves to a string.

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

@Component({

  selector: 'app-example',

  template: `

             <div>

             <table>

             <tr><td [attr.colspan]="3">three</td></tr>

             <tr><td>1</td><td>2</td><td>3</td></tr>

             </table>

             </div>

             `

})

export class ExampleComponent {

}

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.

**Class Binding:**

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.

The class binding syntax is also like property binding. In property binding, we only specify the element between brackets. But in the case of class binding, it starts with the prefix class, followed by a dot (.), and the name of the class. You then bind the class value with CSS class name like class.class-name

The example below shows the standard way of setting class attribute without binding. In this case, we are setting a class attribute with a class name 'myClass' without binding.

<div class="myClass">Setting class without binding</div>

The example below shows setting all the class values with binding. In this case, we are binding class "myClassBinding" with class binding.

<div class="myClass" [class]="myClassBinding">Setting all classes with binding</div>

Whenever the template expression evaluates to true, Angular binds that class name to the class binding. It removes the class when the template expression evaluates to false.

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

@Component({

  selector: 'app-example',

  template: `

             <div>

             <h1 [class.myClass]="isTrue">This class binding is for true value</h1>

             <h1 [class.myClass]="!isTrue">This class binding is for false value</h1>

             </div>

             `

})

export class ExampleComponent {

   isTrue: boolean = true;

}

In the above example we are binding a specific class name with class binding. In this case, if 'isTrue' value evaluates to true then it will bind the 'myClass' to a class property. If it evaluates to false, then it will not bind the 'myClass' to a class property.

While the class binding is a fine way of binding, the ngClass directive is preferred for handling multiple class names at the same time.

**Style Binding:**

Style binding is used to set a style of a view element. We can set inline styles with style binding.

Like with class and attribute 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.

In the below example, we are binding a color style to the 'h1' element. It will display the text within the h1 tags in a blue color.

Style bindings will also have the unit. In the example below, we are setting style font size in "px" and "%" units.

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

@Component({

  selector: 'app-example',

  template: `

             <div>

             <span [style.font-size.px]="isTrue? 20 : 12">This style binding is set for true value</span>

             <span [style.font-size.%]="!isTrue : 120 : 30">This style binding is set for false value</span>

             </div>

             `

})

export class ExampleComponent {

   isTrue: boolean = true;

}

## Styles Binding in Components

**Style and Class Bindings:**

**Class Bindings:** It is going to assign a class to HTML elements, which is based on an expression, which evaluates as true or false. Let’s create a div tag, as shown below.

@Component({

  selector: 'app-root',

  template: `<h1>{{title}}</h1>

          <h2>Class & Style Binding</h2>

          <div [class.myClass] = "applyclass">Applying class Bindings</div>`,

  styles:[`.myClass{

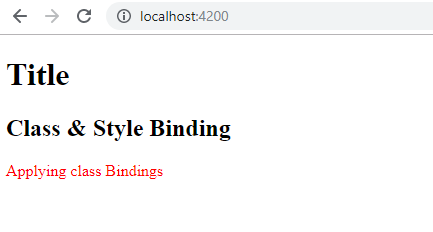
    color:red;

  }`]

})

Here, my class to apply is myclass, as shown above. The class is going to be assigned to div tag, which is based on a variable of true or false.

When we save and run the Application, the output will be, as shown below



**Style Bindings:** That’s how class binding works. Similarly, we have style binding in order to apply inline styles to HTML elements.

Let’s create another property,

<div [style.color] ="applyblue ? 'blue' : 'orange'">Applying Style Bindings</div>

Add another variable, as shown below:

export class AppComponent {

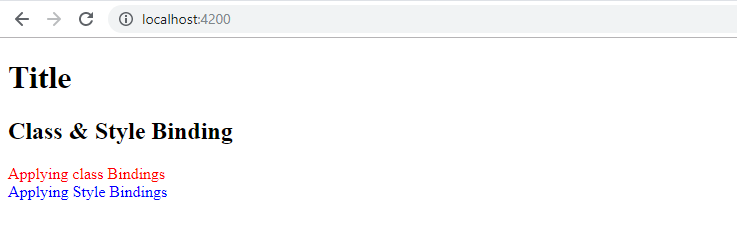
  public title = 'ang-style-class-bind';

  public applyclass = true;

public applyblue = true;

}

Then output will be as follows,

**NgStyle & NgClass (Build-In Directives):**

Both the NgStyle and NgClass directives can be used to conditionally set the view of an application.

NgStyle gives you fine grained control on individual properties. But if you want to make changes to multiple properties at once, creating a class which bundles those properties and adding the class with NgClass makes more sense.

**NgClass:**

Most of the styles that we need to apply are always present i.e., static, and can be simply be applied as standard HTML in our templates, like this:

<button class="btn btn-primary">Button</button>

But there are often styles that are applied conditionally to our templates, they are added to an element only if a certain programmatic condition is met.

Note that many state styles can be natively implemented using browser CSS pseudo-classes, such as for example:

* styles for identifying an element with the focus, via the :focus pseudo class.
* hover styles and on-click active state styles (using :hover and :active).

For these type of state styles natively supported by the browser, it's better to use the CSS pseudo classes whenever possible. So, for these very common cases we won't need ngClass.

But there are many other state styles that are not natively supported by the browser. These styles could for example include:

* styles for identifying the currently selected elements of a list.
* styles for identifying the currently active menu entry in a navigation menu.
* styles to identify a certain feature of an element; for example, to identify a new element in an e-commerce site.

If the element that we are styling only has one of those state styles, we can even apply it simply by using the plain input property template syntax, without any extra directive.

<p>Default Button:</p>

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

<p>Equivalent example using Button:</p>

<button class="btn"

        [class.btn-primary]="true"

        type="submit">Button

</button>

Notice the syntax [class.btn-primary] that is activating the btn-primary CSS class, effectively adding it to the button.

This expression will add or not the class to the element depending on the truthiness of the expression, which in this case is always true.

The ngClass directive will take an expression that will be used to determine which state styles to apply at a given time to the styled element.

The expression passed on to ngClass can be:

* an object
* an array
* a string

Let's go over each one of these 3 cases with examples, and then see how we can make sure that we can keep our templates light and readable.

**Passing an array of css classes using ngClass:** One way of defining what classes should be active at a given moment is to pass an array of strings to the ngClass directive. For Example,

**<p>Passing an Array of classes:</p>**

**<button [ngClass]="['btn', 'btn-primary']">Button</button>**

**Passing a string of CSS classes to ngClass:** It’s also possible to pass to ngClass a string, that contains all the CSS classes that we want to apply to a given element:

This syntax would give the same results as before, meaning that the two CSS classes btn and btn-primary would still be applied.

**<p>Passing a string:</p>**

**<button [ngClass]="'btn btn-primary'" type="submit" (click)="submit()"> Button </button>**

**Passing a configuration object to ngClass:** The last and commonly used way is that configure ngClass by passing an object.

* The keys of that object are the names of the CSS classes that we want to apply.
* The values of the configuration object should be booleans (or an expression that evaluates to a boolean) that indicate if the CSS class should be applied or not.

**<p>Passing a configuration object:</p>**

**<button [ngClass]="{ btn:true, 'btn-primary':true }">Button</button>**

**NgStyle:**

ngStyle becomes much more useful when the value is dynamic. The values in the object literal that we assign to ngStyle can be javascript expressions which are evaluated, and the result of that expression is used as the value of the css property.

For Example,

<div [ngStyle]="{'background-color': 'green'}"></<div>

<div [ngStyle]="{'background-color': person.country === 'UK' ? 'green’: 'red' }"> </<div>

# **Difference between ng-class and ng-style directives:**

ng-style is used to interpolate javascript object into style attribute.

ng-style="{width: 10px}"

The above directive will be translated to style="width:10px".

ng-class directive translates your object into class attribute.

ng-class="{'highlight-class': isActive}"

The above template is translated to class="highlight-class" when isActive variable is true.

# **Angular Component Styles:**

For every angular component, we will define  HTML template, CSS styles that go with that template, specifying any selectors, rules, and media queries that needed. The style sheet may apply in different ways. Those are,

1. Component Inline style
2. Component External Style
3. Template using link directive
4. Template using style directive

The Angular Components maintain their own style & state. But CSS styles are global in scope. The angular encapsulates the component styles using the view encapsulation strategies. Therefore, ensuring that the styles of one component does not bleed to another view.

**Component Inline Style:**

Use the styles metadata of the @Component or @Directive to specify the CSS rules as shown below. Here we can write multiple styles by separating with comma.

@Component({

   selector: 'app-test1',

   templateUrl: './test1.component.html',

  styles: [

     `p { color:blue}`,

`h1 {color:blue}`

   ],

})

**Component External Style:**

In External Styles specify the external style sheets using the styleUrls meta data of the @Component decorator or @Directive decorator. Here also we can add multiple style seat.

@Component({

   selector: 'app-test2',

   templateUrl: './test2.component.html',

  styleUrls: ['./test2.component.css', './another.component.css ']

})

We can Specify Component Inline & External styles together. Similarly for templates also we can styles as shown below.

1. Component inline styles i e. Styles defined at @Component.styles
2. Component External styles i.e. @Component.styleUrls
3. Template Inline Styles using the Style tag
4. Template External Styles using the link tag

**Special selectors:**

**Selector:** The @Component decorator functions like an object, and this object contains a lot of properties. ‘Selector’ is also one of the properties which is Inherited from Directive decorator.

The selector attribute allows us to define how Angular is identified when the component is used in HTML. It tell Angular to create and insert an instance of this component where it finds the selector tag in the Parent HTML file in your angular app.

The selector accepts a string value. We will use it in html to place this component where we want.

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

@Component({

     selector: 'app-root',

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

     template: `

         <h2>ngClass Examples</h2>`

})

Here are some ways that are used to specify the selector attribute and how to use it in HTML.

* element-name: Select by element name. Ex: <app-root></app-root>
* .class: Select by class name. Ex: <div class=”app-root”></div>
* [attribute]: Select by attribute name. Ex: <div app-root></div>

**Special Selector:**

Component styles have a few special selectors from the world of shadow DOM style scoping. The following sections describe these selectors.

### **:host :** Using the :[host](https://angular.io/api/core/Directive#host) pseudo-class selector to target styles in the element that hosts the component (as opposed to targeting elements inside the component's template).

:[host](https://angular.io/api/core/Directive#host) { display: block; border: 1px solid black; color: red }

The: host selector is the only way to target the host element. You can't reach the host element from inside the component with other selectors because it's not part of the component's own template. The host element is in a parent component's template.

The next example targets the host element again, but only when it also has the active CSS class.

:[host](https://angular.io/api/core/Directive#host)(.active) { border-width: 3px; }

**:host-content :** Sometimes it's useful to apply styles based on some condition outside of a component's view. For example, a CSS theme class could be applied to the document <body> element, and you want to change how your component looks based on that.

Use the :host-context() pseudo-class selector, which works just like the function form of :[host](https://angular.io/api/core/Directive#host)(). The :host-context() selector looks for a CSS class in any ancestor of the component host element, up to the document root. The :host-context() selector is useful when combined with another selector.

The following example applies a background-color style to all <h2> elements inside the component, only if some ancestor element has the CSS class theme-light.

:host-context(.theme-light) h2 { background-color: #eef; }

### **:host /deep/, >>>, and ::ng-deep:** Component styles normally apply only to the HTML in the component's own template.

Any style with ::ng-deep applied becomes a global style. In order to scope the specified style to the current component and all its descendants, be sure to include the :host selector before ::ng-deep. If the ::ng-deep combinator is used without the :host pseudo-class selector, the style can bleed into other components.

The following example targets all <h3> elements, from the host element down through this component to all its child elements in the DOM.

:host /deep/ h3 {

   font-style: italic;

}

[**Loading Styles into Components**](https://angular.io/docs/ts/latest/guide/component-styles.html#loading-styles)**:**

In Angular Projects, each component has its own style sheet (CSS, SCSS, LESS, etc.). It’s quite often that might include global styling files in your component.

