Learn Angular Step By Step

What is Angular?

Angular is basically is an open-source, JavaScript-based client-side framework that helps us to develop a web-based application. Actually, Angular is one of the best frameworks for developing any Single Page Application or SPA Applications. The definition of Angular according to the official documentation of Angular in the angular.io web site –

“Angular is a structural framework for dynamic web applications. It lets you use HTML as your template language and lets you extend HTML's syntax to express your application components clearly and succinctly. Its data binding and dependency injection eliminate much of the code you currently have to write. And it all happens within the browser, making it an ideal partner with any server technology.” – (courtesy – Angular Conference, 2014).

So, in a very simple word, Angular is -

- An MVC based structure framework
- A Framework for developing SPA (Single Page Application) based application
- Support client-side templating features.
- Provides us a facility to perform unit test so that our code can be tested before deployment.

Benefits of Using Angular

Angular is a client-side based on an open-source framework maintained by Google. The library is capable of very first reading the HTML page, which provides additional custom tag attributes embedded into it. Those attributes are interpreted as directives that instruct Angular Framework to bind input or output parts of the page to a model that is represented by standard JavaScript variables. The values of those JavaScript variables can be manually set within the code or retrieved from static or dynamic JSON resources including any server-side data provider like REST API or others.

The main benefits of using Angular in web applications are:-

- Angular modifies the page DOM directly instead of adding inner HTML code which is faster.
- Data binding does not occur on each control or value change (no change listeners) but at particular points of the JavaScript code execution. That dramatically improves performance, as a single bulk Model/View update replaces hundreds of cascading data change events.
- There is no need to use observable functions. Angular analyzes the page DOM and builds the bindings based on the Angular-specific element attributes. That requires less writing, meaning the code is cleaner, easier to understand, and has fewer errors.
- Extended features such as dependency injection, routing, animations, view encapsulation, and more are available.
- It is supported by IntelliJ IDEA and Visual Studio .NET IDEs.

- It is supported by Google and a great development community.
- Angular is a comprehensive solution for rapid front-end development. It does not need any other plugins or frameworks.
- Angular is unit testing ready, and that is one of its most compelling advantages.

Why Angular is known as Framework?

So before start discussing Angular, before that we need to understand What is Angular? Why did we call Angular as a Framework? As per the dictionary definition, a framework is an essential supporting structure. This single sentence definition very nicely sums up the Angular. But in spite of that, Angular provides a large and helpful ecosystem in which we can find many new tools and utilities, through which we can solve our problem and also can consider those for the new application structure and design. So, if we want to make our life harder and complex, then we need to write and develop our framework.

So, in real life, it is not possible to develop a framework for ourselves. That's why need to use any existing framework which supports our ideas and requirements. A good and stable framework like Angular is already tested and well understood by the others. The use of frameworks isn't uncommon; many programmers from all environments of coding rely on them. Business application developers use frameworks, such as the Microsoft Entity Framework, to ease their pain and speed up development when building database-related applications. For example, Java programmers use the LibGDX framework to help them create games. I hope I have explained to you the need for a framework and, more specifically, the fact that AngularJS is a great choice.

Install Prerequisites for Angular

If we want to develop an application in Angular , then we need to configure and install the below prerequisites for the environments of the Angular –

1. Latest Node JS i.e. LTA 10.16 or above
2. Install Typescript version 3.4 or above
3. Need to install any IDE like Visual Studio Code or Microsoft Visual Studio 2015 or above
4. Also, required to install Angular CLI to run the angular project

What is TypeScript?

As all, we know that Angular 8 or earlier versions (up to Angular 2.0) is heavily depends on Typescript language. So, it is necessary to understand some basic concepts of Typescript. Typescript is basically a super-set scripting language of JavaScript. So, as per Google, the definition of Typescript is -

"TypeScript is a free and open-source programming language developed and maintained by Microsoft. It is a strict syntactical superset of JavaScript and adds optional static typing to the language. Anders Hejlsberg, the lead architect of C# and creator of Delphi and Turbo Pascal, has worked on the development of TypeScript."

Typescript provides much more privilege related to declarative programming like interfaces, classes, modules, static typings over the popular Javascript library and code to the developers. Typescript is totally based on the OOPs concept. Actually, Typescript is basically acted as a transpiler. Although we can develop and compile the code in Typescript. But it is actually a transpiler. Since code written in Typescript has been converted into another language like Javascript after compilation. And then compiled Javascript-based code is run in the browser for the application. So, in a simple word, transpiler means it basically converts one language to another language.

Some Basic Syntax of TypeScript

In this section, we will demonstrate some basic syntax or annotations in Typescript in compared to the JavaScript.

JavaScript	TypeScript
var num = 5;	var num : number = 5;
var num = “Speros”;	var num : string = “Speros”;
var anydata = 123;	var anydata : any = 123;
var list = [1,2,3];	var list : Array<number> = [1,2,3];

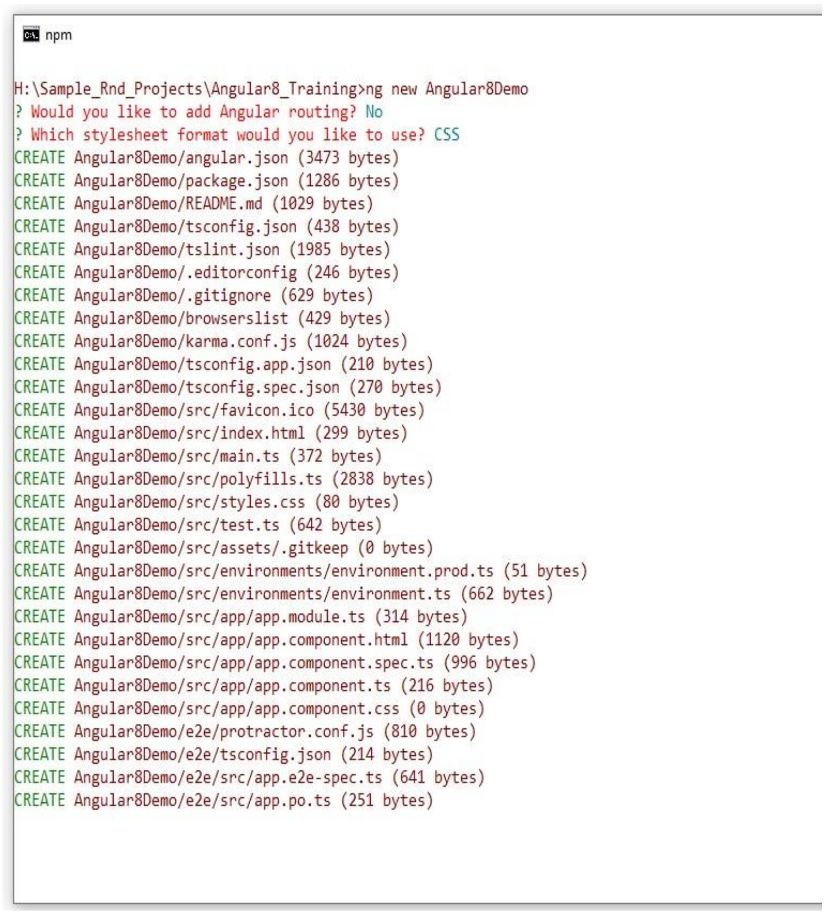
function square(num){
return num * num ;
}
	function square(num : number) : number {
	return num * num ;
	}

The above table shows some basic syntax related to the variable declaration or function declaration. In spite of that, we can also create any class and its related functionalities or objects in Typescript.

Create New Project using Angular CLI

If we want to create a new project on Angular 8, we can use the Angular CLI Commands for that. So, create a new project using Angular CLI we need to perform the below steps –

1. Open the Command Prompt and create a folder
2. Now run the command - ng new AngularDemo
3. Now, Put “N” for Would you like to Add Angular Routing options (Since it needs to be Y when we want to use Routing in our applications).
4. Choose stylesheet style as CSS and then press enter

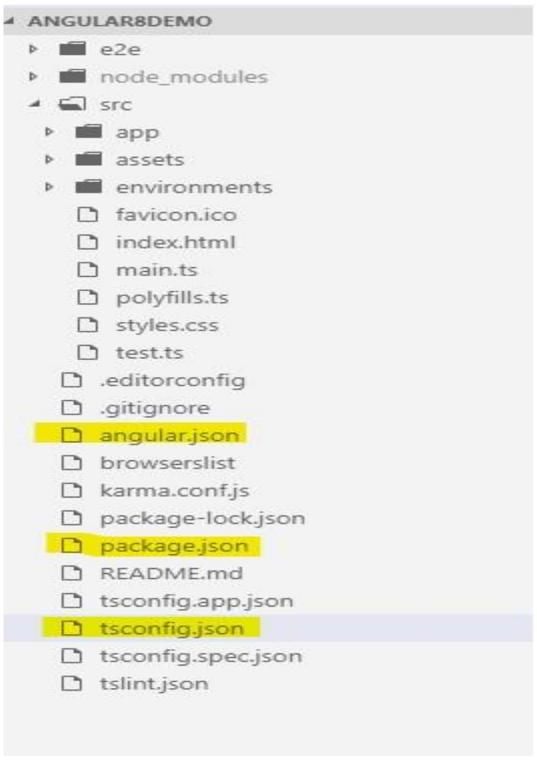


```
npm

H:\Sample_Rnd_Projects\Angular8_Training>ng new Angular8Demo
? Would you like to add Angular routing? No
? Which stylesheet format would you like to use? CSS
CREATE Angular8Demo/angular.json (3473 bytes)
CREATE Angular8Demo/package.json (1286 bytes)
CREATE Angular8Demo/README.md (1029 bytes)
CREATE Angular8Demo/tsconfig.json (438 bytes)
CREATE Angular8Demo/tslint.json (1985 bytes)
CREATE Angular8Demo/.editorconfig (246 bytes)
CREATE Angular8Demo/.gitignore (629 bytes)
CREATE Angular8Demo/browserslist (429 bytes)
CREATE Angular8Demo/karma.conf.js (1024 bytes)
CREATE Angular8Demo/tsconfig.app.json (210 bytes)
CREATE Angular8Demo/tsconfig.spec.json (270 bytes)
CREATE Angular8Demo/src/favicon.ico (5430 bytes)
CREATE Angular8Demo/src/index.html (299 bytes)
CREATE Angular8Demo/src/main.ts (372 bytes)
CREATE Angular8Demo/src/polyfills.ts (2838 bytes)
CREATE Angular8Demo/src/styles.css (80 bytes)
CREATE Angular8Demo/src/test.ts (642 bytes)
CREATE Angular8Demo/src/assets/.gitkeep (0 bytes)
CREATE Angular8Demo/src/environments/environment.prod.ts (51 bytes)
CREATE Angular8Demo/src/environments/environment.ts (662 bytes)
CREATE Angular8Demo/src/app/app.module.ts (314 bytes)
CREATE Angular8Demo/src/app/app.component.html (1120 bytes)
CREATE Angular8Demo/src/app/app.component.spec.ts (996 bytes)
CREATE Angular8Demo/src/app/app.component.ts (216 bytes)
CREATE Angular8Demo/src/app/app.component.css (0 bytes)
CREATE Angular8Demo/e2e/protractor.conf.js (810 bytes)
CREATE Angular8Demo/e2e/tsconfig.json (214 bytes)
CREATE Angular8Demo/e2e/src/app.e2e-spec.ts (641 bytes)
CREATE Angular8Demo/e2e/src/app.po.ts (251 bytes)
```

Now, Angular CLI will create required files for running the Angular Projects along with related packages which will be downloaded in the node_modules folder.