So that we can import that global file in any component we need. Let create a global styles folder inside src folder. Create a file variables.scss as shown below,

// In\_variables.scss file

$brand-color: #800000;

The $brand-color variable is in stylings/\_variables.scss file. We need to import the file in order to use it:

// hello.component.scss

@import "../../../stylings/variables";

h1 {color: $brand-color;}

If we have to repeat this ../../../stylings/ in another tens or hundreds of components and you need to remember the relative path. This is not cool, in order to avoid this use short cut with angular CLI.

If project is generated with Angular CLI,then add a configuration stylePreprocessorOptions > includePaths in. angular.cli.json file. This configuration allows to add extra base paths that will be checked for imports. It tells Angular CLI to look for styling files in the mentioned paths before processing each component style file.

{...

"apps": [{

"root": "src",

...

"stylePreprocessorOptions": {

"includePaths": [

"./stylings"

]}

}]

}

So that we can update hello.componet.css as follows,

@import "variables"; // change to just variables,

h1 {color: $brand-color;}

**View Encapsulation:**

View encapsulation defines whether the template and styles defined within the component can affect the whole application or vice versa. By default, styles for our components are encapsulated, that means that they don’t leak out and affect the rest of the application. Angular provides three encapsulation strategies.

**Emulated (default)** - Styles from main HTML propagate to the component. Styles defined in this component's @Component decorator are scoped to this component only.

**Native** - Styles from main HTML do not propagate to the component. Styles defined in this component's @Component decorator are scoped to this component only.

**None** - Styles from the component propagate back to the main HTML and therefore are visible to all components on the page. Be careful with apps that have None and Native components in the application. All components with None encapsulation will have their styles duplicated in all components with Native encapsulation.

By default, the renderer uses ViewEncapsulation.Emulated if the view has styles, otherwise ViewEncapsulation.None. There is also a ViewEncapsulation.Native method which uses the shadow DOM to encapsulate the view.

For Example,

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

@Component({

   moduleId: module.id,

   selector: 'app-root',

   templateUrl: 'app.component.html',

   styles: [`

     .card {

       background: gray;

     }

   `],

   encapsulation: ViewEncapsulation.None

})

class AppComponent {

   @Input() title: string;

}

**ViewChild & ViewChildren:**

The @ViewChild and @ViewChildren decorators in Angular provide a way to access and manipulate DOM elements, directives and components.

The use of ViewChild is to query one element from the DOM where as ViewChildren is for multiple elements.

**ViewChild:** It is a decorator that creates a view or DOM query by taking following meta information.

* Selector - The selector of the element to query. This can be a directive type or a name. (ex: i.e. components and directives, Template reference variables, Providers)
* Read - To read a different token from the queried elements.
* Static - It indicates whether to resolve query results before change detection runs.

let's see how we can query the child component using ViewChild. Inside app.component.ts:

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

import { HelloComponent } from './hello.component';

@Component({

selector: 'my-app',

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

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

})

export class AppComponent implements AfterViewInit {

name = 'Angular';

@ViewChild(HelloComponent, {static: false}) hello: HelloComponent;

ngAfterViewInit() {

console.log('Hello ', this.hello.name);

}

}

OUTPUT: In the console, you should get ‘**Hello Angular’.**

Here we imported HelloComponent and ViewChild and AfterViewInit from the @angular/core package.

Next, we create a query called hello that takes HelloComponent as the selector and has static equals to false.

Next, in the ngAfterViewInit() life-cycle hook, we can use the query to access the DOM element for the hello component.here, we accessed the name property of the component, after it's mounted in the DOM, which contains the Angular string. We can access any properties and even methods from the queried component.

## Querying Standard HTML Elements with Template References: We can also query standard HTML elements using ViewChild and template reference variables.

## Change src/app/app.component.html file it as follows:

<hello name=""></hello>

<p #pRef>

Start editing to see some magic happen :)

</p>

let's change src/app/app.component.ts to access the component using its reference.

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

import { HelloComponent } from './hello.component';

@Component({

selector: 'my-app',

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

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

})

export class AppComponent implements AfterViewInit {

name = 'Angular';

@ViewChild(HelloComponent, {static: false}) hello: HelloComponent;

@ViewChild('pRef', {static: false}) pRef: ElementRef;

ngAfterViewInit() {

console.log(this.pRef.nativeElement.innerHTML);

this.pRef.nativeElement.innerHTML = "DOM updated successfully!!!";

}

}

We import ElementRef and we create a query configuration to access the <p> DOM element with the #pRef template reference as follows:

Next, in the ngAfterViewInit() method we can access and modify the native DOM element using the nativeElement object of ElementRef.

## ViewChildren: It is another property decorator which is used to query the DOM for multiple elements and return a query list. Any time a child element is added, removed, or moved, the query list will be updated, and the changes observable of the query list will emit a new value.

For example, Update src/app/app.component.html as follows:

<hello name="Angular 6"></hello>

<hello name="Angular 7"></hello>

<hello name="Angular 8"></hello>

Here we are diplsaying the hello component three times. Let's now query the DOM. Open the src/app/app.component.ts file and change it as follows:

import { Component, ViewChildren, AfterViewInit, QueryList } from '@angular/core';

import { HelloComponent } from './hello.component';

@Component({

selector: 'my-app',

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

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

})

export class AppComponent implements AfterViewInit {

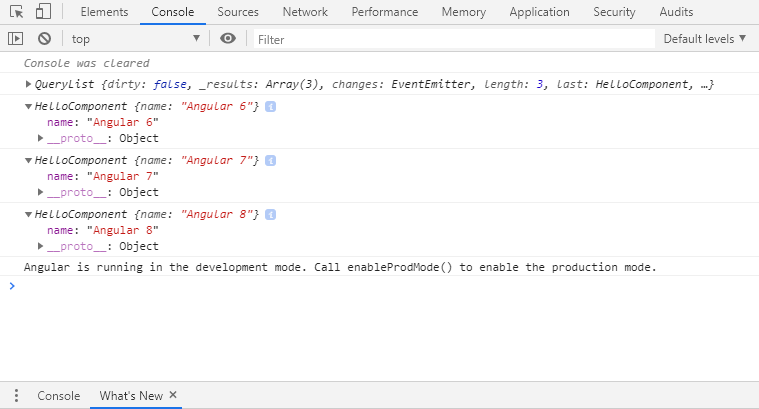
name = 'Angular';

@ViewChildren(HelloComponent) hellos: QueryList<any>;

ngAfterViewInit() {

this.hellos.forEach(hello => console.log(hello)); } }

OUPUT: Inside console we can see the query list as follows.



**ContentChild & ContentChildren:**

There are some cases when a parent component needs access to his children. For that we use ContentChild and ContentChildren.

The concept of a content child is similar to that of a view child.

* ViewChild **don’t** include elements that exist within the ng-content tag.
* ContentChild includes **only** elements that exists within the ng-content tag.

To get a reference to that child we can use either the @ContentChild or the @ContentChildren decorators. Here, @ContentChild returns one child and @ContentChildren returns a QueryList.

Just like in previous (view child), one of the component lifecycle hooks need to use, this time it’s AfterContentInit.

For Example:

@Component({

  selector: 'tab',

  template: `

    <p>{{title}}</p>

  `,

})

export class TabComponent {

  @Input() title;

}

@Component({

  selector: 'tabs',

  template: `

    <ng-content></ng-content>

  `,

})

export class TabsComponent {

 @ContentChildren(TabComponent) tabs: QueryList<TabComponent>

 ngAfterContentInit() {

   this.tabs.forEach(tabInstance => console.log(tabInstance))

 }

}

@Component({

  selector: 'my-app',

  template: `

    <tabs>

     <tab title="One"></tab>

     <tab title="Two"></tab>

    </tabs>

  `,

})

export class App {}

**Template Driven Forms**

**Introduction:**

Forms are playing an important role in many aspects(like scheduling a meeting, placing an order, booking for travelling) from so long.

Template Driven froms are used when our application needs very basic for and logic.

These are called template-driven as everything that we are going to use in the application is defined in the template that we are defining along with the component.

Behind the scenes, template-driven forms are converted to the model-driven equivalent by Angular, to the internal functioning is the same.

We need to import the FormsModule into the Application module file (i.e. **app.module.ts**).

FormsModule: Exports the required providers and directives for template-driven forms, making them available for import by NgModules that import this module.

Do the Import from: @angular/Forms.

@angular/Forms: Implements a set of directives and providers to communicate with native DOM elements when building forms to capture user input.

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

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

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

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

@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule,

    FormsModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

**Create the component that controls the form:**

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

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

@Component({

  selector: 'app-template-form',

  templateUrl: './template-form.component.html',

  styleUrls: ['./template-form.component.css']

})

export class TemplateFormComponent implements OnInit {

  constructor() { }

  ngOnInit() {

  }}

**Create a template with the initial form layout:**

<form  #regForm='ngForm'class="p-5">

    <div class="row">

        <div class="form-group col-xs-5 col-lg-2">

            <label>First Name</label>

            <input type="text"

                class="form-control">

        </div>

        </div>

      <div class="row">

        <div class="form-group col-xs-5 col-lg-2">

            <label>Last Name</label>

            <input type="text"

                class="form-control">

        </div>

      </div>

    <div class="row">

        <div class="form-group col-xs-5 col-lg-2">

            <label>Email</label>

            <input type="email"

                class="form-control">

        </div>

    </div>

    <div class="row">

        <div class="form-group col-xs-5 col-lg-2">

            <label>Password</label>

            <input type="password"

                class="form-control">

        </div>

    </div>

    <div class="row">

        <div class="form-group col-xs-5 col-lg-2">

            <label>Language</label>

            <select class="form-control">

                <option value="">Please select a language</option>

                <option \*ngFor="let lang of langs"

                        [value]="lang">

                        {{lang}}

                </option>

            </select>

        </div>

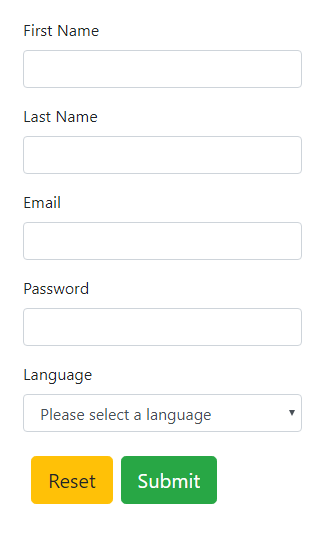
    </div>

    <button class="btn btn-warning btn-lg btn-radius m-2">Reset</button>

    <button class="btn btn-success btn-lg btn-radius">Submit</button>

  </form>

**Output:**



**NgForm**

It is the directive which helps to create the control groups inside the form directive It is attached to the <form> element in the HTML and supplements the form tag with some additional features. Some interesting things we can say about the view is that whenever we use the directive in the application view we need to assign some selector to it; in this case, the form is the selector.

**Bind data properties to each form input control with the ngModel two-way data binding syntax:**

ngModel directive is that it lets us setup twoway data binding between a template form control and a variable on our component.

So when the user changes the value in the template form control the value of the variable on the component automatically updates and when we change the variable on the component the template form control automatically updates.

The NgModel directive creates the FormControl instance to manage the template form control and the name attribute tells the NgModel directive what *key* to store that FormControl under in the parent FormGroup.

    <div class="row">

        <div class="form-group col-xs-5 col-lg-2">

            <label>Email</label>

            <input type="email"

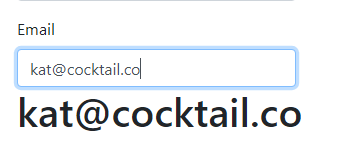
                class="form-control" [(ngModel)]="email">

                <h1>{{email}}</h1>

        </div>

    </div>

**Output:**



**Add the name attribute to each form input control:**

<div class="row">

        <div class="form-group col-xs-5 col-lg-2">

            <label>Email</label>

            <input type="email"

                class="form-control" name="email" [(ngModel)]="email">

                <h1>{{email}}</h1>

        </div>

    </div>

Similarly add name attribute for every input control.

**Show and hide validation error messages:**

**HTML:**

<div class="form-group col-xs-5 col-lg-2" [ngClass]="{

            'border-red': regForm.form.controls.email?.invalid && (regForm.form.controls.email?.dirty || regForm.form.controls.email?.touched),

            'border-green': regForm.form.controls.email?.valid && (regForm.form.controls.email?.dirty || regForm.form.controls.email?.touched)

       }">

            <label>Email</label>

            <input type="email"

                class="form-control" name="email" [(ngModel)]="email" required pattern="[^ @]\*@[^ @]\*">

</div>

**Shorter Validation Expressions:**

<div class="form-group col-xs-5 col-lg-2" [ngClass]="{

            'border-red': emailId.invalid && (emailId.dirty || emailId.touched),

            'border-green': emailId.valid && (emailId.dirty || emailId.touched)

       }">

            <label>Email</label>

            <input type="email"

                class="form-control" name="email" [(ngModel)]="email" required pattern="[^ @]\*@[^ @]\*" #emailId="ngModel">

        </div>

**CSS:**

.border-red input{

    border-color: red;

    border-width: unset;

}

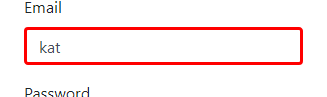
.border-green input{

    border-color: green;

    border-width: unset;

}

**Output:**





**Handle form submission with ngSubmit:**

We need a submit button, this is just button with a type="submit" somewhere between the opening and closing form tags.

**HTML:**

<form novalidate class="p-5" #regForm='ngForm' (ngSubmit)="Register(regForm)">

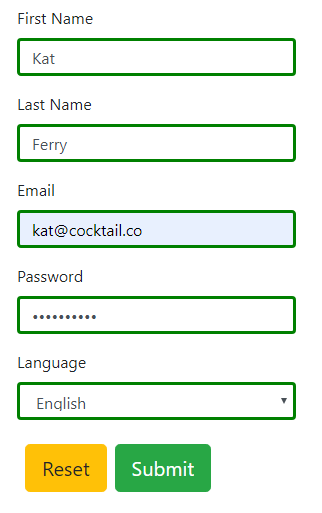
**TS:**

  Register(regForm:NgForm){

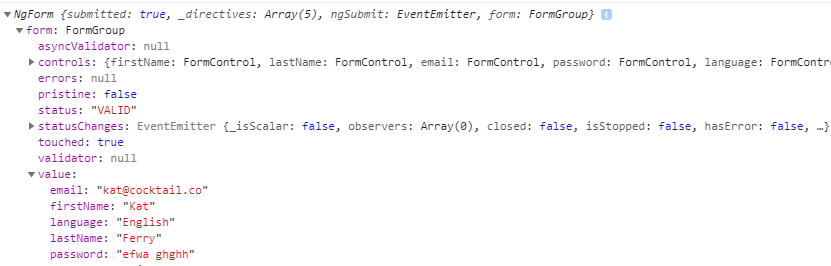
    console.log(regForm);

  }

**Output:**



On clicking submit button:



**Disable the form’s submit button until the form is valid:**

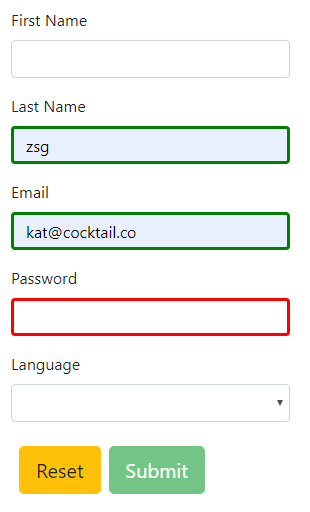
<button type="submit" class="btn btn-success btn-lg btn-radius" [disabled] ="regForm.invalid" >

  Submit

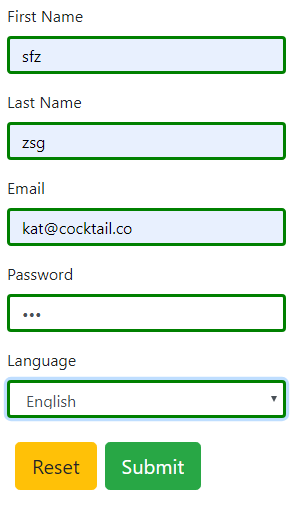
  </button>

**Output:**

When some fields are not filled.



When all the fields are filled according to their input patterns



**Resetting the form:**

Register(regForm:NgForm){

    console.log(regForm);

    regForm.reset();

  }

}

The above line "regForm.reset()" resets the form after form submission.

**Reactive Forms**

**Reactive Forms Introduction:**

When we talk about "reactive" forms (also known as model-driven), we’ll be avoiding directives such as ngModel and required. The idea is that instead of declaring that we want Angular to power things for us, we can actually use the underlying APIs to do them for us. In a sense, instead of binding Object models to directives like template-driven forms, we in fact boot up our own instances inside a component class and construct our own JavaScript models. This has more power and is extremely productive to work with as it allows us to write expressive code, that is very testable and keeps all logic in the same place, instead of scattering it around different form templates.

Each change to the form state returns a new state, which maintains the integrity of the model between changes.

Lets create a basic user registration form using Reactive forms.

First, import ReactiveFormsModule in app.module.ts and then add it to imports.

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

@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule,

    ReactiveFormsModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

Creating a layout for registration form with 3 fields.

<form [formGroup]="reactiveForm">

  <label class="label">Name</label>

  <input class="input"

         type="text"

         formControlName="name">

  <label class="label">Email</label>

  <input class="input"

         type="text"

         formControlName="email">

  <label class="label">Password</label>

  <input class="input"

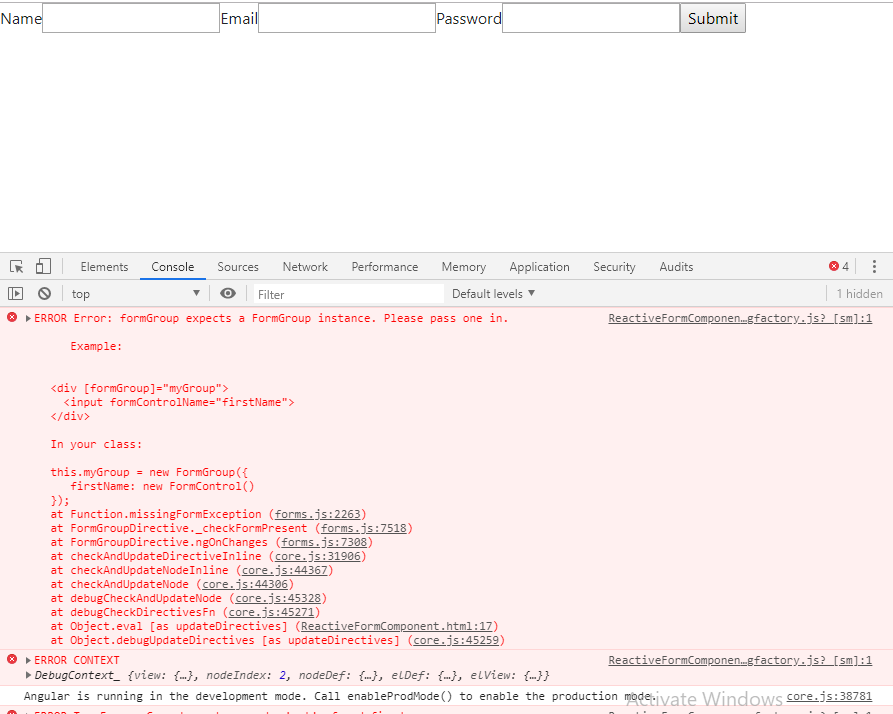
         type="password"

         formControlName="password">

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

</form>

[formGroup]="reactiveForm" - This directive allows us to group the FormControls within our form. We add the [formGroup] directive to our form element and give it a name such as reactiveForm.

Even form template is setup correctly, we will get to see this error.

To fix this error we need to add imports for FormBuilder and FormGroup to our component. Since FormBuilder is a service we inject that into our component’s constructor as well.

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 {

  reactiveForm: FormGroup;

  constructor(private formBuilder: FormBuilder) {

  }

  ngOnInit() {

  }

}

In addition to our imports, we also add a component property reactiveForm of type FormGroup. This property name, reactiveForm, must be the same name we provided in our template to the directive [formGroup].

ngOnInit() {

  this.createForm();

}

createForm() {

  this.reactiveForm = this.formBuilder.group({

    name: [''],

    email: [''],

    password: ['']

  });

}

Here we’ve created a createForm function that will create the FormGroup object using the FormBuilder service. We set our reactiveForm property to a new FormGroup instance by calling .group(). Within this we provide an object of FormControls.

The key for each FormControl maps to the names we provided in our template to formControlName. After each key is an array with its first value, an empty string, being the initial value for the form control.

Now when you go back to your browser you should have a form visible with no errors now that our FormGroup has been initialized.

**More Form Controls:**

The core idea behind form controls is the ability to access a control’s value. This is done with a set of directives that implement the ControlValueAccessor interface.

ControlValueAccessor is an interface for communication between a FormControl and the native element. It abstracts the operations of writing a value and listening for changes in the DOM element representing an input control.

interface ControlValueAccessor {

  /\*\*

  \* Write a new value to the element.

  \*/

 writeValue(obj: any): void;

  /\*\*

  \* Set the function to be called when the control receives a change event.

  \*/

 registerOnChange(fn: any): void;

  /\*\*

  \* Set the function to be called when the control receives a touch event.

  \*/

 registerOnTouched(fn: any): void;

  /\*\*

  \* This function is called when the control status changes to or from "DISABLED".

  \* Depending on the value, it will enable or disable the appropriate DOM element.

  \* @param isDisabled

  \*/

 setDisabledState?(isDisabled: boolean): void;

 }

**setValue and patchValue:**

The Angular Forms has three main building blocks i.e **FormControl, FormGroup & FormArray**. All these components have methods setValue & patchValue and behave differently.

**SetValue:**

We use the SetValue to update the FormControl , FormGroup or FormArray. When we use it to update the FormGroup or FormArray the SetValue requires that the object must match the structure of the FormGroup or FormArray exactly. Otherwise, it will result in an error.

setValue(value: { [key: string]: any; }, options: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

**PatchValue:**

The PatchValue is used to update only a subset of the elements of the FormGroup or FormArray. It will only update the matching objects and ignores the rest.

patchValue(value: { [key: string]: any; }, options: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

**OnlySelf:**

The Angular checks the validation status of the form, whenever there is a change in value. The validation starts from the control whose value was changed and propagates to the top level FormGroup. This is the default behavior

There may be circumstances, where you do not want angular to check the validity of the entire form, whenever you change the value using the setValue or patchValue. We do that by setting the **onlySelf=true** as the argument. In such cases, the angular only checks the validity of the current control, but does not check any other control and does not propagate the validity checking to the parent form group.

**Example:**

withOnlySelf(){

this.reactiveForm.get("firstname").setValue("",{onlySelf:true});

}

**EmitEvent**:

The Angular forms emit two events. One is ValueChanges & the other one is StatusChanges. The ValueChanges event is emitted whenever the value of the form is changed. The StatusChanges event is emitted whenever angular calculates the validation status of the Form. This is the default behavior

We can stop that from happening, by setting the **emitEvent=false**

**Example:**

withEmitEvent(){

this.reactiveForm.get("firstname").setValue("",{emitEvent:false});

}

**SetValue vs PatchValue:**

Setvalue and Patchvalue are methods from the Angular Formgroup. They both set the value of a control in a formgroup. The clear difference is that setvalue cannot exclude some controls while the patchvalue is able to do just that.

So let’s assume we have a formgroup with 2 controls: Name & Age

if we want to **set the value** of one control, this will not work, therefor we have to set the value of both controls:

*formgroup.setValue({name: ‘Mocrosoft’, age: ‘25’});*

It is necessary to mention all the controls inside the method. If this is not done, it will throw an error.

On the other hand **patchvalue** is a lot easier on that part, let’s say we only want to assign the name as a new value:

*formgroup.patchValue({name:’Mocrosoft’});*

See how we exclude age and this will work without throwing any errors.

**Validating Form Elements:**

Now we’ll add some validation to our form to prevent the user from submitting until all fields are valid.

First, we’ll need to import Validators which will allow us to set specific validation rules for FormControls.

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

Now update your createForm() to make field values are required to submit form.

createForm(){

    this.reactiveForm = this.formBuilder.group({

      name: ['', Validators.required],

      email: ['', Validators.required],

      password: ['', Validators.required]

    });

Now we can make these validators to disable our form’s submit button until the form is valid.