About Project Folder Structure

During the Angular Project creation, Angular CLI create a new folder as per project name i.e. Angular8Demo. So, now open that project in any code editor like Visual Studio Code or Microsoft Visual Studio. The said project folder contains the below folder structure –



The created project contains the following folders –

1. e2e - This folder is for an end to end testing purposes. It contains the configuration files related to performing the unit test of the projects.
2. node_modules - This folder contains the downloaded packages as per the configuration.
3. src - This folder contains the actual source code. It contains 3 subfolders as –
   - app - App folder contains the Angular project-related files like components, HTML files, etc.
   - assets - Assets folder contains any static files like images, stylesheets, custom javascript library files (if any required), etc.
   - environments - Environments folder contains the environment-related files which are required during deployment or build of the projects.

About Different Config Files

When we create any Angular based project using Angular CLI, then every time it will create 3 different configuration files which help us to configure the projects along with its related dependencies. These configuration files are –

tsconfig.json – If tsconfig.json files exist within the project root folder, that means that the project is a basically TypeScript project. The tsconfig.json file specifies the root files and the compiler options required to compile the project. Sample of tsconfig.json as below –

```
{
  "compileOnSave": true,
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/out-tsc",
    "sourceMap": true,
    "declaration": false,
    "module": "esnext",
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "importHelpers": true,
    "target": "es2015",
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [
      "es2018",
      "dom"
    ]
  }
}
```

package.json – package.json is basically a JSON file that contains all information related to the required packages for the project. Also, with the help of this configuration files, we can maintain the Project Name and its related version by using the “name” and “version” property. Also, we can provide the build definition of the project using this file.

```
{
  "name": "angular8-demo",
  "version": "0.0.0",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  "private": true,
  "dependencies": {
    "@angular/animations": "~8.0.0",
    "@angular/common": "~8.0.0",
    "@angular/compiler": "~8.0.0",
    "@angular/core": "~8.0.0",
    "@angular/forms": "~8.0.0",
    "@angular/platform-browser": "~8.0.0",
    "@angular/platform-browser-dynamic": "~8.0.0",
    "@angular/router": "~8.0.0",
    "rxjs": "~6.4.0",
    "tslib": "^1.9.0",
    "zone.js": "~0.9.1"
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "~0.800.0",
    "@angular/cli": "~8.0.2",
    "@angular/compiler-cli": "~8.0.0",
    "@angular/language-service": "~8.0.0",
    "@types/node": "~8.9.4",
    "@types/jasmine": "~3.3.8",
    "@types/jasminewd2": "~2.0.3",
    "codelyzer": "^5.0.0",
    "jasmine-core": "~3.4.0",
    "jasmine-spec-reporter": "~4.2.1",
    "karma": "~4.1.0",
    "karma-chrome-launcher": "~2.2.0",
    "karma-coverage-istanbul-reporter": "~2.0.1",
    "karma-jasmine": "~2.0.1",
    "karma-jasmine-html-reporter": "^1.4.0",
    "protractor": "~5.4.0",
    "ts-node": "~7.0.0",
    "tslint": "~5.15.0",
    "typescript": "~3.4.3"
  }
}
```

angular.json – angular.json file is an Angular Application Environment based JSON file which contains all the information related to the project build and deployment. It tells the system which files need to change when we use ng build or ng serve command.

main.ts - The main.ts file acts as the main entry point of our Angular application. This file is responsible for the bootstrapper operation of our Angular modules. It contains some important statements related to the modules and some initial setup configurations like

- **enableProdMode** – This option is used to disable Angular's development mode and enable Productions mode. Disabling Development mode turns off assertions and other model-related checks within the framework.
- **platformBrowserDynamic** – This option is required to bootstrap the Angular app n the browser.
- **AppModule** – This option indicates which module acts as a root module in the applications.
- **environment** – This option stores the values of the different environment constants.

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

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

```

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

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

@ngModule Metadata

In every Angular application, at least one angular module file is required. An Angular application may contain more than one Angular module. Angular modules is a process or system to assemble multiple angular elements, like components, directives, pipes, service, etc. so that these Angular elements can be combined in such a way that all elements can be related with each other and ultimately create an application.

In Angular, @NgModule decorator is used to defining the Angular module class. Sometimes, this class is called a NgModule class. @NgModule always takes a metadata object, which tells Angular how to compile and launch the application in the browser. So, to define the Angular module, we need to define some steps as follows:

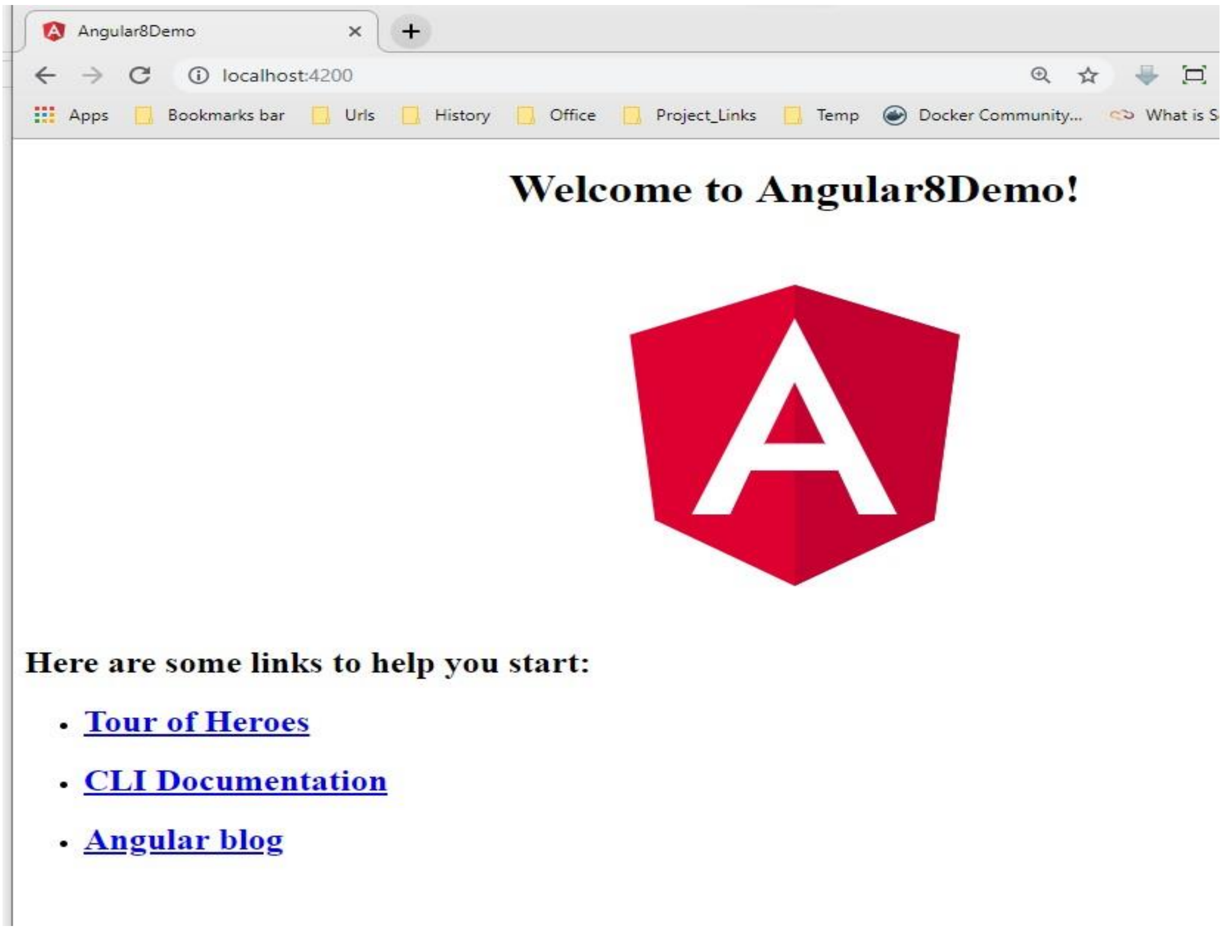
1. First, we need to import Angular BrowserModule into the Angular module file at the beginning. This BrowserModule class is responsible for running the application in the browser.
2. In the next step, we need to declare the Angular elements like component within the Angular module so that those components or elements can be associated with the Angular module.

In the last step, we need to mention one Angular component as a root component for the Angular module. This component is always known as a bootstrap component. So, one Angular module can contain hundreds of components. But out of those components, one component needs to be a root or bootstrap component that will be executed first when the Angular module will be bootstrapped in the browser.

```
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 { }
```

Demo 1: First Program

So, when we create an Angular Project using Angular CLI, it creates new projects along with a module and a default component file. These files normally exist within the app folder. So, first, simply run the Angular Project using ng serve command and the below output will be visible in the browser.



Welcome to Angular8Demo!

Here are some links to help you start:

- Tour of Heroes
- CLI Documentation
- Angular blog

Now, we make some changes in the app.component.ts file and app.component.html file as below –

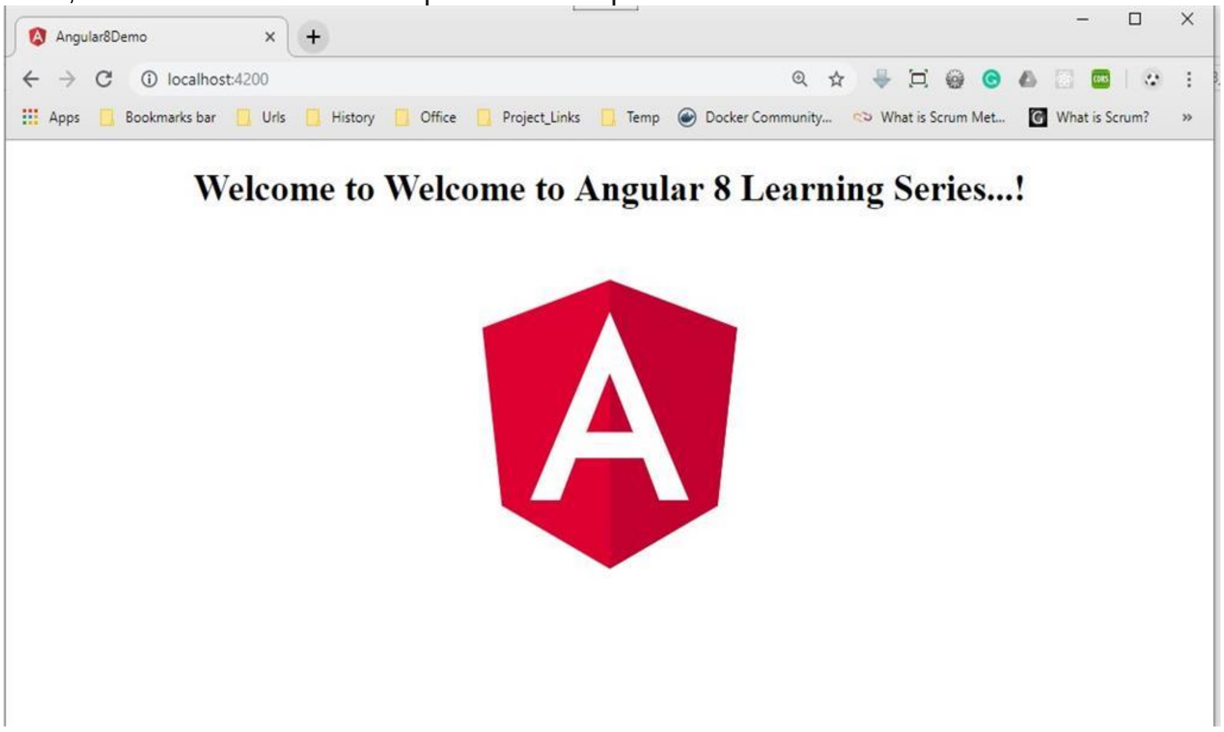
Code of app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Welcome to Angular 8 Learning Series...';
}
```

Code of app.component.html

```
<!--The content below is only a placeholder and can be replaced.-->
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  <img width="300" alt="Angular Logo" src="data:image/svg+xml;base64,PHN2ZyB4bWxucz0iaHR0cDovL3d3dy53My5vcmcvMjAwMC9zdmciIHZpZXdCb3g9IjAgMCAyNTAgMjUwIj4KICAgIDxwYXRoIGZpbGw9IiNERDAwMzEiIGQ9Ik0xMjUgMzBMMzEuOSA2My4ybDE0LjIgMTIzLjFMMTI1IDIzMGw3OC45LTQzLjcgMTQuMi0xMjMuMXoiIC8+CiAgICA8cGF0aCBmaWxsPSIjQzMwMDJGIiBkPSJNMTI1IDMwdjIyLjItLjFWMjMwbDc4LjktNDMuNyAxNC4yLTEyMy4xTDEyNSAzMHoiIC8+CiAgICA8cGF0aCAgZmlsbD0iI0ZGRkZGRiIgZD0iTTEyNSA1Mi4xTDY2LjggMTgyLjZoMjEuN2wxMS43LTI5LjJoNDkuNGwxMS43IDI5LjJIMTgzTDEyNSA1Mi4xem0xNyA4My4zaC0zNGwxNy00MC45IDE3IDQwLjl6IiAvPgogIDwvc3ZnPg==">
</div>
```

Now, refresh the browser to update the output.