  <button type="submit" [disabled]="!reactiveForm.valid">Submit</button>

C:\Users\Pujitha.t\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\85AA2BB5.tmp

**Error Messages:** Manipulate createForm() as per the requirement

createForm(){

    this.reactiveForm = this.formBuilder.group({

      name: ['', Validators.required],

      email: ['', [Validators.required, Validators.email]],

      password: ['', [Validators.required, Validators.minLength(6)]]

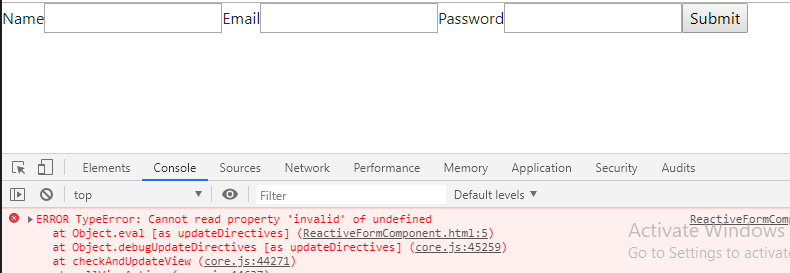
    });

To add more validators to a FormControl we simply add any additional validators to the desired control’s second argument. As you can see, if a FormControl has more than one validator we simply provide it an array of validators instead.

For each input we’ve added a conditional error message. For an input like name, we display the message if the value is invalid and its control isn’t pristine meaning its value has modified.

*You can see a full list of FormControl properties such as pristine and their descriptions* <https://angular.io/api/forms/AbstractControl#properties>.

At this point, open up your browser, open the developer console, and you will see an error like “ERROR TypeError: Cannot read property ‘invalid’ of undefined”.



This error is due to our FormControls are(name, email and password) not being accessible through our template. To fix this, we need to create getter methods in our component to get the specific controls within our form.

get name() { return this.reactiveForm.get('name'); }

get email() { return this.reactiveForm.get('email'); }

get password() { return this.reactiveForm.get('password'); }



**Submitting and Resetting forms:**

**Submit:** Now we can wire up our form’s ngSubmit event to a component function onSubmit.

<form [formGroup]="reactiveForm" (ngSubmit)="onSubmit()">

Then create our onSubmit function in respective component.

  onSubmit() {

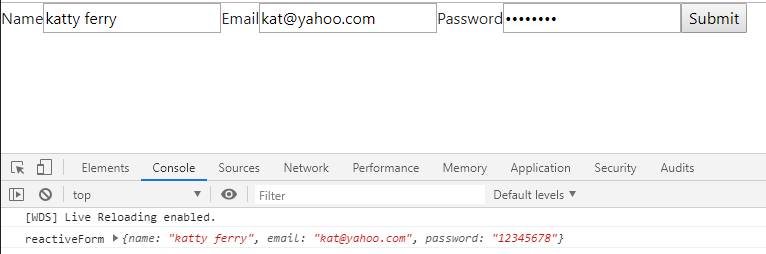
    console.log('reactiveForm' , this.reactiveForm.value);

  }

Now when we fill out the form and submit, we can see your form’s values logged to the console.

**Reset:**

In a model driven form to *reset* the form we just need to call the function *reset()* on our *reactiveForm* model.



  onSubmit() {

    console.log('reactiveForm' , this.reactiveForm.value);

    this.reactiveForm.reset();

  }

The form now resets, all the input fields go back to their initial state and any *valid, touched* or *dirty* properties are also reset to their starting values.

**Using FormBuilder:**

Angular has a helper Class called FormBuilder. FormBuilder allows us to explicitly declare forms in our components. This allows us to also explicitly list each form control’s validators.

We can see how we have used FormBuilder in our form creation.

  createForm(){

    this.reactiveForm = this.formBuilder.group({

      name: ['', Validators.required],

      email: ['', [Validators.required, Validators.email]],

      password: ['', [Validators.required, Validators.minLength(6)]]

    });

  }

**Working with Pipes**

**Intro:** A pipe takes in data as input and transforms it to a desired output. Initially we shall know how to use pipes in Angular.

app.component.ts

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

@Component({

  selector: 'app-root',

  template: "<p>The hero's birthday is {{ birthday | date }}</p>",

  styleUrls: ['./app.component.css']

})

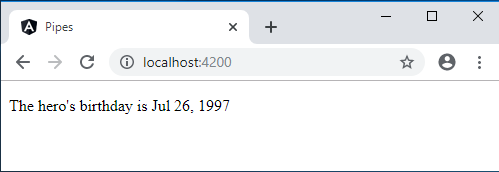
export class AppComponent {

  title = 'pipes';

  birthday = new Date(1997 , 6, 26); // July 26, 1997

}

Output:



**Built-in Pipes:**

Angular 7 provides the following set of built-in pipes.

* AsyncPipe
* DecimalPipe
* DeprecatedDecimalPipe
* I18nSelectPipe
* LowerCasePipe
* TitleCasePipe
* CurrencyPipe
* DeprecatedCurrencyPipe
* DeprecatedPercentPipe
* JsonPipe
* PercentPipe
* UpperCasePipe
* DatePipe
* DeprecatedDatePipe
* I18nPluralPipe
* KeyValuePipe
* SlicePipe

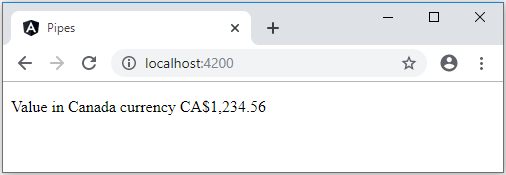
**CurrencyPipe:**

This pipe is used for formatting currencies. Its first argument is an abbreviation of the currency type (e.g. "EUR", "USD", and so on).

*app.component.html:*

<p>Value in Canada currency {{ 1234.56 | currency:'CAD' }}</p>

*Output:*



If instead of the abbreviation of CA$ we want the currency symbol to be printed out we pass as a second parameter the string as shown below

*app.component.html:*

<div class="card card-block">

  <h4 class="card-title">Currency</h4>

  <div class="card-text">

    <p ngNonBindable>{{ 1234.56 | currency:'CAD' }}</p>

    <p>{{ 1234.56 | currency:"CAD" }}</p>

    <p ngNonBindable>{{ 1234.56 | currency:'CAD':'code' }}</p>

    <p>{{ 1234.56 | currency:'CAD':'code'}}</p>

    <p ngNonBindable>{{ 1234.56 | currency:'CAD':'symbol' }}</p>

    <p>{{ 1234.56 | currency:'CAD':'symbol'}}</p>

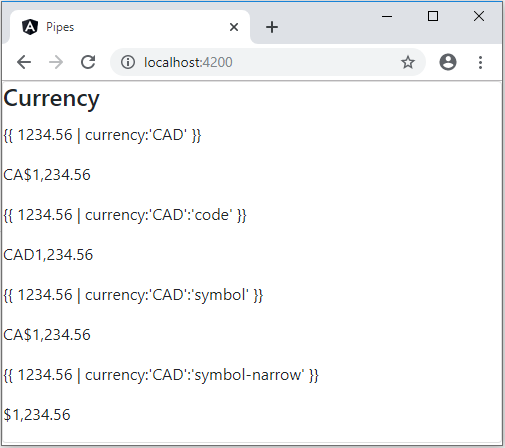
    <p ngNonBindable>{{ 1234.56 | currency:'CAD':'symbol-narrow' }}</p>

    <p>{{ 1234.56 | currency:'CAD':'symbol-narrow'}}</p>

  </div>

</div>

Output:



**DatePipe:** This pipe is used for the transformation of dates. The first argument is a format string.

app.component.ts:

dateVal = new Date();

*app.component.html:*

<div class="card card-block">

<h4 class="card-title">Date</h4>

  <div class="card-text">

    <p ngNonBindable>{{ dateVal | date:'shortTime' }}</p>

    <p>{{ dateVal | date: 'shortTime' }}</p>

    <p ngNonBindable>{{ dateVal | date:'fullDate' }}</p>

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

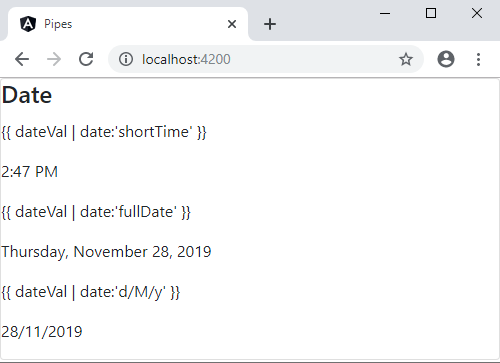
    <p ngNonBindable>{{ dateVal | date:'d/M/y' }}</p>

    <p>{{ dateVal | date: 'd/M/y' }}</p>

  </div>

</div>

*Output:*



**DecimalPipe:** This pipe is used for transformation of decimal numbers.

The first argument is a format string of the form "{minIntegerDigits}. {minFractionDigits}-{maxFractionDigits}".

*app.component.html:*

<div class="card card-block">

    <div class="card-text">

      <h4 class="card-title">DecimalPipe</h4>

      <p ngNonBindable>{{ 3.14159265 | number: '3.1-2' }}</p>

      <p>{{ 3.14159265 | number: '3.1-2' }}</p>

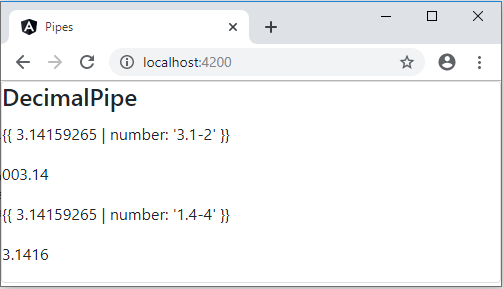
      <p ngNonBindable>{{ 3.14159265 | number: '1.4-4' }}</p>

      <p>{{ 3.14159265 | number: '1.4-4' }}</p>

    </div>

  </div>

*Output:*



**JsonPipe:** This transforms a JavaScript object into a JSON string, like so:

*app.component.ts:*

jsonVal = { moo: 'foo', goo: { too: 'new' }};

*app.component.html:*

<div class="card card-block">

      <h4 class="card-title">JsonPipe</h4>

      <div class="card-text">

        <p ngNonBindable>{{ jsonVal }}</p> (1)

        <p>{{ jsonVal }}</p>

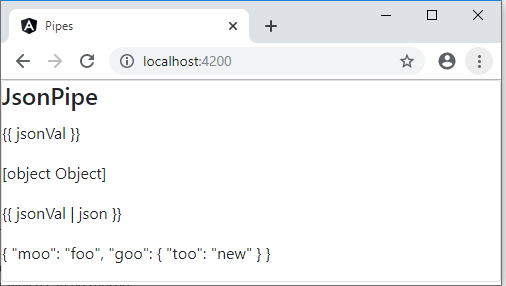
        <p ngNonBindable>{{ jsonVal | json }}</p>

        <p>{{ jsonVal | json }}</p>

      </div>

    </div>

*Output:*



**LowerCasePipe & UpperCasePipe:**

LowerCasePipe transforms a string to lowercase and UpperCasePipe transforms a string to uppercase.

*app.component.html:*

<div class="card card-block">

        <h4 class="card-title">LowerCasePipe</h4>

        <div class="card-text">

          <p ngNonBindable>{{ 'ASIM' | lowercase }}</p>

          <p>{{ 'ASIM' | lowercase }}</p>

        </div>

        <div class="card-text">

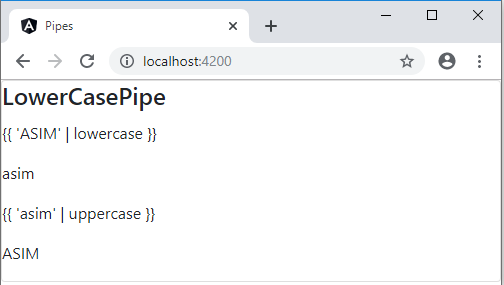
          <p ngNonBindable>{{ 'asim' | uppercase }}</p>

          <p>{{ 'asim' | uppercase }}</p>

        </div>

</div>

*Output:*



**PercentPipe:** Formats a number as a percent

*app.component.html:*

<div class="card card-block">

<h4 class="card-title">PercentPipe</h4>

     <div class="card-text">

  <p ngNonBindable>{{ 0.123456 | percent }}</p>

  <p>{{ 0.123456 | percent }}</p>

       <p ngNonBindable>{{ 0.123456 | percent: '2.1-2' }}</p>

       <p>{{ 0.123456 | percent: '2.1-2' }}</p>

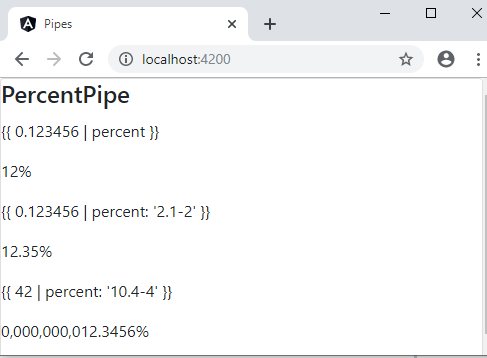
       <p ngNonBindable>{{ 42 | percent: '10.4-4' }}</p>

       <p>{{ 0.123456 | percent : "10.4-4" }}</p>

     </div>

</div>

*Output:*



**SlicePipe:** This returns a *slice* of an array. The first argument is the start index of the slice and the second argument is the end index.

If either indexes are not provided it assumes the start or the end of the array and we can use negative indexes to indicate an offset from the end

*app.component.html:*

<div class="card card-block">

          <h4 class="card-title">SlicePipe</h4>

          <div class="card-text">

            <p ngNonBindable>{{ [1,2,3,4,5,6] | slice:1:3 }}</p>

            <p>{{ [1,2,3,4,5,6] | slice:1:3 }}</p>

            <p ngNonBindable>{{ [1,2,3,4,5,6] | slice:2 }}</p>

            <p>{{ [1,2,3,4,5,6] | slice:2 }}</p>

            <p ngNonBindable>{{ [1,2,3,4,5,6] | slice:2:-1 }}</p>

            <p>{{ [1,2,3,4,5,6] | slice:2:-1 }}</p>

            <pre ngNonBindable>

        &lt;ul&gt;

          &lt;li \*ngFor=&quot;let v of [1,2,3,4,5,6] | slice:2:-1&quot;&gt;

            {{v}}

          &lt;/li&gt;

        &lt;/ul&gt;

            </pre>

            <ul>

              <li \*ngFor="let v of [1,2,3,4,5,6] | slice:2:-1">

                {{v}}

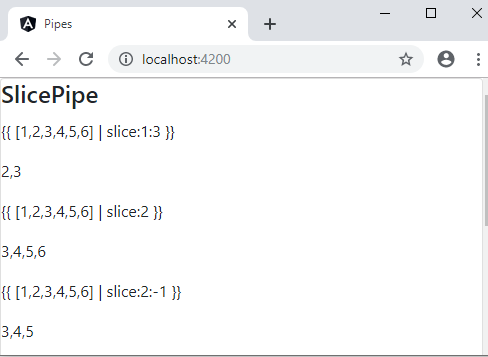
              </li>

            </ul>

          </div>

        </div>

*Output:*



slice:1:3 means return the items from the 1st to the 3rd index inclusive (indexes start at 0).

slice:2 means return the items from the 2nd index to the end of the array.

|  |
| --- |
| slice:2:-1 means return the items from the 2nd index to one from the end of the array. |

We can use slice inside for loops to only loop over a subset of the array items.

**AsyncPipe:**

The Async pipe is considered the best practice when you are getting data in the form of observables. The async pipe subscribes to an Observable/Promise automatically and returns the transmitted values.

The async pipe subscribes to an Observable or Promise and returns the latest value it has emitted. When a new value is emitted, 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.

Normally to render the result of a promise or an observable we have to:

1. Wait for a *callback*.
2. Store the result of the callback is a *variable*.
3. *Bind* to that variable in the template.

With AsyncPipe we can use promises and observables directly in our template, without having to store the result on an intermediate property or variable.

AsyncPipe accepts as argument an observable or a promise, calls subcribe or attaches a then handler, then waits for the asynchronous result before passing it through to the caller.

**AsyncPipe with Promises**:

app.component.ts:

constructor() {

      this.getPromise().then(result => this.promiseData = result);

  }

  getPromise(){

    return new Promise((resolve,reject)=>{

      setTimeout(() => resolve('Promise Resolved'),2000)

    });

  }

*app.component.html:*

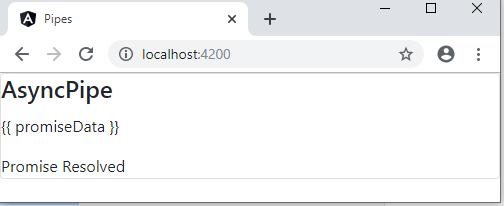
<div class="card card-block">

          <h4 class="card-title">AsyncPipe</h4>

          <p class="card-text" ngNonBindable>{{ promiseData }}</p>

          <p class="card-text">{{ promiseData }}</p></div>

*Output:*



|  |
| --- |
| We use ngNonBindable so we can render out {{ promiseData }} as is without trying to bind to to the property promiseData |

getPromise returns a promise which 2 seconds later resolves with the value "Promise Resolved!"

In the constructor we wait for the promise to resolve and store the result on a property called promiseData on our component and then bind to that property in the template.

To save time we can use the async pipe in the template and bind to the promise directly, as shown below

*app.compontnt.ts:*

   constructor() {

    this.promiseData= this.getPromise();

  }

  getPromise(){

    return new Promise((resolve,reject)=>{

      setTimeout(() => resolve("Promise Resolved"),2000)

    });

  }

*app.componet.html:*

<div class="card card-block">

<h4 class="card-title">AsyncPipe</h4>

     <p class="card-text" ngNonBindable>{{ promiseData }}</p>

     <p class="card-text">{{ promiseData | async }}</p>

</div>

We pipe the output of our promise to the async pipe.

The property promise is the actual unresolved promise that gets returned from get Promise without then being called on it. The above results in the same behavior as before, we just saved ourselves from writing a then callback and storing intermediate data on the component.

**AsyncPipe with observables:**

The example below binds the time Observable to the view. The Observable continuously updates the view with the current time.

*app.component.ts:*

import { Observer, Observable } from 'rxjs';

@Component({

  selector: 'app-root',

  template: "<div><code>observable|async</code>: Time: {{ time | async }}</div>",

})

export class AppComponent {

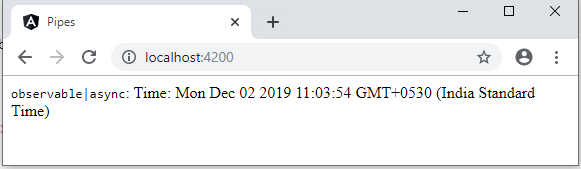
  time = new Observable<string>((observer: Observer<string>) => {

    setInterval(() => observer.next(new Date().toString()), 1000);

  });

}

*Output:*



**Using parameters and chaining Pipes:**

A pipe can accept any number of optional parameters to fine-tune its output. To add parameters to a pipe, follow the pipe name with a colon ( : ) and then the parameter value (such as currency:'EUR'). If the pipe accepts multiple parameters, separate the values with colons (such as slice:1:5)

In the above examples itself we had covered parameterized pipes. Modify the birthday template to give the date pipe a format parameter. After formatting July 26, 1997, it renders as 07/26/97:

*app.component.ts:*

@Component({

  selector: 'app-root',

  template: "<p> Birthday is {{ birthday | date:'MM/dd/yy' }} </p>",

})

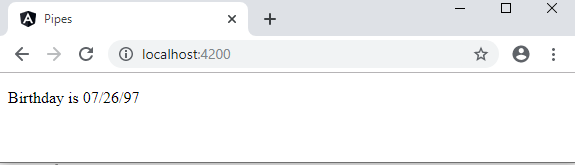
export class AppComponent {

  title = 'pipes';

  birthday = new Date(1997 , 6, 26); // July 26, 199

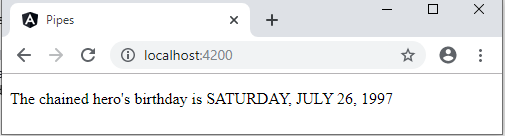
}

*Output:*



You can chain pipes together in potentially useful combinations. In the following example, to display the birthday in uppercase, the birthday is chained to the DatePipe and on to the UpperCasePipe.

<p> The chained hero's birthday is  {{  birthday | date:'fullDate' | uppercase}} </p>



**Custom Pipes:**

We can generate custom pipes file with the help of Angular CLI.

ng generate pipe *pipe-name*

You can write your own custom pipes. Here's a custom pipe named ExponentialStrengthPipe.

*exponential-strength.pipe.ts:*

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

@Pipe({

  name: 'exponentialStrength'

})

export class ExponentialStrengthPipe implements PipeTransform {

  transform(value: number): number {

    return Math.pow(value, 10);}}

To tell Angular that this is a pipe, you apply the @Pipe decorator, which you import from the core Angular library.

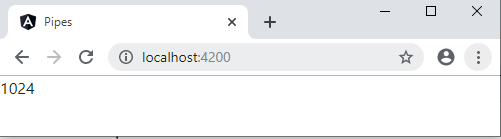
The @Pipe decorator allows you to define the pipe name that you'll use within template expressions. It must be a valid JavaScript identifier. Your pipe's name is exponentialStrength.

The pipe class implements the PipeTransform interface's transform method that accepts an input value followed by optional parameters and returns the transformed value.

*app.component.html:*

<p>{{ 2 | exponentialStrength }}</p>

*Output:*



**Parameterized Custom Pipe:**

There will be one additional argument to the transform method for each parameter passed to the pipe. Your pipe has one such parameter: the exponent.

*exponential-strength.pipe.ts:*

 transform(value: number, exponent?: number): number {

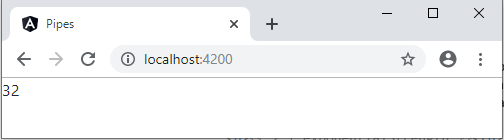
    return Math.pow(value, isNaN(exponent) ? 1 : exponent);

  }

*app.component.html:*

<p>{{ 2 | exponentialStrength:5 }}</p>

*Output:*



**Pure and Impure Pipes:**

When writing a custom pipe in Angular you can specify whether you define a pure or an impure pipe.

There are two categories of pipes: *pure* and *impure*. Pipes are pure by default.

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.

**Custom Directives**

**Custom Attribute Directive:**

We can create custom attribute directives and custom structural directives using a @Directive decorator. Using ‘custom attribute directive’, we can change appearances such as text color, background color and font size of a body of an HTML element that can be called the host element. To change appearance Angular provides ElementRef class that can directly access DOM.

**Steps to create a custom attribute directive:**

1. Create a class decorated with @Directive.
2. Assign the attribute directive name to the selector metadata of @Directive decorator.
3. Use ElementRef class to access DOM to change host element appearance and behavior.
4. Use @Input() decorator to accept user input in our custom directive.
5. Use @HostListener() decorator to listen to events in custom attribute directive.
6. Configure custom attribute directive class in the application module in the declarations metadata of @NgModule.

For Example, Lets create a custom attribute directive which changes colors dynamically.

**Using HostListener:** @HostListener decorator is a function that accepts an event name as an argument. When that event gets fired on the host element it calls the associated function.

The below custom attribute directive that will perform changes in appearance in DOM when an event is fired. To listen to any event we need to use @HostListener(), decorator. We need to assign event name to @HostListener() decorator. Here we will create an attribute directive that will change the background color of host element when mouseover and mouseleave events are fired.

import {Directive, ElementRef, HostListener, Input} from '@angular/core';

@Directive({

     selector: '[dynamicColor]'

})

export class DynamicColorDirective {

   @Input('dynamicColor') 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;

   }

}

**Using HostBinding:** For listening to output events from the host element a directive can also bind to input properties in the host element with @HostBinding.

This directive can change the properties of the host element, such as the list of classes that are set on the host element as well as a number of other properties.

Using the @HostBinding decorator a directive can link an internal property to an input property on the host element. So, if the internal property changed the input property on the host element would also change.