Welcome to Welcome to Angular 8 Learning Series...!

What is a Component?

A Component is basically a class that is defined for any visible element or controls on the screen. Every component class has some properties and by using them, we can manipulate the behavior or looks of the element on the screen. So, we can create, update or destroy our own components as per the requirement at any stage of the application. But in TypeScript, a component is basically a TypeScript class decorated with an @Component() decorator. From an HTML point of view, a component is a user-defined custom HTML tag that can be rendered in the browser to display any type of UI element along with some business logic.

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Welcome to Angular 8 Learning Series...';
 }
```

Decorators are mainly JavaScript functions that amend the decorated class in some way. A component is an independent, complete block of code that has the required logic, view, and data as one unit. As a developer for creating a component, we just need to define a class by providing some related configuration objects as an argument or parameter to the decorator function. Logically every component in Angular 8 acts as an MVC concept itself. Since each component class is an independent unit, it is highly reusable and can be maintained without messing with other components.

Why need Component-Based Architecture?

As per the current trend in web application development, the component-based architecture will be act as the most usable architecture in future web development. Because, with the help of this technique, we can reduce both development time and cost in a large volume in any large-scale web development projects. That’s why technical experts currently recommend implementing this architecture in web-based application development. So, before going to discuss components in-depth, let's discuss why this component-based architecture is required for the web-based development.

- **Reusability** – Component-based frameworks are much more useful due to its reusability features in the development. In this framework, components are the most granular units in the development process, and development with components allows gives us a provision of reusability in future development cycles. Since today, technology is changing rapidly. So, if we develop an application in a component-based format, then we are able to swap the best components in and out. A component-based architecture approach allows your application to stay up-to-date over time instead of rebuilding it from scratch.

- **Increase Development Speed** – Component-based development always supports agile methodology based development. Components can be stored in a library that the team can access, integrate, and modify throughout the development process. In a broad sense, every developer has specialized skills. As an example, someone can be an expert in JavaScript, another in CSS, etc. With this framework, every specialized developer can contribute to developing a proper component.
- **Easy Integration** – So, in the component-based framework we can develop a library repository related to the component. This component repository can be used as a centralized code repository for the current development as well as the future new development also. As the other centralized code repository, we can maintain this library in any source control. In this way, the developer can access those repositories and can be updated with new features or functionality as per the new requirement and submit for approval through their own process.
- **Optimize Requirement and Design** – We can use component-library as a base UI component reference source and so using this source analysis team members like product managers, business analysis or technical leaders need to spend less time for finalizing the UI design for their new requirements. Because they already have a bunch of fully tested components with full functionality. Just they need to decide the process about the enhancement points including new business logic only. In this way, this component-based framework provides faster speed for the development process lifecycle.
- **Lower Maintenance Costs** – Since the component-based framework supports reusability, this framework reduces the requirement of the total number of developers when we want to create a new application. Logic-based components are normally context-free, and UI-based components always come with great UX and UI. So, the developer can now focus on integrating those components in the application and how to establish connections between these types of components. Also, other system attributes like security, performance, maintainability, reliability, and scalability, (which are normally known as non-functional requirements or NFRs) can also be tested.

@Component Metadata

So in Angular, when we want to create any new component, we need to use the @Component decorator. @Component decorator basically classifies a TypeScript class as a component object. Actually, @Component decorator is a function that takes different types of parameters. In the @Component decorator, we can set the values of different properties to finalize or manipulate the behavior of the components. The most commonly used properties of the @Component decorator are as follows:

1. **selector** – A component can be used by the selector expression. Many people treat components like a custom HTML tag because finally when we want to use the component in the HTML file, we need to provide the selector just like an HTML tag.
2. **template** – The template is the part of the component which is rendered in the browser. In this property, we can pass the HTML tags or code directly as inline code. Sometimes, we call this the inline template. To write multiple lines of HTML code, all code needs to be covered within the tilt (`) symbol.
3. **templayeUrl** – This is another way of rendering HTML tags in the browser. This property always accepts the HTML file name with its related file path.

Sometimes it is known as the external template. The use of this property is much better when we want to design any complex UI within the component.

4. **moduleId** – This is used to resolve the related path of template URL or style URL for the component objects. It contains the Id of the related modules in which the component is attached or tagged.
5. **styles / stylesUrls** – Components can be used in their own style by providing custom CSS, or they can refer to external style sheet files, which can be used by multiple components at a time. To provide an inline style, we need to use styles, and to provide an external file path or URL, we need to use styleUrls.
6. **providers** – In the real-life application, we need to use or inject different types of custom services within the component to implement the business logic for the component. To use any user-defined service within the component, we need to provide the service instance within the provider. Basically, the provider property is always allowed array-type value. So that we can define multiple service instance names that can be provided by comma separation within this property at a time.

In the below example, we demonstrate how to define a component using some of the above properties like selector and template:

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

@Component({
  selector: 'app-root',
  template: 'Welcome to Angular 8 Learning Series...'
})
export class AppComponent {
}
```

Now, in the below example, we will demonstrate how to use other @Component decorator properties like templateUrls:

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

@Component({
  moduleId: module.id,
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  title = 'Welcome to Angular 8 Learning Series...';
 }
```

So, in the above example, we will separate the HTML file for storing the HTML part related to the components. As per the above example, we need to place both the TypeScript file and HTML file in the same location. If we want to place the HTML in a separate folder, then we can use that file in the component decorator using a relative file path. Below is the sample code is written in the app.component.html file:

```
<div style="text-align:center">
  <h1>
```

```
    Welcome to {{ title }}!
  </h1>
</div>
```

Life Cycle of a Component

Just like other frameworks, Angular components have their own life cycle events that are mainly maintained by Angular itself. Below is the list of life cycle events of any Angular components. In Angular, every component has a life-cycle, a number of different stages it goes through from initialization to destruction. There are eight different stages in the component lifecycle. Every stage is called a life cycle hook event. So, we can use these component lifecycle events in different stages of our application to obtains complete controls on the components.

- **ngOnChanges** – This event executes every time a value of an input control within the component has been changed. This event activates first when the value of a bound property has been changed.
- **ngOnInit** – This event executed at the time of Component initialization. This event is called only once, just after the ngOnChanges() events. This event is mainly used to initialize data in a component.
- **ngDoCheck** – This event is executed every time when the input properties of a component are checked. We can use this life cycle method to implement the checking on the input values as per our own logic check.
- **ngAfterContentInit** – This lifecycle method is executed when Angular performs any content projection within the component views. This method executes only once when all the bindings of the component need to be checked for the first time. This event executes just after the ngDoCheck() method.
- **ngAfterContentChecked** - This life cycle hook method executes every time the content of the component has been checked by the change detection mechanism of Angular. This method is called after the ngAfterContentInit() method. This method is can also be executed on every execution of ngDoCheck() event.
- **ngAfterViewInit** – This life cycle method executes when the component completes the rendering of its view full. This life cycle method is used to initialize the component’s view and child views. It is called only once, after ngAfterContentChecked(). This lifecycle hook method only applies to components.
- **ngAfterViewChecked** – – This method is always executed after the ngAterViewInit() method. Basically, this life cycle method is executed when the change detection algorithm of the angular component occurs. This method automatically executed every execution time of the ngAfterContentChecked().
- **ngOnDestroy** – This method will be executed when we want to destroy the Angular components. This method is very useful for unsubscribing the observables and detaching the event handlers to avoid memory leaks. It is called just before the instance of the component being destroyed. This method is called only once, just before the component is removed from the DOM.

Nested Component

In the above section, we discussed the different aspects of components, like the definition, metadata, and life-cycle events. So, when we develop an application, it is very often that there are some requirements or scenarios where we need to implement a nested component. A nested component is one component inside another component, or we can say it is a parent-child component. The first question that might be raised in our mind: “Does Angular framework support these types of components?” The answer is yes. We can put any number of components within another component. Also, Angular supports the nth level of nesting in general.

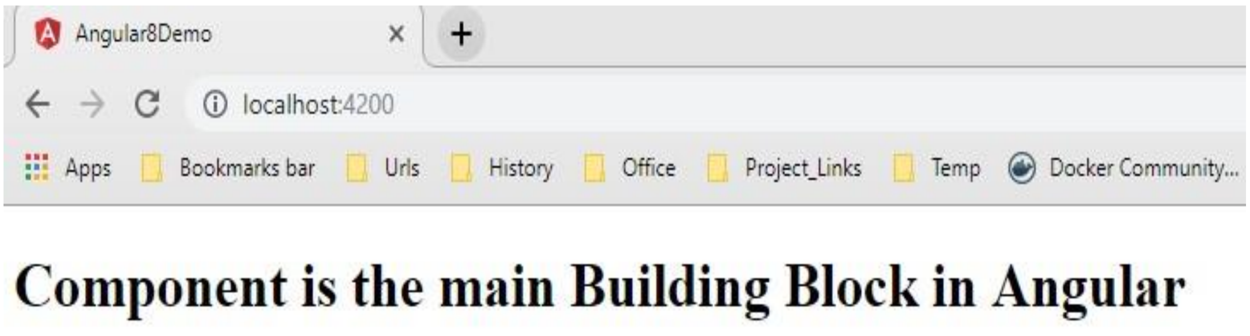
Demo 1: Basic Component

Now, we need to create a new component in our Angular 8 Projects. So, either we can create a new project or using the same project as the previous article. In the project folder, we have a component file named app.component.ts. Now, we will first develop a component with inline HTML contains. For that purpose, we will make the below changes in the existing app.component.ts file.

app.component.ts

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

@Component({
  selector: 'app-root',
  template: '<h1>Component is the main Building Block in Angular
  </h1>'
})
export class AppComponent {

}
```

Now after making the following changes, run the application in the browse.