For example,

import { Directive, ElementRef, HostListener, Input, Output, HostBinding } from '@angular/core';

@Directive({

     selector: '[dynamicColor]'

})

export class DynamicColorDirective {

   @Input('dynamicColor') dynamicColor: string;

   @Input() defaultValue: string;

   @HostBinding('class.addBorder') private ishovering: boolean;

   constructor(private elRef: ElementRef) {

   }

   @HostListener('mouseover') onMouseOver() {

     this.changeBackgroundColor(this.dynamicColor || this.defaultValue);

     this.ishovering = true;

   }

  @HostListener('click') onClick() {

        window.alert('Host Element Clicked');

        console.log("hii");

    }

   @HostListener('mouseleave') onMouseLeave() {

     this.changeBackgroundColor('white');

     this.ishovering = false;

   }

   private changeBackgroundColor(color: string) {

     this.elRef.nativeElement.style.backgroundColor = color;

   }

}

Here @HostBinding decorator is add to ishovering property as shown above.

By using the @HostListener and @HostBinding decorators we can both listen to output events from our host element and also bind to input properties on our host element as well.

**Custom Validator Directive:**

Angular supports a few very useful native validators:

1. **required**: validate if the field is mandatory.
2. **minlength**: validate the minimum length of the field.
3. **maxlength**: validate the maximum length of the field.
4. **pattern**: validate if the input value meets the defined pattern, e.g. email.

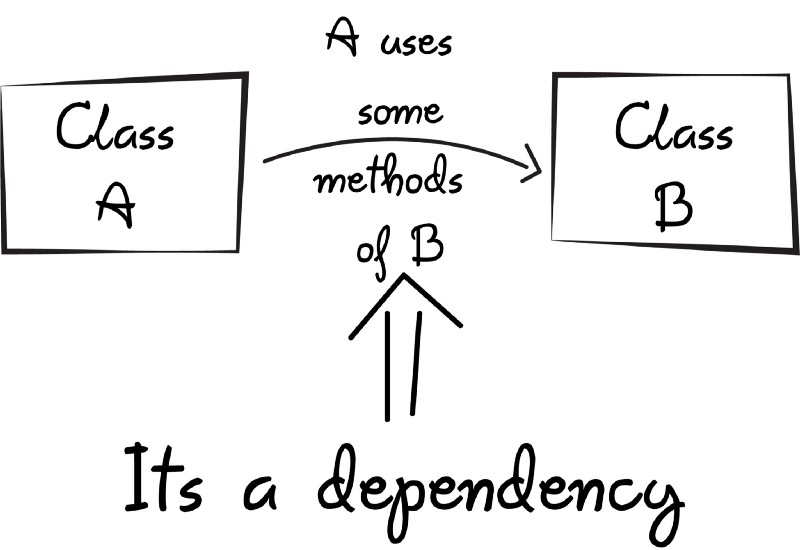
A form was implemented which shows about custom validator directive.

## Dependency Injection

**Intro:**

Dependency Injection (DI) is a design pattern that takes away the responsibility of creating dependencies from a class, which resulting in a loosely coupled system.

**What is Dependency?**



Let us discuss in terms of coding language as shown below. The Customer class declares a variable of type DatabaseHelper. The DatabaseHelper class is supposed to be doing some database operations such as SELECT and UPDATE. The constructor of the Customer class instantiates the DatabaseHelper class and the instance is stored in the helper variable. In this case the Customer class is dependent on the DatabaseHelper class for its functioning. Here, DatabaseHelper is a ‘dependency’ of the Customer class.

public class Customer

{

private DatabaseHelper helper;

public Customer()

{

helper = new DatabaseHelper();

}

}

The problem with the above design is that the Customer class and DatabaseHelper class are “Tightly coupled”. If we need to substitute DatabaseHelper with some other class (say XMLHelper) then we will have to change the code of the Customer class since it directly instantiates the DatabaseHelper. To avoid this, tight coupling **Dependency Inversion Principle (DIP)** is used.

**Dependency Inversion Principle:** High-level modules should not depend on low-level modules. Both should depend on abstractions.

i.e. That means Customer should not depend on a DatabaseHelper implementation but rather should depend on an abstraction. At code level this means the Customer class won't have a variable of DatabaseHelper type, instead it will have a variable of some interface (or abstract class) type. Which is shown below.

public class Customer

{

private IStorageHelper helper;

public Customer()

{

helper = new DatabaseHelper();

}

}

IStorageHelper is supposed to be an interface that is implemented by the DatabaseHelper class and all such classes. Thus, most of the code of the Customer class is now using an abstraction in the form of IStorageHelper.

Although the above code is better than the original implementation, it still has a problem. The helper variable is still instantiated inside the Customer class. This problem arises because the Customer class is responsible for creating its dependency i.e. DatabaseHelper. **The Inversion of Control principle (IoC)** comes to avoid such cases.

**Inversion of Control:**  It is the fifth principle of **S.O.L.I.D** (Thefive basic principles of object-oriented programming).

It states that the control of creating the decencies should be with the external system rather than the class itself. In the above example this means that the Customer class shouldn't create an instance of DatabaseHelper, rather it should be received from the external system.

Therefore, Dependency Injection is a way to implement IoC such that the dependencies are "injected" into a class from some external source. The injected dependencies can either be received as constructor parameters of a class or can be assigned to properties of that class designed for that purpose.

public class Customer

{

private IStorageHelper helper;

public Customer(IStorageHelper helper)

{

this.helper = helper;

}

}

#### There are basically three types of dependency injection:

1. **Constructor Injection:** The dependencies are provided through a class constructor.
2. **Setter Injection:** The client exposes a setter method that the injector uses to inject the dependency.
3. **Interface Injection:** The dependency provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a [setter method](https://en.wikipedia.org/wiki/Setter_method) that accepts the dependency.

**Dependency Injection in Angular Framework:**

Angular has its own dependency injection framework. The DI framework in Angular consists of 4 concepts working together:

**Token:** This uniquely identifies something that we want injected. A dependency of our code.

**Dependency:** The actual code we want injected.

**Provider:** This is a map between a token and a list of dependencies.

**Injector:** @Inject decorator is used to explicitly tell Angular what we want injected for the other Service parameter.

For ex:

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

class SimpleService {

  otherService: OtherService;

  constructor(@Inject (OtherService) otherService: OtherService) {

      this.otherService = otherService;

  };

}

The first param to @Inject is the token we want to resolve this dependency with. The above now works, when Angular tries to construct the class it gets the instance of Other Service passed in from the DI framework.

**@Injectable Decorator:** It is actually a shortcut for having to decorate every parameter in your constructor with @Inject. The other decorators in Angular, such as @Component, @Directive, already perform the same function as @Injectable.

For Example,

Consider a Service names as popcorn as shown below.

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

@Injectable()

export class PopcornService {

  constructor() {

    console.log("Popcorn has been injected!");

    alert("Service has been injected");

  }

  cookPopcorn(qty) {

    console.log(qty, "bags of popcorn cooked!");

    alert("Service method has called");

  }

}

Now import popcorn service in app.module.ts as follows:

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

import { PopcornService } from './popcorn.service';

@Component({

  selector: 'my-app',

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

  styleUrls: ['./app.component.css'],

  providers: [PopcornService]

})

export class AppComponent {

  constructor(private popcorn: PopcornService) {}

  cookIt(qty) {

    this.popcorn.cookPopcorn(qty);

  }

}

The cookIt() method in the template calls the cookPopcorn() method in the injected service. Let’s make use of our cookIt() method in our template i.e. app.component.html

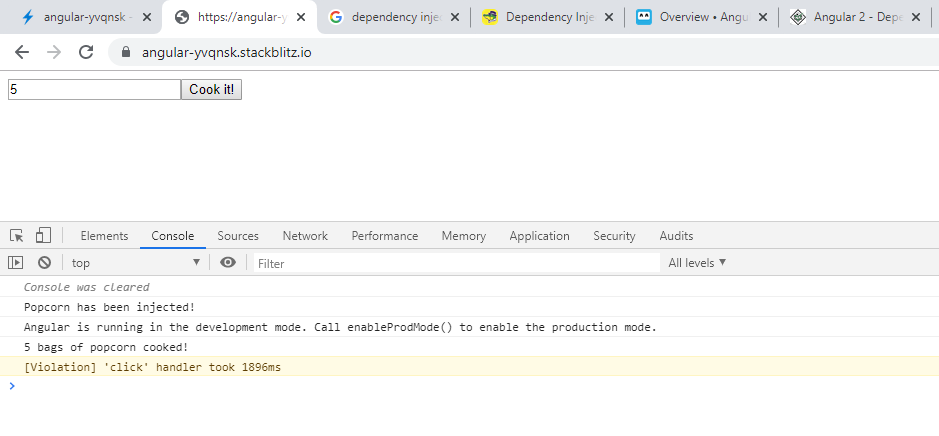
<input type="number" #qty placeholder="How many bags?">

<button type="button" (click)="cookIt(qty.value)">

  Cook it!

</button>

Then final output will be like as follows,



**Reflective Injector**:

The **Reflective Injector** class creates and stores the instances of classes, and also retrieve an instance using the reflection capabilities of Reflect metadata. It uses the class name as the key to retrieve the class instance.

For ex,

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

class HttpService {

    constructor() {}

    get(url: string) {

        return ['rice', 'beans']

    }

}

@Injectable()

class FoodService {

    constructor(private http: HttpService) {}

    getFoods() {

        return this.http.get('foods')

    }

}

const injector = ReflectiveInjector.resolveAndCreate([

    HttpService,

    FoodService

]);

const foodService = injector.get(FoodService);

console.log(foodService.getFoods())

//OUTPUT : ['rice', 'beans']

We import our injector ReflectiveInjector from @angular/core. We construct two classes of Http Service and Food Service. Http Service returns an array of food in it get method, Food Service takes an instance of Http Service in its constructor, it uses the instance in it get Foods methods to get the array of food from Http Service. Next, we configure our injector by calling the static method resolveAndCreate in ReflectiveInjector. It takes an array of classes as parameters, these classes are stored in a key-value store with their class name as key and their instances as the value. That is the reason we have to supply the class name to get its instance.

**Provider:**

Providers and dependency injection are one of the most critical and unique features of Angular. A ‘**provider’** is an object declared to Angular so that it can be injected in the constructor of your components, directives and other classes instantiated by Angular. Whereas ‘**service’** is a particular type of provider that is declared with its class name. It can be declared either in your NgModule or in a specific component simply by adding providers: [MyService] to your configuration metadata.

There are many ways to declare providers by value, alias, factory, etc. Providers declared for a component [are only injectable within that component](https://angular.io/guide/hierarchical-dependency-injection), and they override any provider with the same “name” that were declared by parent components or by the module itself.

**Services in Angular**

**Intro:**

Angular services are singleton objects which get instantiated only once during the lifetime of an application.

Services allow you to define code that's accessible and reusable throughout multiple components. A common use case for services is when you need to communicate with a backend of some sort to send and receive data

The separation of concerns is the main reason why Angular services came into existence. An Angular service is a stateless object and provides some very useful functions. These functions can be invoked from any component of Angular, like Controllers, Directives, etc. This helps in dividing the web application into small, different logical units which can be reused.

separation of concerns (SoC) is a design principle for separating a computer program into distinct sections such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program.

**Building and Injecting Custom Services:** Command to create service

***ng generate service service-name***

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

@Injectable({

  providedIn: 'root'

})

export class DataService {

  constructor() { }

  clicked() {

    return console.log('clicked');

  }

}

Create a function in service.

There are three ways to register a service

1. First way is to use @Injectable decorator and specify provided In property as above.

‘root’ means that we want to provide the service at the root level (AppModule).

When you provide the service at the root level, Angular creates a single, shared instance of service and injects into any class that asks for it.