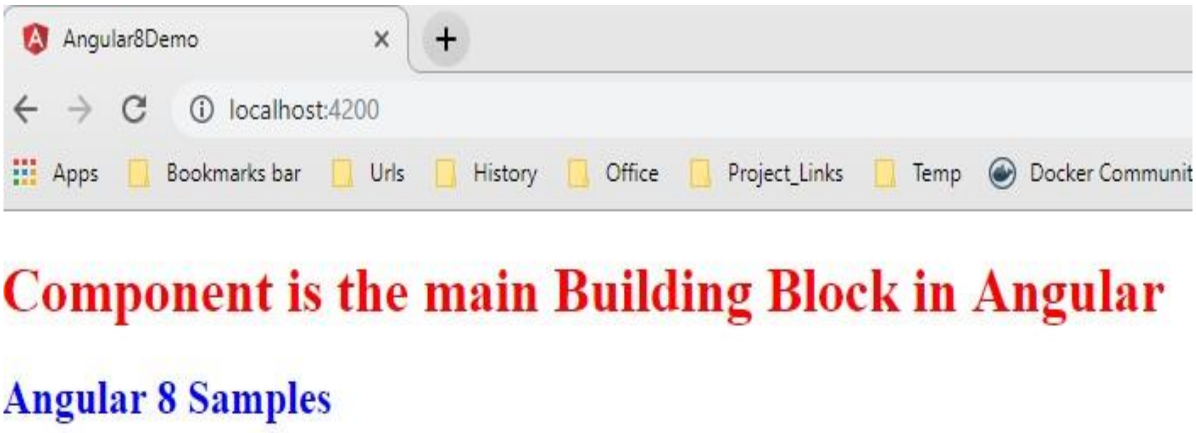
Component is the main Building Block in Angular

Demo 2: Apply Style into the Content

Now, we need to apply the styles in the above component. So, for that, we will make the below changes in the component. First, we will apply the inline styles in the component.

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

@Component({
  selector: 'app-root',
  template: '<h1>Component is the main Building Block in Angular
</h1> <h2>Angular 8 Samples</h2>',
  styles: ['h1{color:red;font-weight:bold}','h2{color:blue}']
})
export class AppComponent {

 }
```

Now after making the following changes, run the application in the browse.



Component is the main Building Block in Angular

Angular 8 Samples

Demo 3: Use External Stylesheet File for Component

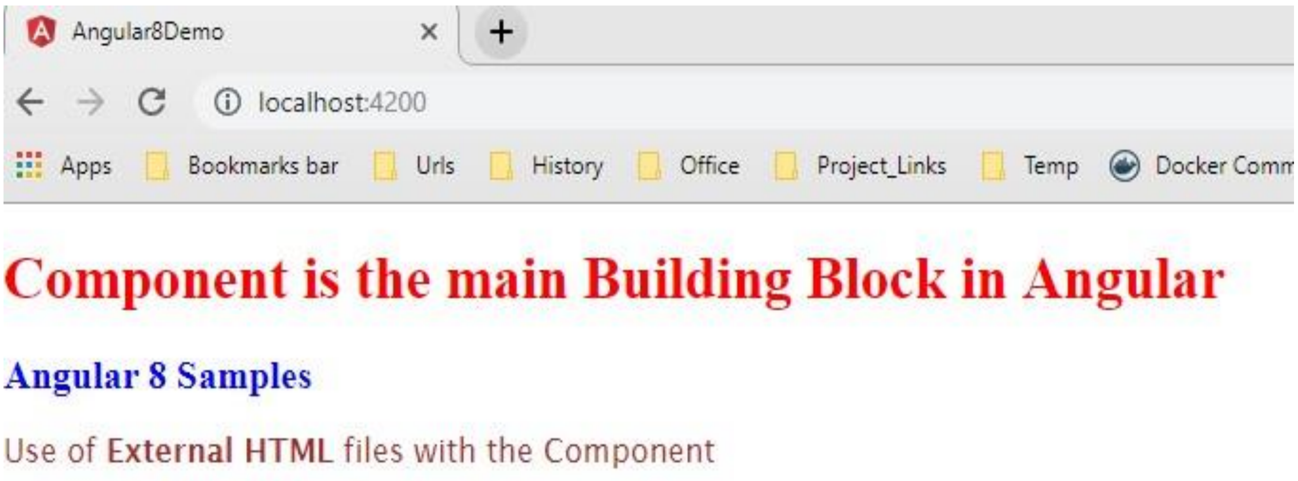
In the previous demo, we used the inline style sheet to decorate the HTML content within the component. Now it is fine when we just need to use styles in one component. But, if we want to apply the same styles in all the components, then we need to use the external stylesheet within the component. Now, we will discuss how to use an external style sheet in any component. First, we need to create a new style sheet file named custom.css within the project and then add the below code within the same file.

```
/* You can add global styles to this file, and also import other
 style files */
h1{
    color:red;
    font-weight:bold;
    font-size: 30px;
}
h2{
    color:blue;
    font-size: 20px;
}

p{
    color:brown;
    font-
family: 'Lucida Sans', 'Lucida Sans Regular', 'Lucida Grande', '
Lucida Sans Unicode', Geneva, Verdana, sans-serif;
}
```

Now, just put the style.css reference path in the styleUrls properties within the app.component.ts file.

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

@Component({
  selector: 'app-root',
  template: '<h1>Component is the main Building Block in Angular
</h1> <h2>Angular 8 Samples</h2>',
  styleUrls : ['./custom.css']
})
export class AppComponent {

}
```

Now reload the browser for the output (which is just same as the previous demo)



Component is the main Building Block in Angular

Angular 8 Samples

Use of External HTML files with the Component

Demo 4: Use External HTML File for Component Content

Similar to the external style, we can also use the external HTML file for the HTML code part. For that, we first need to add a new HTML file called app.component.html in the app folder and write down the below code.

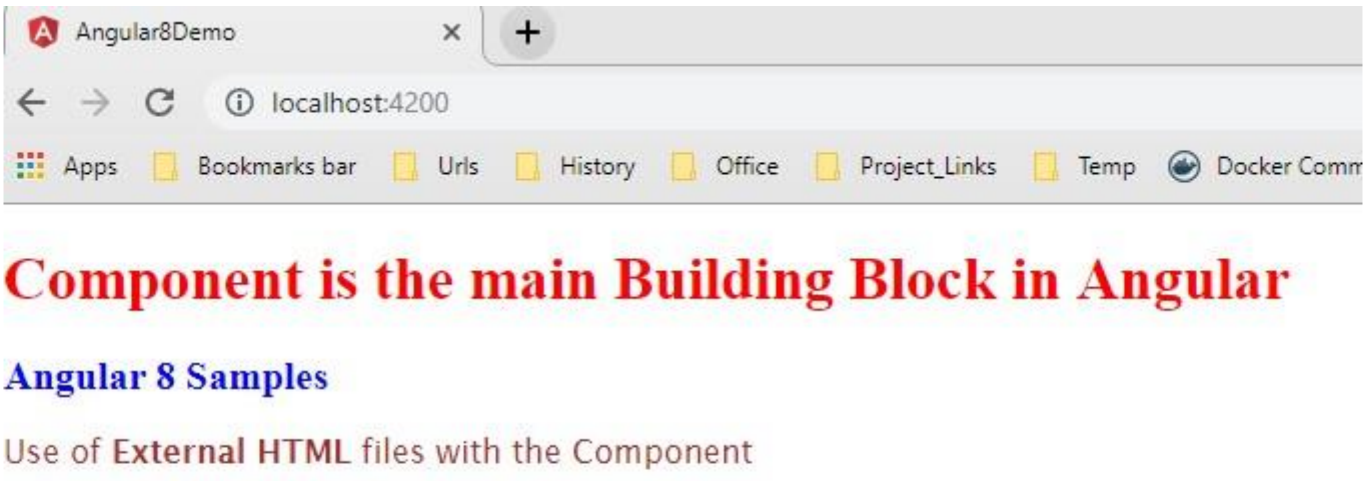
```
<h1>Component is the main Building Block in Angular</h1>
```

```
<h2>Angular 8 Samples</h2>
<p>
    Use of <b>External HTML</b> files with the Component
</p>
```

Now make the below changes in the app.component.ts file for pass the external HTML file path reference.

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

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

 }
```

Now, refresh the browser for the output –



Component is the main Building Block in Angular

Angular 8 Samples

Use of **External HTML** files with the Component

Demo 5: Component Life Cycle Demo

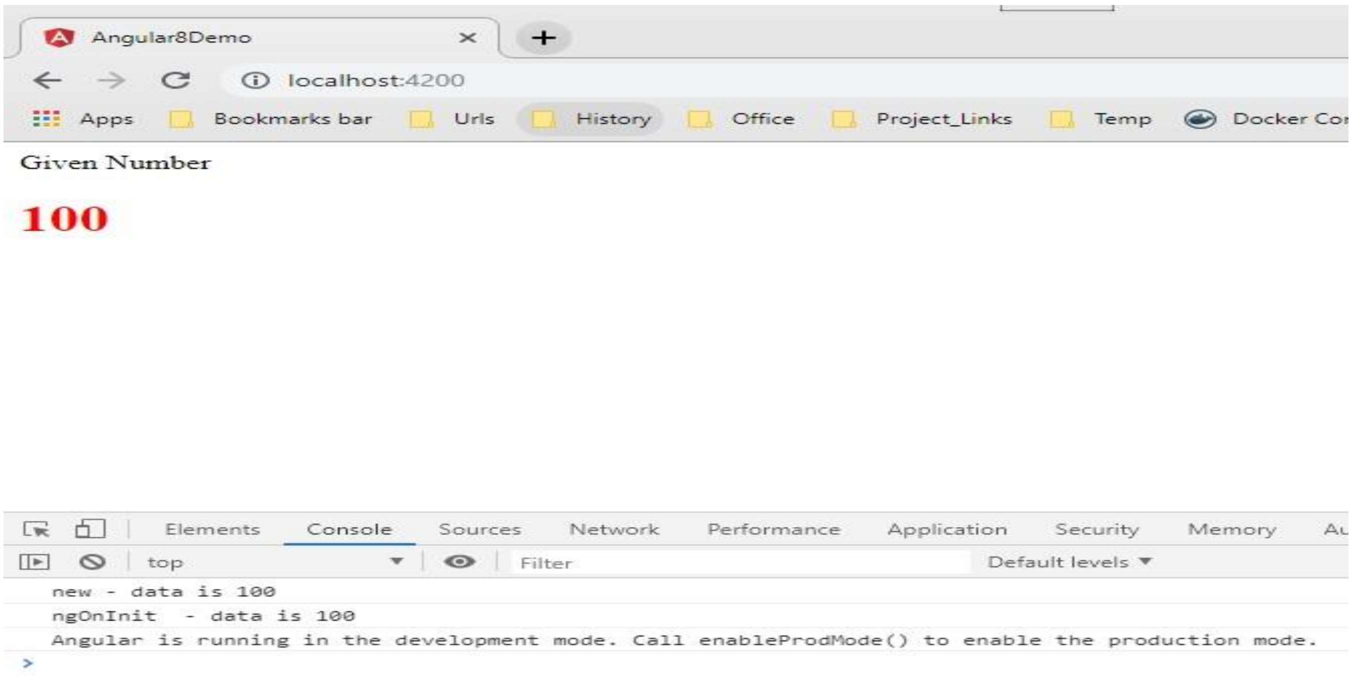
Now in this demo, we will demonstrate the life cycle events of a component. For that, add the below code in the app.component.ts file –

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls : ['./custom.css']
})
export class AppComponent {
  data:number=100;
     constructor() {
         console.log(`new - data is ${this.data}`);
     }
     ngOnChanges() {
         console.log(`ngOnChanges - data is ${this.data}`);
     }
     ngOnInit() {
         console.log(`ngOnInit  - data is ${this.data}`);
     }
 }
```

Now add the below code in app.component.html file –

```
<span class="setup">Given Number</span>
<h1 class="punchline">{{ data }}</h1>
```

Now refresh the browser to check the output –



Given Number

100

```
new - data is 100
ngOnInit  - data is 100
Angular is running in the development mode. Call enableProdMode() to enable the production mode.
```

Demo 6: Nested or Parent-Child Component

In Angular, we can develop any component as a parent-child concept. For that purpose, we need to use the child component selector within the parent component HTML file. So, first, we need to develop a child component as below.

child.component.ts

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

@Component({
  selector: 'child',
  templateUrl: './child.component.html',
  styleUrls : ['./custom.css']
})
export class ChildComponent {