1. Registering the Service in providers of preferred Module. Like.,

import { DataService } from './\_services/data.service';

@NgModule({

  declarations: [

    AppComponent,

    ServicesExampleComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

  ],

  providers: [

    DataService

  ],

  bootstrap: [AppComponent]

})

export class AppModule { }

1. Registering the service at component itself. Like.,

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

import {DataService} from '../\_services/data.service'

@Component({

  selector: 'app-services-example',

  template:

    '<button class="btn btn-primary" (click)="clicked()">Click me</button>',

  styleUrls: ['./services-example.component.css'],

  providers: [DataService]

})

export class ServicesExampleComponent implements OnInit {

  constructor(private data: DataService) { }

  ngOnInit() {

  }

  clicked(){

    this.data.clicked();

  }

}

Now, create a component that makes use of above service

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

import {DataService} from '../\_services/data.service'

@Component({

  selector: 'app-services-example',

  template:

    '<button class="btn btn-primary" (click)="clicked()">Click me</button>',

  styleUrls: ['./services-example.component.css']

})

export class ServicesExampleComponent implements OnInit {

  constructor(private data: DataService) { }

  ngOnInit() {

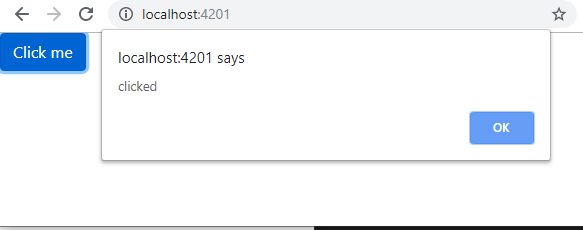
  }

  clicked(){

    this.data.clicked();

  }

}



**Service using another Service:**

Let’s say that you have a MathService and CalculatorService in an regular Angular application.

**math.service.ts**

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

@Injectable()

export class MathService {

  multiply(a: number, b: number): number {

    return a \* b;

  }

}

**calculator.service.ts**

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

import { MathService } from './math.service';

@Injectable()

export class CalculatorService {

  constructor(

    private mathService: MathService

  ) { }

  square(a: number): number {

    return this.mathService.multiply(a, a);

  }

  cube(a: number): number {

    return this.mathService.multiply(a, this.mathService.multiply(a, a));

  }

  multiply(a: number, b: number): number{

    return this.mathService.multiply(a, b);

  }

}

By seeing above code, we can say Calculator Service depends on Math Service. Make a call to Calculator Service from a component.

**services-example.component.ts**

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

import { CalculatorService } from '../\_services/calculator.service';

@Component({

  selector: 'app-services-example',

  templateUrl:

    'services-example.component.html',

  styleUrls: ['./services-example.component.css'],

  providers: [CalculatorService]

})

export class ServicesExampleComponent implements OnInit {

  result1: number;

  result2: number;

  result3: number;

  constructor(private calculatorService: CalculatorService) { }

  ngOnInit() {

    this.result1 = this.calculatorService.square(2);

    this.result2 = this.calculatorService.multiply(2,3);

    this.result3 = this.calculatorService.cube(3);

  }

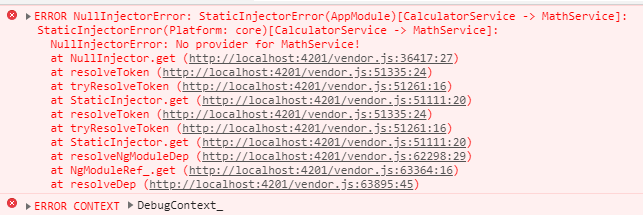
}

**services-example.component.html**

<h1>Result1: {{result1}}</h1>

<h2>Result2: {{result2}}</h2>

<h3>Result3: {{result3}}</h3>

Now if we run our application, we'll see something like this:

So, whats happening here?

1. Angular is instantiating *CalculatorService* because we injected it into *ServicesExampleComponent* and declared it as a provider.
2. To complete this task, Angular will check the dependencies of *CalculatorService* and, in this case, it is *MathService.* But, how Angular would know that *MathService* is a provider to *CalculatorService*? There is no providers field in @Injectable

We must tell Angular to instantiate *MathService* to be available (instatiated) before *CalculatorService*.

Hence, we declare it as a provider into our module. In this small example application, we have only one module, the app.module.ts.

**app.module.ts**

import { ServicesExampleComponent } from './services-example/services-example.component';

import { MathService } from './\_services/math.service';

@NgModule({

  declarations: [

    AppComponent,

    ServicesExampleComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule

  ],

  providers: [

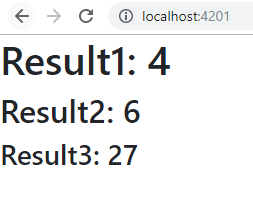
    MathService

  ],

  bootstrap: [AppComponent]

})

export class AppModule { }

**Output:**

**Reactive Extension for JavaScript**

**Introduction:** *RxJS* stands for \*R\*eactive E\*x\*tensions for \*J\*ava\*S\*cript.

*RxJS* is a framework for reactive programming that makes use of Observables, making it really easy to write asynchronous code.

This project is a kind of reactive extension to JavaScript with better performance, better modularity, better debuggable call stacks, while staying mostly backwards compatible, with some breaking changes that reduce the API surface. It is the official library used by Angular to handle reactivity, converting pull operations for call-backs into Observables.

RxJS is the JavaScript implementation of the ReactiveX API

The API has multiple implementations in differentlanguages, so if you learn RxJS you’ll know how to write RxJAVA, Rx.NET, RxPY etc…​

**Observable and Observer:**

In ReactiveX an observer subscribes to an Observable. Observable follows the observer design pattern. Then that observer reacts to whatever item or sequence of items the Observable emits. This pattern facilitates concurrent operations because it does not need to block while waiting for the Observable to emit objects, but instead, it creates a sentry in the form of an observer that stands ready to react appropriately at whatever future time the Observable does so.

In other documents and other contexts, what we are calling an “observer” is sometimes called a “subscriber,” “watcher,” or “reactor.” This model, in general, is often referred to as the "reactor pattern".

Javascript promises still work well with Angular and Observable is an alternative and Angular embraces it. What makes Observable better over promises.

1. Using observer operators are the major advantage of using Observable.
2. Most operators operate on an Observable and return an Observable. This allows you to apply these operators one after the other, in a chain. Each operator in the chain modifies the Observable that results from the operation of the previous operator.
3. Catch the failure and throw the user friendly error message.
4. Cancel the request.
5. Interceptors returns the Observable(both request and response).
6. Define the response type(text/JSON). By default,it returns the JSON response.

Create a Student Service file(ng g s student --spec=false).

We have created a service because we will use the service to handle the data that needs to be displayed on the frontend.

Also, create a new file*student.model.ts* .

*student.model.ts*

export class Student {

    id: Number;

    name: String;

    enrollmentnumber: Number;

    college: String;

    university: String;

}

Now, we need to add the demo data inside the *student.service.ts*file. The data is the type of Student model which we have defined above.

*student.service.ts*

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

import { Student } from './student.model';

import { Observable } from 'rxjs';

@Injectable({

  providedIn: 'root'

})

export class StudentService {

  students: Student[] = [{

    id: 1,

    name: 'Krunal',

    enrollmentnumber: 110470116021,

    college: 'VVP Engineering College',

    university: 'GTU'

},

{

    id: 2,

    name: 'Rushabh',

    enrollmentnumber: 110470116023,

    college: 'VVP Engineering College',

    university: 'GTU'

},

{

    id: 3,

    name: 'Ankit',

    enrollmentnumber: 110470116022,

    college: 'VVP Engineering College',

    university: 'GTU'

}];

  constructor() { }

}

Now, we have defined the private data. We need to create one function inside the service that will return that data in the form of observable. So we can subscribe to it, and we get the data and display it on the frontend.

Add to *student.service.ts*

public getStudents(): any {

  const studentsObservable = new Observable(observer => {

         setTimeout(() => {

             observer.next(this.students);

         }, 1000);

  });

  return studentsObservable;

}

So, here we have done is first import the Observable from rxjs. It has then defined one function that will return an observable. The observable object gets one argument that has a timeout function. So after 1 second, it will produce the whole student’s array if the subscriber subscribes the observable.

In simple terms, here studentObservable are publishing our primary data array that is students. So if any entity needs to get the values out of observable, then it first needs to subscribe that observable and then studentObservable starts to publish the values, and then subscriber get the values.

We have created the Publisher for the Observables. Now, we need to create a subscriber. So write the following code inside the *app.component.ts*file.

*app.component.ts*

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

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

import { Student } from './student.model';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

  students: Student[];

  constructor( private studentservice: StudentService) {

  }

  ngOnInit(){

    const studentsObservable = this.studentservice.getStudents();

        studentsObservable.subscribe((studentsData: Student[]) => {

            this.students = studentsData;

        });

  }

}

Here, we have subscribed the observable and get the students data.

*app.component.html*

<div class="container">

  <div class="row" style="margin-top: 30px">

      <div class="col-md-3 col-xs-6" \*ngFor="let student of students">

          <div class="card">

              <div class="card-body">

                  <h5 class="card-title">{{ student.name }}</h5>

                  <h6 class="card-subtitle">{{ student.enrollmentnumber }}</h6>

                  <p class="card-text">{{ student.college }}</p>

                  <p class="card-text">{{ student.university }}</p>

                  <a class="btn btn-primary" href="#" >Go somewhere</a>

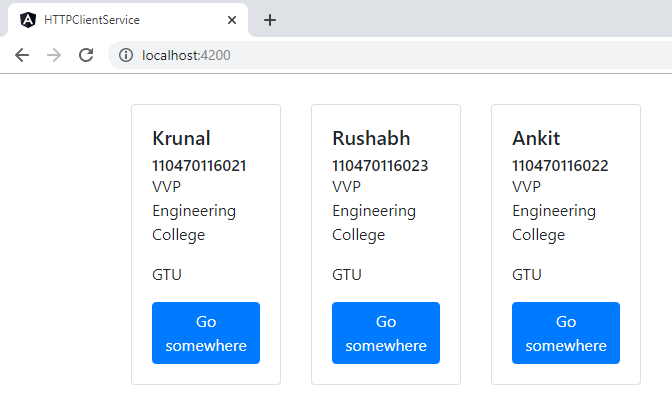
              </div>

          </div>

      </div>

  </div>

</div>

*Output:*

Remember, Observables producing the values Asynchronously. Observables are the lazy collections of multiple values or streams over time. It is like a subscription to the newsletter, if you keep that subscription open, you will get the new one every once and a while.

**Reactive Operators:**

RxJS provides the implementation of Observable concept but also a variety of operators that allows you to compose Observables.

**Http Client Service**

**HttpClientModule and HttpClient Classes:** HttpClient class provides easiest way to handle HTTP requests.

HttpClientModule will have HttpClient Class which is ready to use for sending network requests to server.

So, Before you can use a HttpClient, you need to import an Angular HttpClientModule. Most apps do import in the root AppModule.

*app.module.ts*

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

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

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

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

@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule,

    // import HttpClientModule after BrowserModule.

    HttpClientModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

Now, the Angular HttpClient is ready to use or inject with the Angular service or component.

If we need to send a POST request to the server, then we import the HttpClient inside the service file and make an AJAX request to the server with the data object.

Create a Service in Angular

*config.service.ts:*

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

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

@Injectable()

export class ConfigService {

  constructor(private http: HttpClient) { }

}

**Writing Service with Get / Add / Edit / Delete:**

We have created a service file. Now, we need to write the code that sends a GET request to the server.

*config.service.ts*

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

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

@Injectable()

export class ConfigService {

  constructor(private http: HttpClient) { }

  url = '<http://localhost:4000/api>';

  getCharacters() {

    return this

            .http

            .get(`${this.url}/responses`);

  }

}

And here is the example of a small service that queries the database above using an HTTP GET, and fetch the data.

Let’s break down the following example step-by-step:

1. We are using the new HttpClient client module, and injecting it in the constructor
2. Then we are calling the get() method, which is returning the data.

The HttpClient**.**get() method parsed a server response into an anonymous Object type. It doesn’t know what shape or attributes of that object is.

You can tell the HttpClient the type of response to make consuming the output easier and more prominent.

First, define the interface for the expected Response objects.

*config.service.ts*

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

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

import { Observable } from 'rxjs/internal/Observable';

@Injectable()

export class ConfigService {

  constructor(private http: HttpClient) { }

  url = '<http://localhost:4000/api>';

  getResponses(): Observable<Response[]> {

    return this

            .http

            .get<Response[]>(`${this.url}/responses`);

  }

}

export interface Response {

  id: Number;

  header: String;

  body: Number;

}

Just like in the case of GET request, we can also use the Angular HTTP Client to do all the other available HTTP methods, namely

POST(Add), PUT(Edit) and DELETE(Delete).