 }
```

child.component.html

```
<h2>It is a Child Component</h2>
<p>
    A component is a Reusable part of the application.
</p>
```

app.component.ts

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

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

 }
```

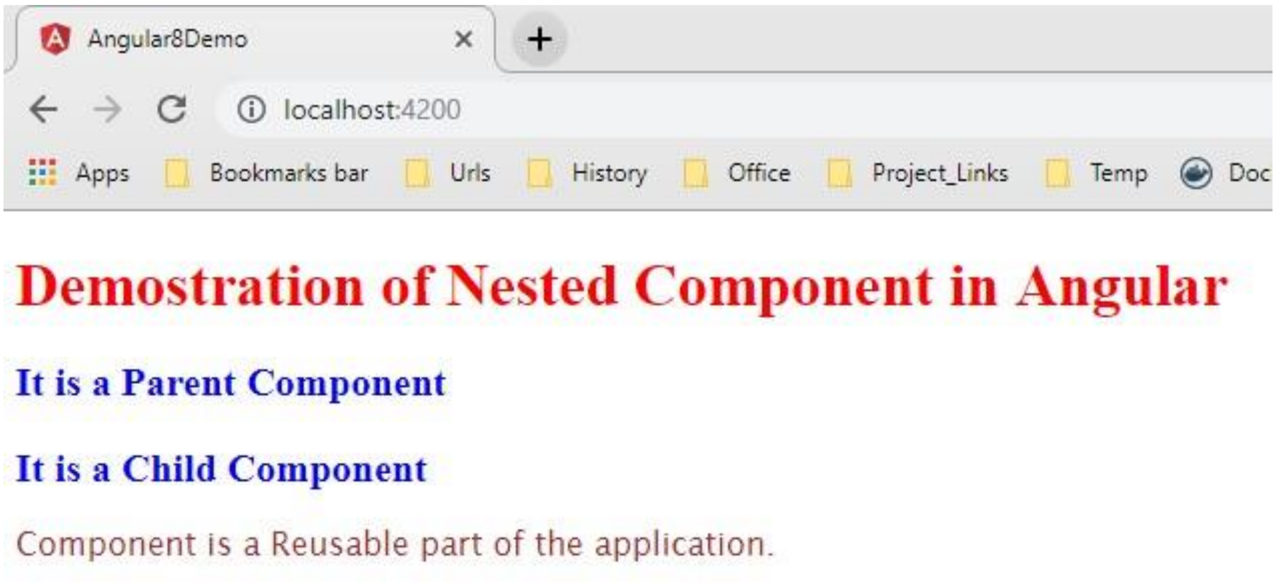
app.component.html

```
<h1>Demostration of Nested Component in Angular</h1>
<h3>It is a Parent Component</h3>
<child></child>
```

Now include the child component into the app.module.ts file as below -

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

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

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

Now, refresh the browser to check the output –



Demostration of Nested Component in Angular

It is a Parent Component

It is a Child Component

Component is a Reusable part of the application.

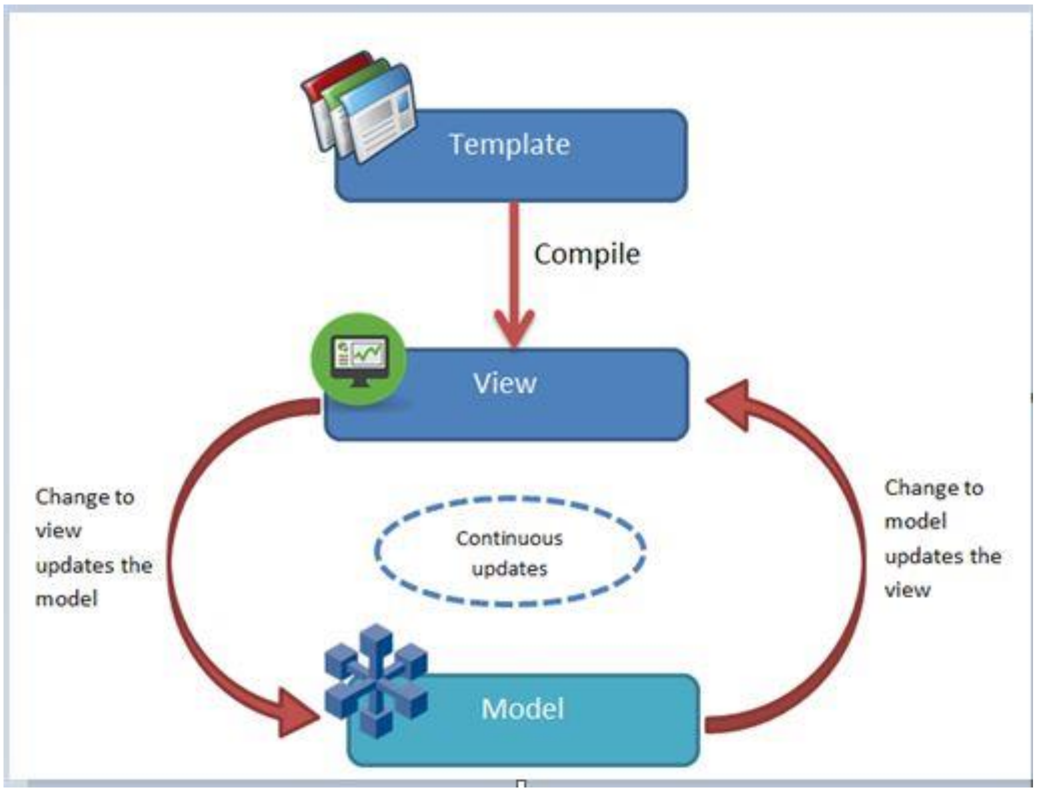
What is Data Binding?

Data binding is one of the finest and useful features of the Angular framework. Due to this feature, the developer needs to write less code compared to any other client-side library or framework. Data binding in an Angular application is the automatic synchronization of data between the model and view components. In Angular, we can always treat the model as a single-source-of-truth of data in our web application with the data binding. In this way, the UI or the view always represents the data model at all times. With the help of data binding, developers can be established a relation between the application UI and business logic. If we establish the data binding in a correct way, and the data provides the proper notifications to the framework, then, when the user makes any data changes in the view, the elements that are bound to the data reflect changes automatically.

Basic Concept of Data Binding

In case of any web-based application development, we always need to develop a connection bridge between the back end, where data is stored and the front end or user interface through which user performs their application data manipulation. Now, this entire process always depends on consecutive networking interactions, repetitive communication between the server (i.e. back end) and the client (i.e. front end).

Due to the consecutive and repetitive communication between client and server, most web framework platforms focus on one-way data binding. Basically, this involves reading the input from DOM, serializing the data, sending it to the back end or server, and waiting for the process to finish. After that, the process modifies the DOM to indicate if any errors occurred or reload the DOM element if the call is successful. While this process provides a traditional web application all the time it needs to perform data processing, this benefit is only really applicable to web apps with highly complex data structures. If your application has a simpler data structure format, with relatively flat models, then the extra work can needlessly complicate the process and decrease the performance of your application.


Template
Compile
View
Change to view updates the model
Continuous updates
Change to model updates the view
Model

The angular framework addresses this with the data binding concept. Data binding provides a continuous data upgradation process so that when the user makes any changes in the interface that will automatically update the data and vice-versa. In this way, the data model of the application always acts as an atomic unit so that the view of the application can be always updated in respect of the data. This process can be done with the help of a complex series of event handlers and event listeners in many frameworks. But that approach can be fragile very quickly. In Angular Framework, this process related to the data becomes one of the primary parts of its architecture. Instead of creating a series of call-backs to handle the changing data, Angular does this automatically without any needed intervention by the programmer. Basically, this feature is a great relief for the programmer and reduces much more development time.

So, the first and foremost advantage of data binding is that it updates the data models automatically in respect of the view. So, when the data model updates, that will automatically update the related view element in the application. In this way, angular provides us a correct data encapsulation on the front end and it also reduced the requirement to perform complex and destructive manipulation of the DOM elements.

Why Data Binding Required?

From day one, the Angular framework provides this special and powerful feature called Data Binding which always brings smoothness and flexibility in any web application. With the help of data binding, developers can gain better control over the process and steps which are related to the data binding process. This process makes the developer's life easier and reduces development time with respect to other frameworks. Some of the reasons related to why data binding is required for any web-based application are listed below –

1. With the help of data binding, data-based web-pages can be developed quickly and efficiently.
2. We always get the desired result with a small volume of coding size.
3. Due to this process, the execution time increases. As a result, it improves the quality of the application.
4. With the help of event emitter, we can achieve better control over the data binding process.

Different Types of Data Binding

In Angular 8, there are four different types of Data binding processes available. They are:

- Interpolation
- Property Binding
- Two-Way Binding
- Event Binding

Interpolation

Interpolation data binding is the most popular and easiest way of data binding in Angular 8. This feature is also available in previous Angular framework versions. Actually, the context between the braces is the template expression that Angular first evaluates and then convert into strings. Interpolation uses the braces expression i.e. {{ }} to render the bound value to the component template. It can be a static string, numeric value, or an object of your data model. In Angular, we use it like this: {{firstName}}.
The below example shows how we can use the interpolation in the component to display data in the front end.

```
<div>
    <span>User Name : {{userName}}</span>
```

```
</div>
```

Property-Based Binding

In Angular 8, another binding mechanism exists, which is called Property Binding. In nature, it is just the same as interpolation. Some people also called this process as one-way binding like the previous AngularJS concept. Property binding used [] to send the data from the component to the HTML template. The most common way to use property binding is to assign any property of the HTML element tag into the [] with the component property value, i.e:

```
<input type=”text” [value]=”data.name”/>
```

To implement the property binding, we will just make the below changes in the previous HTML file from the interpolation sample i.e. interpolation.component.html

```
<div>
    <input [value]="value1" />
    <br /><br />
</div>
```

Event Binding

Event binding is another of the data binding techniques available in Angular. This data binding technique does not work with the value of the UI elements—it works with the event activities of the UI elements like click-event, blur-event, etc. In the previous version of AngularJS, we always used different types of directives like ng-click, ng-blur to bind any particular event action of an HTML control. But in the current Angular version, we need to use the same property of the HTML element (like click, change, etc.) and use it within parentheses. In Angular 8, for properties, we use square brackets, and in events, we use parentheses.

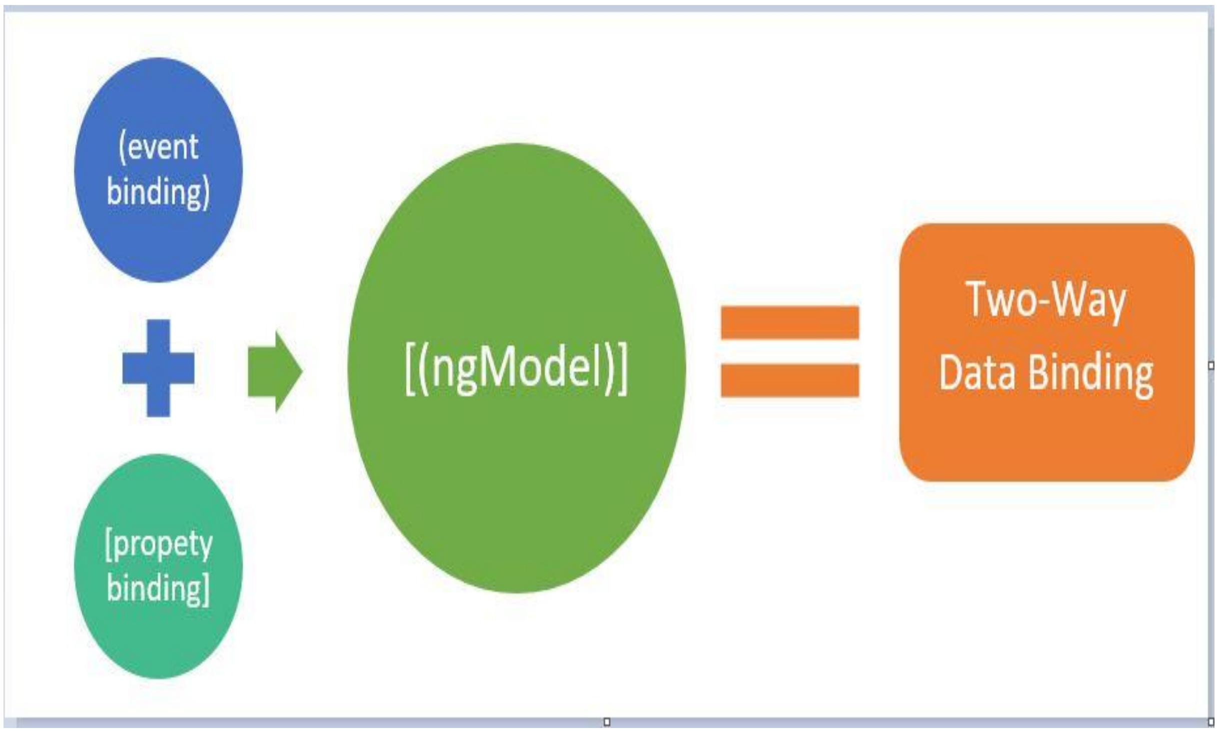
```
<div>
    <input type="submit" value="Submit" (click)="fnSubmit()">
</div>
```

Two-Way Data Binding

In Angular Framework, the most used and important data binding techniques are known as Two-Way Data Binding. Two-way binding is mainly used in the input type field or any form element where the user can provide input values from the browser or provides any value or changes any control value through the browser. On the other side, the same is automatically updated into the component variables and vice versa. Similarly, in Angular 8 we have a directive called ngModel, and it needs to be used as below:

```
<input type="text" [(ngModel)] ="firstName"/>
```

We use [] since it is actually a property binding, and parentheses are used for the event binding concept i.e. the notation of two-way data binding is [()].



ngModel performs both property binding and event binding. Actually, the property binding of the ngModel (i.e. [ngModel]) performs the activity to update the input element with a value. Whereas (ngModel) ((ngModelChange) event) instructs the outside world when any change occurred in the DOM Element.

The below example demonstrates the implementation of two-way binding. In this example, we define a string variable called strName and assign that variable with a Textbox control. So, whenever we change any content in the textbox, the value of the variable will be changed automatically.

```
<div>
    <input [(ngModel)]="strName" type="text"/>
</div>
```

@Input() Decorator

In Angular Framework, each and every component used either as a stateless component or stateful component. Normally, the components are used as a stateless way. But sometimes we need to use some stateful components. The reason behind using a stateful component in a web application is due to the data passing or receiving from the current component to either parent component or a child component. So, in this way, we need to intimate Angular that what type of data or which data may come to our current component. To implement this concept, we need to use the @Input() decorator against any variable. The key features of @Input() decorator are as follows:

- @Input is a decorator to mark an input property. With the help of this property, we can define an input parameter property just like normal HTML tag attributes to pass and bind that value into the component as a property binding.
- @Input decorator always provides a one-way data communications from parent component to child component. Using this feature, we can provide some of the value against any property of a child component from the parent components.
- The component property should be annotated with the @Input decorator to act as an input property. A component can receive value from another component using component property binding.

It can be annotated as any type of property, such as number, string, array, or user-defined class. To use an alias for the binding property name, we need to assign an alias name as @Input(alias). Find the use of @Input with the string data type.

```
@Input() caption : string;
@Input('phoneNo') phoneNo : string;
```

In the above example, testValue is the alias name. The alias name is needed when we want to pass the value to the input properties. If an alias is not defined, then we need to use the input property name to pass the value.
Now, when we use this component, we need to pass the input values as below:

```
<demo-app [name]="'Debasis Saha'" [phoneNo]="9830098300"></demo-
app>
```

@Output() Decorator

In Angular 8, @Output is a decorator to normally used as an output property. @Output is used to define output properties to achieve custom event binding. @Output will be used with the instance of Event Emitter. The key features of the @Output() decorator are as follows:

- The @Output decorator binds a property of a component to send data from one component (child component) to a calling component (parent component).
- This is one-way communication from a child to a parent component.
- @Output binds a property of the EventEmitter class. If we assume component as a control, then output property will act as an event of that control.
- The @Output decorator can also provide options to customize the property name by using the alias name as @Output(alias). In that case, this custom alias name will act as an event binding property name of the component.

It can be entered in any type of property such as number, string, array, or user-defined class. To use an alias for the binding property name, we need to assign an alias name as @Output(alias). Find the use of @Output with the string data type.
Just like Input decorator, output decorator also need to first declare as below –

```
@Output('onSubmit') submitEvent = new EventEmitter<any>();
```

Now, we need to raise the emitter from a point within the component so that the event can be raised and tracked by the parent component.

```
this.submitEvent.emit();
```

If we want to pass any value through this event emitter, then we need to pass that value as a parameter through the emit().
In the parent component, that output property will be defined as below –

```
<demo-app (onSubmit)="receiveData($event)"></demo-app>
```

Demo 1: Interpolation

Now, in this demo, we will show how to use or implement Interpolation in Angular 8 applications for different data types.

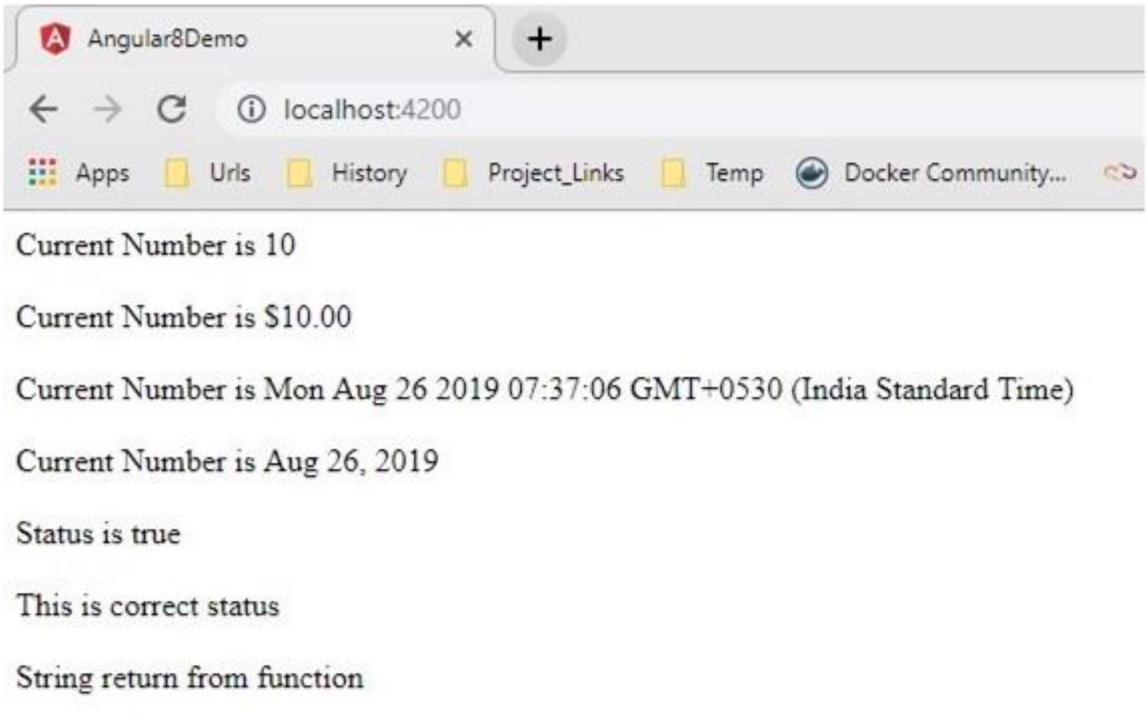
app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls : ['./custom.css']
})
export class AppComponent {
  public value1: number = 10;
    public array1: Array<number> = [10, 22, 14];
    public dt1: Date = new Date();

    public status: boolean = true;

    public returnString(): string {
        return "String return from function";
    }
 }
```

app.component.html

```
<div>
    <span>Current Number is {{value1}}</span>
    <br /><br />
    <span>Current Number is {{value1 | currency}}</span>
    <br /><br />
    <span>Current Number is {{dt1}}</span>
    <br /><br />
    <span>Current Number is {{dt1 | date}}</span>
    <br /><br />
     <span>Status is {{status}}</span>
     <br /><br />
     <span>{{status ? "This is correct status" :"This is false
status"}}</span>
     <br /><br />
     <span>{{returnString()}}</span>
</div>
```

Now, the output of the above code is as below -



```
Current Number is 10

Current Number is $10.00

Current Number is Mon Aug 26 2019 07:37:06 GMT+0530 (India Standard Time)

Current Number is Aug 26, 2019

Status is true

This is correct status

String return from function
```

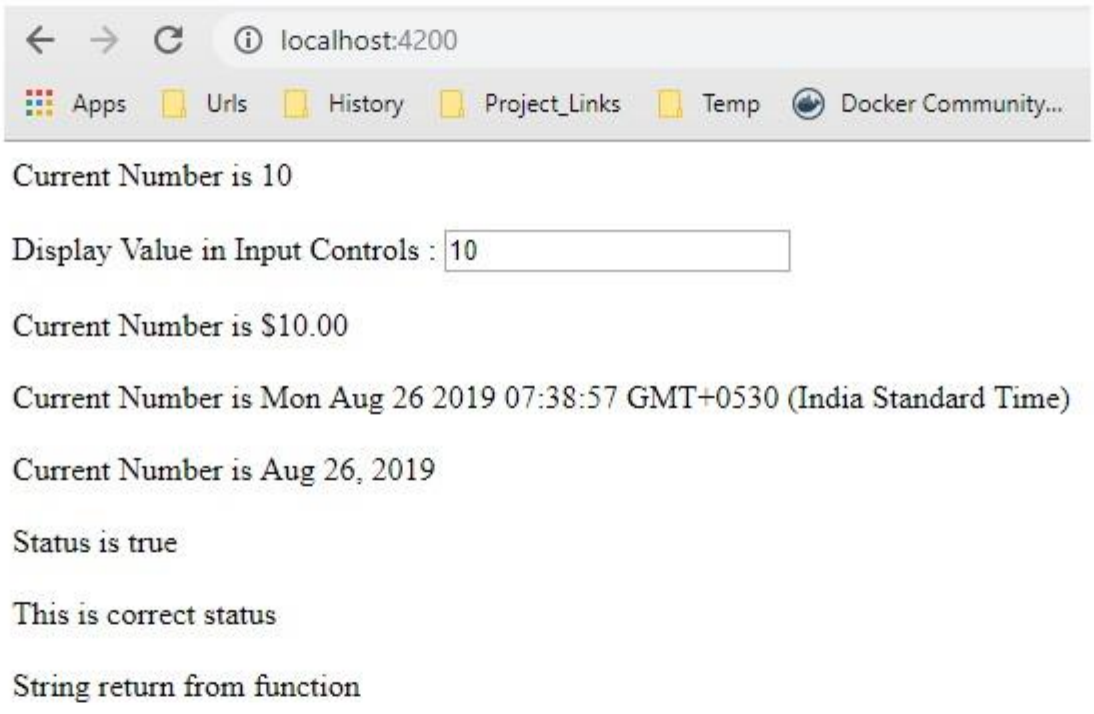
Demo 2: Property-Based Binding

Now, in this demo, we will demonstrate how to use Property-Based binding in Angular 8. For that, we will add one input type text box in the app.component.html file and bind the text box with variable value1 using property binding.

app.component.html