**Http Headers:**

Many servers require extra headers for save operations. For example, they may require a "Content-Type" header to explicitly declare the MIME type of the request body; or the server may require an authorization token.

**Adding headers:**

The ConfigService defines such headers in an httpOptions object that will be passed to every [HttpClient](https://angular.io/api/common/http/HttpClient) save method.

*config.service.ts*

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

const httpOptions = {

  headers: new HttpHeaders({

    'Content-Type':  'application/json',

    'Authorization': 'my-auth-token'

  })

};

**Updating headers:**

You can't directly modify the existing headers within the previous options object because instances of the [HttpHeaders](https://angular.io/api/common/http/HttpHeaders) class are immutable.

Use the set() method instead, to return a clone of the current instance with the new changes applied.

Here's how you might update the authorization header (after the old token expired) before making the next request.

httpOptions.headers = httpOptions.headers.set('Authorization', 'my-new-auth-token');

Let's see an example for POST call with Http Headers included

addResponse (response: Response): Observable<Response> {

    return this.http.post<Response>(this.url, response, httpOptions);

  }

**Using Service in Component:**

To use Service in Component you need to add preferred service in Providers and Inject the service in the component that you prefer to use.

*app.component.ts*

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

import { ConfigService, Response } from './config.service';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  responses: Response[];

  newResponse: Response;

  constructor(private configService: ConfigService) {

  }

  addResponse(){

    this.configService

  .addResponse(this.newResponse)

  .subscribe(res => this.responses.push(res));

  }}

When the server responds successfully with the newly added response, the component adds that response to the displayed responses list.

**Angular Routing**

**Introduction:**

To navigate to different pages in your application you will need routing. A Routing component can lead to sub-routes without changing the complete URL (in case of web applications) or completely reloading the whole page, thus, satisfying the application to be a SPA (Single Page Application).

Angular Router is a powerful JavaScript router built and maintained by the Angular core team that can be installed from the @angular/router package. It provides a complete routing library with the possibility to have multiple router outlets, different path matching strategies, easy access to route parameters and route guards to protect components from unauthorized access.

**THE ROUTER-OUTLET:**

The Router-Outlet is a directive that’s available from the router library where the Router inserts the component that gets matched based on the current browser’s URL. You can add multiple outlets in your Angular application which enables you to implement advanced routing scenarios.

<router-outlet></router-outlet>

**Configuring and Navigating:**

Let us understand how to navigate to different web pages based on routes that we configure with an example

Lets create 3 components namely 'component1', 'component2' and 'component3'.

Go to *app-routing.module.ts* and configure routes as following

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

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

import { Component1Component } from './component1';

import { Component2Component } from './component2';

import { Component3Component } from './component3';

const routes: Routes = [

  { path: '', component:  Component1Component},

  { path: 'component2', component: Component2Component},

  { path: 'component3', component: Component3Component}

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule { }

Based on a route definition that we provide (via a static RouterModule.forRoot(routes) method), the Router is able to navigate the user to a specific view.

*component1.component.html*

<h1>component1 works!</h1>

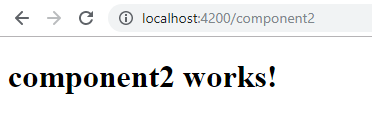
Similarly add for *component2.component.html* and *component3.component.html*

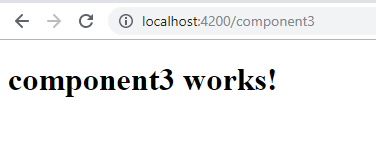
*app.component.html*

<router-outlet></router-outlet>

Now run your application







**Parameterized routes:** When we think of above example there is better way to displaying them than creating one component for each

i.e., with the help of parameterised route we can achieve this

*app.component.html*

<h1>Hi this is App Component</h1>

<router-outlet></router-outlet>

*app-routing.module.ts*

const routes: Routes = [

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

  {path: 'component/:id', component: Component1Component},

  {path: 'home', component: HomeComponent}

];

The path has a variable called id, we know it’s a variable since it begins with a colon :

A path can have any number of variables as long as they all start with : and have different names.

So how do we pass the value of the *id* variable? If we visited */component/1* how does Component know the id is 1 and therefore to show the appropriate data.

To do that we use something called an*ActivatedRoute.*

*ActivatedRoute* provides access to information about a route associated with a component that is loaded in an outlet.

We import it first and then inject it into the constructor of Component1Component. It exposes an Observable which we can subscribe to, like so:

*component1.component.ts*

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

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

@Component({

  selector: 'app-component1',

  template: '<h1>component {{id}} works!</h1>',

  styleUrls: ['./component1.component.css']

})

export class Component1Component implements OnInit {

  id: number;

  constructor(private route: ActivatedRoute) {

    this.route.params.subscribe( params => this.id = params.id );

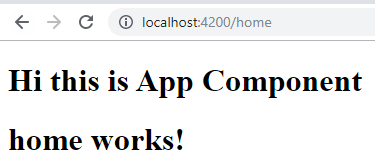
  }

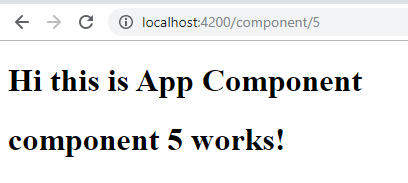
  ngOnInit() {

  }

}

C:\Users\Pujitha.t\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\EC8B0868.tmp*Output:*

It automatically redirects to below url



**Nested (or) Child Routes:**

Create a component that needs to be displayed as child. I created a component named Child Component

child.component.ts

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

@Component({

  selector: 'app-child',

  template: '<h1>Child works!</h1>',

})

export class ChildComponent implements OnInit {

  constructor() { }

  ngOnInit() {

  }

}

*app-routing.module.ts*

const routes: Routes = [

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

  {path: 'component', component: Component1Component  },

  {path: 'component/:id', component: Component1Component,

    children: [

      {

      path:  'nested',

      component:  ChildComponent

      }

    ]

  },

  {path: 'home', component: HomeComponent  }

]

*component1.component.html*

<h1>component {{id}} works!</h1>

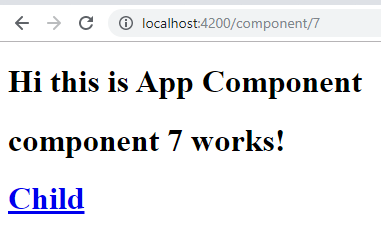
<h1><a [routerLink]="['nested']">Child</a> </h1>

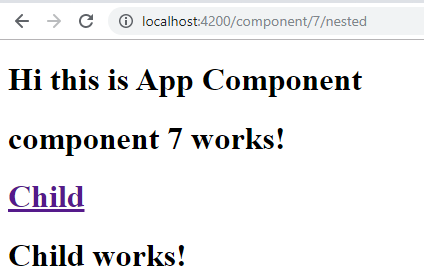
<router-outlet></router-outlet>

Now we have two router-outlets one nested inside another Angular figures out *which* outlet to insert the component in by the *nesting level* of the route and the router outlet.

The Angular Router provides the routerLink directive to create navigation links. This directive takes the path associated with the component to navigate to. For example:

<a [routerLink]="['nested']">Child</a>

*Output:*



**Router Guards & Routing Strategies:**

The Angular Router provides different ***route matching strategies***. The default strategy is simply checking if the current browser’s URL is prefixed with the *path*.

For example our previous route:

{path: 'home', component: HomeComponent  }

Could be also written as:

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

The patchMath attribute specifies the matching strategy. In this case, it’s **prefix** which is the default.

The second  matching strategy is **full**. When it’s specified for a route, the router will check if the the path is exactly **equal** to the path of the current browser’s URL:

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

A ***route guard*** is a feature of the Angular Router that allows developers to run some logic when a route is requested, and based on that logic, it allows or denies the user access to the route. It’s commonly used to check if a user is logged in and has the authorization before he can access a page.

You can add a route guard by implementing the CanActivate interface available from the @angular/router package and extends the canActivate() method which holds the logic to allow or deny access to the route. For example, the following guard will always allow access to a route:

class MyGuard implements CanActivate {

 canActivate() {

      return true;

    }

  }

You can then protect a route with the guard using the canActivate attribute:

{path: 'home', canActivate:[MyGuard], component: HomeComponent  }

**Angular Modules**

**Angular Modules:** Module is just a class, like components and [services](https://malcoded.com/posts/angular-fundamentals-services). Angular code is organized by modules only. These are like packages or bundles containing the required code for a specific use case.

The most prominent module is App-module, which is generated by the CLI for every new angular application. There are other modules also like Http-Client-Module, Forms-Module, etc. App-Module is the root-module of any application which can import other modules. All properties of a module are defined inside @NgModule decorator. Those are,

**Bootstrap -** Defines the root-component of the Application.

**Exports** **-** Defines the components, directives and pipes that need to be exported to other modules.

**Declarations** **-** Array of components, directives and pipes that are used in current module.

**Imports -** Imports Sub-Modules that are useful in present module.

**Providers** **-** Defines any @Injectables via Dependency Injection.

For Example,

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

import { NgModule } from '@angular/core'

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

@NgModule({

  declarations: [AppComponent],

  imports: [BrowserModule],

  providers: [],

  bootstrap: [AppComponent],

})

export class AppModule {}

**Feature Module:** It is an ordinary Angular module just an additional class with the @NgModule decorator and registered metadata, except the fact that isn’t the root module. It partitions areas of the application and collaborates with the root.

The main aim for feature modules is delimiting the functionality that focuses on particular internal business inside a dedicated module, in order to achieve modularity. In addition, it restricts the responsibilities of the root module and assists to keep it thin. Another advantage - it enables to define multiple directives with an identical selector, which means avoiding from directive conflicts.

There are [five types](https://angular.io/guide/module-types) of feature modules i.e. Domain, Routed, Routing, Service and Widget. Each of them concentrates and provides a particular type of utilities.

**Domain Featured Component** - Domain feature modules deliver a user experience dedicated to a particular application domain like editing a customer or placing an order.

**Routed feature modules:** Routed feature modules are domain feature modules whose top components are the targets of router navigation routes. Ex: Lazy-loading module

**Routing modules:** A routing module provides routing configuration for another module and separates routing concerns from its companion module.

**Service feature modules:** Service modules provide utility services such as data access and messaging. Ideally, they consist entirely of providers and have no declarations. Ex: [HttpClientModule](https://angular.io/api/common/http/HttpClientModule). The root AppModule is the only module that should import service modules.

**Widget feature modules:**  A widget module makes components, directives, and pipes available to external modules. Many third-party UI component libraries are widget modules.

**Lazy-loading Module:**

**Shared Modules:** The scenario where you need to create a component that is going to be use on different modules of your application is quite usual. If you try to add the component to multiple modules, angular is going to throw you an error.

Is in this situation that shared modules makes sense. They are going to allow you to use components in multiples modules, share instances of services and also should be the place to create application common pipes and directives.

For Example, let us create a shared module which will perform all operations on dates.

Create a basic module and import the date picker component, but this is not enough, we need to make the Date Picker Component accessible for other modules using this module. For that we need to add the date picker component to the module exports array.

Similarly, can export pipes and directives but you can’t use this to export services. For services you need to add the function forRoot() to the shared Module.

The forRoot() static method provides and configures services at the same time. In the case of this kind of shared Modules (that’s expose services), we need to add them and call the forRoot in root module (i.e. AppModule usually) to be certain of having a global instance of the exposed services.

You can also pass a params to the forRoot() method to configure the service initialization. Which is shown below.

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

import { CommonModule } from '@angular/common';

import { DatepickerComponent } from './datepicker/datepicker.component';

import { DateformaterService } from './datepicker/dateformater.service';

@NgModule({

  imports: [

    CommonModule

  ],

  declarations: [DatepickerComponent],

  exports:[

    DatepickerComponent

  ]

})

export class SharedDatePickerModule {

  static forRoot(culture: string){

    return{

      ngModule: SharedDatePickerModule,

      providers:[{

        provide:DateformaterService, useValue: culture

      }]

    };

  }

 }

Here in service we receive all the passed parameters inside the constructor.

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

@Injectable()

export class DateformaterService {

  constructor(@Optional() private culture: string) { }

}