```
<div>
    <span>Current Number is {{value1}}</span>
    <br/><br />
    Display Value in Input Controls : <input [value]="value1" />

    <br /><br />
    <span>Current Number is {{value1 | currency}}</span>
    <br /><br />
    <span>Current Number is {{dt1}}</span>
    <br /><br />
      <span>Current Number is {{dt1 | date}}</span>
      <br /><br />
      <span>Status is {{status}}</span>
      <br /><br />
      <span>{{status ? "This is correct status" :"This is false
   status"}}</span>
      <br /><br />
      <span>{{returnString()}}</span>
 </div>
```

Now, check the output in the browsers –



```
Current Number is 10

Display Value in Input Controls : 10

Current Number is $10.00

Current Number is Mon Aug 26 2019 07:38:57 GMT+0530 (India Standard Time)

Current Number is Aug 26, 2019

Status is true

This is correct status

String return from function
```

Demo 3: Event-Based Binding

In this demo, we will demonstrate how to implement event-based binding in Angular 8.

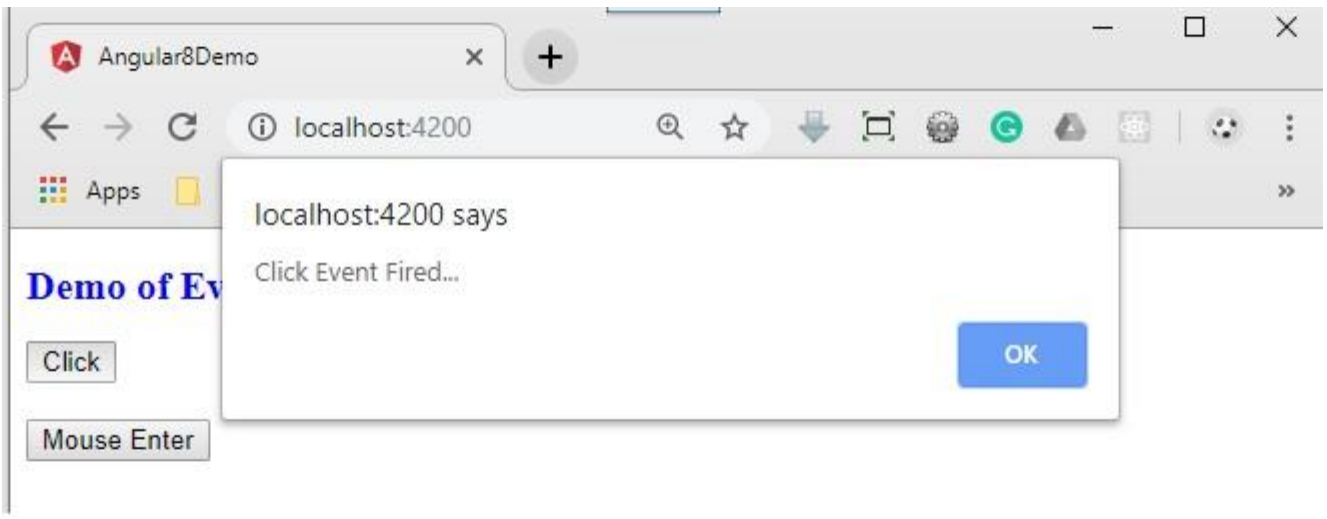
app.component.ts

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

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

  public showAlert() : void {
    console.log('You clicked on the button...');
    alert("Click Event Fired...");
  }
}
```

app.component.html

```
<div>
    <h2>Demo of Event Binding in Angular 8</h2>
    <input type="button" value="Click" class="btn-block" (click)="showAlert()" />
    <br /><br />
    <input type="button" value="Mouse Enter" class="btn-block" (mouseenter)="showAlert()" />
</div>
```

Now, check the output in the browser:-



localhost:4200 says

Click Event Fired...

Demo 4: Two Way Data Binding

Now, in this demo, we will demonstrate how to use two-way data binding in angular. For this purpose, create the below-mentioned components.

app.component.ts

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

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

   public val: string = "";
 }
```

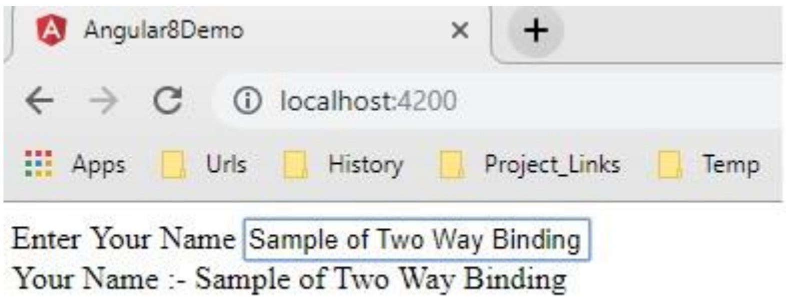
app.component.html

```html
<div>
    <div>
        <span>Enter Your Name </span>
        <input [(ngModel)]="val" type="text"/>
    </div>
    <div>
        <span>Your Name :- </span>
        <span>{{val}}</span>
    </div>
 </div>
```

Now, when we use ngModel or two-way data binding in our components, then we need to include FormsModule of Angular in our own defined app module file as below. Since ngModel will not work without this FormsModule.

```
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 { }
```

Now, check the output in the browser –



Enter Your Name Sample of Two Way Binding
Your Name :- Sample of Two Way Binding

Demo 5: Pass Input Value into Component

In this demo, we will demonstrate how to use or implement the input property of a component. For that purpose, we need to develop the first one component in which input property will be defined.

message.component.ts

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

@Component({
  selector: 'message-info',
  templateUrl: './message.component.html',
  styleUrls : ['./custom.css']
})
export class MessageComponent {

     @Input() public message :string = '';

     @Input('alert-pop') public message1 :string= ''

     public showAlert():void{
         alert(this.message1);
     }
 }
```

message.component.html

```
<div>
    Message : <h2>{{message}}</h2>
    <input type="button" value="Show Alert" (click)="showAlert()
"/>
</div>
```

Now, we need to consume this message-info component into another component and need to pass the input value using input properties.

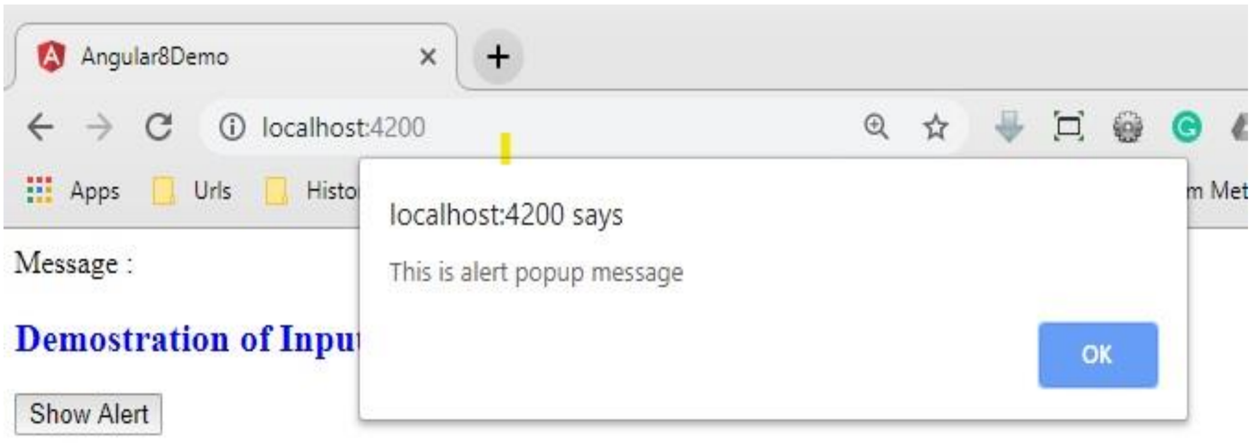
app.component.ts

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

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

   public val: string = "This is alert popup message";

 }
```

app.component.html

```
<div>
    <message-
info [message]="'Demostration of Input Property of a Component'"
 [alert-pop]="val"></message-info>
</div>
```

Now, check the output at the browser –



Message :

Demostration of Inpu

Show Alert

localhost:4200 says

This is alert popup message

OK

Demo 5: Return Output Value from a Component

Now in this demo, we will discuss how to use the output property of any component. For that purpose, we make the following changes in the message-info component to define output property.

message.component.ts

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

@Component({
  selector: 'message-info',
  templateUrl: './message.component.html',
  styleUrls : ['./custom.css']
})
export class MessageComponent {

     @Input() public message :string = '';
     @Input('alert-pop') public message1 :string= ''

     @Output() onSignup  = new EventEmitter<any>();

     public data:any={};

     public showAlert():void{
         alert(this.message1);
     }

     public onSubmit() :void{
       this.onSignup.emit(this.data);
     }
 }
```

message.component.html

```
<div>
    Message : <h2>{{message}}</h2>
    <input type="button" value="Show Alert" (click)="showAlert()"/>
    <br/><br/>
    Provide Full Name : <input type="text" [(ngModel)]="data.name">
    <br/>
    Provide Email Id : <input type="email" [(ngModel)]="data.email">
    <br>
    <input type="button" value="Sign Up" (click)="onSubmit()"/>

 </div>
```

So, in the above part, we defined an output property called onSignup() within the message-info component. Now, we need to consume that event in the parent component as shown below –

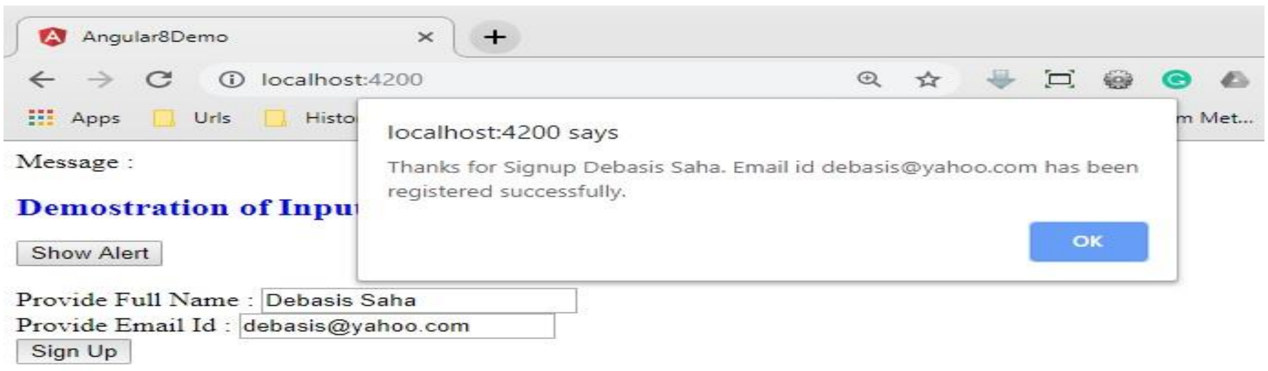
app.component.html

```
<div>
    <message-info [message]="'Demostration of Input Property of a Component'" [alert-pop]="val" (onSignup)="onSignup($event)"></message-info>
</div>
```

app.component.ts

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

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

    public val: string = "This is alert popup message";

    public onSignup(data:any):void{
        let strMessage:string ="Thanks for Signup " + data.name + ". ";
        strMessage += "Email id " + data.email + " has been registered successfully.";
        alert(strMessage);
    }
 }
```

Now refresh the browser to check the output –



localhost:4200 says

Thanks for Signup Debasis Saha. Email id debasis@yahoo.com has been registered successfully.

Message :

Demostration of Inpu

Provide Full Name : Debasis Saha
Provide Email Id : debasis@yahoo.com

What is the Directive?

A directive modifies the DOM by changing the appearance, behavior, or layout of DOM elements. Directives just like Component are one of the core building blocks in the Angular framework to build applications. In fact, Angular 8 components are, in large part, directives with templates. In Angular 8 components have assumed a lot of the roles directives used to. In Angular 8, the major issues related to the template injection and dependency injection are solved with the help of components and issues related to the providing the modification for the generic behavior of the application is done by the help of directives.

So, if we consider the high-level definition of directives, then Directives are acts as a HTML markers on any DOM element (like as an attribute or element name or comment or CSS based style class) which instructed Angular's HTML compiler to attach a specified behaviour on that particular DOM element or to transform that DOM element.

Basic Concept of Directives

In Angular Framework, one of the most important elements is Directives. And if we analyze the directive in brief, then we will discover that the main building block of the Angular 8 framework which is known as Component is basically a directive. So, in a simple word, each and every Angular component is just a directive with a custom HTML template. So, in real word when defining a component as the main building block of Angular application, actually we want to say that directives are the main building blocks of Angular applications.

In general, a directive is a TypeScript based function that executes whenever Angular compiler identified it within the DOM element. Directives are used to provide or generate new HTML based syntax which will extend the power of the UI in an Angular Application. Each directive must have a selector name just like the same as a component – either that name can be from Angular predefined patterns like ng-if or a custom developer-defined name which can be any name but need to indicate the main purpose of the directive. Also, every directive can act as an element or an attribute or a class or a comment in the HTML section.

Why Directives Required?

In Angular Framework, Directives always ensure the high-level of reusability of the UI controls throughout the application. With the help of Directives, we can develop UIs with many movable parts and at the same time, we can streamline the development flow for the engineers. The main reason for using directives in any Angular applications are –

1. **Reusability** – In an Angular application, the directive is a self-sufficient part of the UI. As a developer, we can reuse the directive across the different parts of the application. This is very much useful in any large-scale applications where

multiple systems need the same functional elements like search box, date control, etc.

2. **Readability** – Directive provides much more readability for the developers to understand the production functionality and data flow.
3. **Maintainability** – One of the main use of directive in any application is the maintainability. We can easily decouple the directive from the application and replace the old one with a new one directives.

Component vs Directives

The comparison between Component and directives are as below –

Component	**Directives**
A component is defined with the @Component decorator	A Directive is defined with the @Directives decorator
A component is a directive that uses a shadow DOM to create encapsulated visual behavior called a component. Components are typically used to create UI widgets.	Directive mainly used to provide new behavior within the existing DOM elements.
With the help of the component, we can break down the application in multiple small parts.	With the help of the directive, we can design any type of reusable component.
In the browser DOM, only one component can be activated as a parent component. Other components will act like a child component in that case.	Within a single DOM element, any no of directives can be used.
@View decorator or templateUrl template is mandatory in the component.	Directives don't use View.

@Directive Metadata

@Directive decorator is used to defining any class as an Angular Directive. We can always define any directive to specify the custom behavior of the elements within the DOM. @Directive() metadata is contained below-mentioned options –

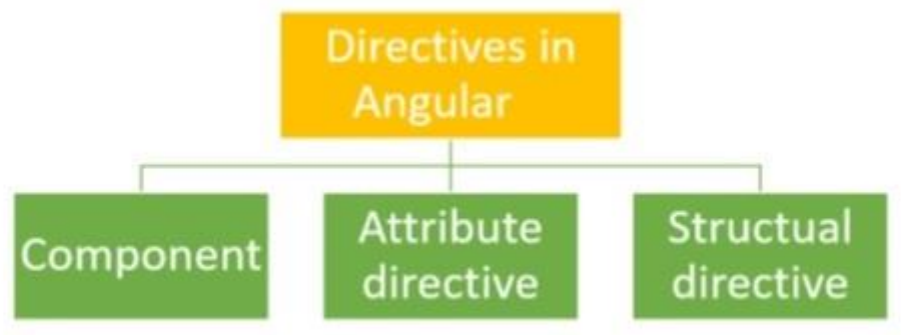
1. **selector** – The selector property is used to identify the directive within the HTML template. Also, with the help of this property, we can initialize the directive from the DOM elements
2. **inputs** - It is used to provide the set of data-bound input properties of the directives.
3. **outputs** – It is used to enumerates any event properties from the directives.
4. **providers** – It is used to inject any provider type like service or components within the directives.

5. **exportAs** – It is used to define the name of the directives which can be used to assign a directive as a variable.

Types of Directives

There are three main types of directives in Angular 8:

- **Component** – Directives with templates.
- **Attribute Directives** – Directives that change the behavior of a component or element but don't affect the template.
- **Structural Directives** – Directives that change the behavior of a component or element by affecting the template or the DOM decoration of the UI.



What is an Attribute Directive?

Attribute directives are mainly used for changing the appearance or behavior of a component or a native DOM element. Attribute directives actually modify the appearance or behavior of an element. These directives actually act as a simple HTML attribute for any HTML tag. There some inbuild attribute directive is available in the framework like ngModel. But, we can also create any type of custom attribute-based directive as per our requirement. In that case, use the attribute directive selector name as an attribute within the HTML tag in the HTML code section.

In-Built Attribute Directives

Angular 8 provides some inbuilt Attribute directives which can be used to change the style or attributes of the HTML elements in the DOM. Those attribute directives are –

- ngClass - ngClass directive changes the class attribute that is bound to the component or element it's attached to.
- ngStyle - ngStyle is mainly used to modify or change the element's style attribute. This attribute directive is quite similar to using style metadata in the component class.

What is Structural Directive?

Other types of directives in the Angular framework are the Structural Directive. Structural directives are mainly used to change the design pattern of the UI DOM elements. In HTML, these directives can be used as a template tag. Using this type of directives, we can change the structure of any DOM elements and can redesign or redecorate those DOM elements. In Angular Framework, there are some system generate structural directives is available like ngIf, ngFor and ngSwitch. We can also create any custom structural directive. The most common example of any custom structural directives is components. Since we can consider every component as a structural directive if that component makes some change in the UI DOM elements style or design. All the system generated structural directives have a template name along with some property value that needs to provide when we define that directives in the HTML code.

In-Build Structural Directive

Angular 8 provides below mentioned built-in directives which can be used within a component to change the elements structure or design.

- ngIf
- ngFor
- ngSwitch

Custom Directive

To create attribute directives, we always need to use or inject the below objects in our custom attribute directive component class. To create an attribute directive, we need to remember the below topics:

1. Import required modules like directives, ElementRef, and renderer from the Angular core library.
2. Create a TypeScript class.
3. Use the @Directive decorator in the class.
4. Set the value of the selector property in the @directive decorator function. The directive will be used, using the selector value on the elements.
5. In the constructor of the class, inject ElementRef and the renderer object.
6. You need to inject ElementRef in the directive’s constructor to access the DOM element.
7. You also need to inject the renderer in the directive’s constructor to work with the DOM’s element style.
8. You need to call the renderer’s setElementStyle function. In this function, we need to pass the instance of the current DOM element with the help of ElementRef as a parameter and setting the behavior or property of the current element.

ElementRef- While creating a custom attribute directive, we inject ElementRef in the constructor to access the DOM element. ElementRef provides access to the underlying native element. With the help of ElementRef, we can obtain direct access to the DOM element using its nativeElement property. In that case, ElementRef is behaving just like a service. That’s all we need to set the element’s color using the browser DOM API.

Renderer- While creating a custom attribute directive, we inject Renderer in the constructor to access the DOM element’s style. Actually, we call the renderer’s setElementStyle function. In this function, we pass the current DOM element with the help of the ElementRef object and set the required attribute of the current element.

HostListener - Sometimes we may need to access the input property within the attribute directive so that, as per the given attribute directive, we can apply a related attribute within the DOM Element. To trap user actions, we can call different methods to handle user actions. We need to use this method to perform any user action. For that purpose, we need to decorate the method with @HostListener method.

Demo 1: Attribute Directive

Now in this demo, we will demonstrate how to use inbuilt attribute directives in Angular Framework.

app.component.ts

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

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

   showColor: boolean = false;

   constructor() { }

   public changeColor(): void {
       this.showColor = !this.showColor;
   }
 }
```

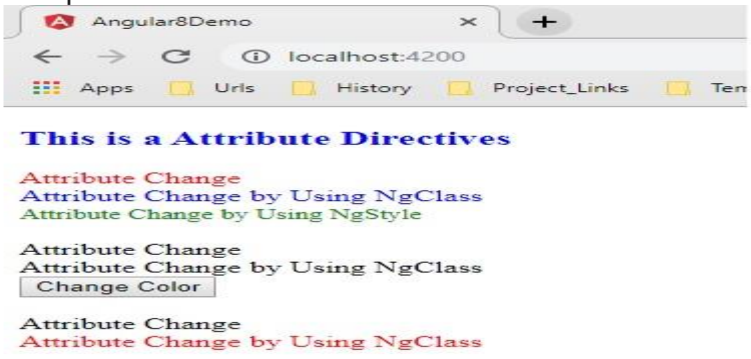
app.component.html

```
<div>
    <h3>This is a Attribute Directives</h3>
    <span [class.red]="true">Attribute Change</span><br />
    <span [ngClass]="{'blue':true}">Attribute Change by Using Ng
Class</span><br />
    <span [ngStyle]="{'font-
size':'14px','color':'green'}">Attribute Change by Using NgStyle
</span>
    <br /><br />
    <span [class.cyan]="showColor">Attribute Change</span><br />
    <span [ngClass]="{'brown':showColor}">Attribute Change by Using NgClass</span><br />
    <input type="button" value="Change Color" (click)="changeColor()" />
     <br /><br />
     <span [class.cyan]="showColor">Attribute Change</span><br />
     <span [ngClass]="{'cyan':showColor, 'red' : !showColor}">Attribute Change by Using NgClass</span><br />
     <br />
 </div>
```

custom.css

```
.red {color:red;}
.blue {color:blue}
.cyan {color : cyan}
.brown {color : brown}
```

Now check the output in the browser.



This is a Attribute Directives

Attribute Change
Attribute Change by Using NgClass
Attribute Change by Using NgStyle

Attribute Change
Attribute Change by Using NgClass

Attribute Change
Attribute Change by Using NgClass

Demo 2: ngIf

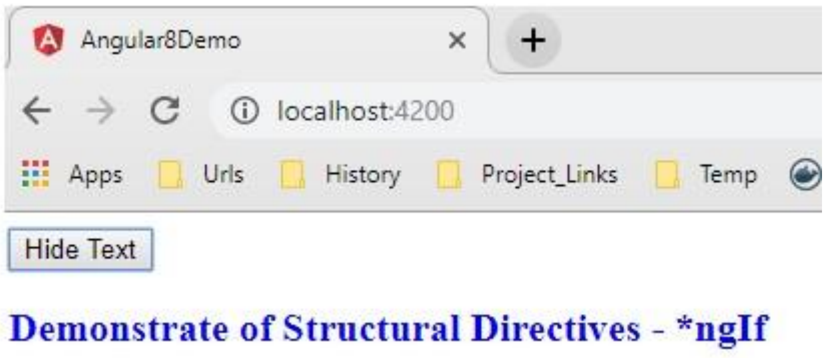
Now in this demo, we will explain how to use ngIf in our angular application.
app.component.ts

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

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

   showInfo: boolean = false;
   caption: string = 'Show Text';

  constructor() { }

  public changeData(): void {
      this.showInfo = !this.showInfo;
      if (this.showInfo) {
          this.caption = 'Hide Text';
      }
      else {
          this.caption = 'Show Text';
      }
  }
}
```

app.component.html

```
<div>
    <input type="button" value="{{caption}}" (click)="changeData()"/>
    <br />
    <h2 *ngIf="showInfo"><span>Demonstrate of Structural Directives - *ngIf</span></h2>
</div>
```

Now check the output in the browser.



Demonstrate of Structural Directives - *ngIf

Demo 3: ngFor

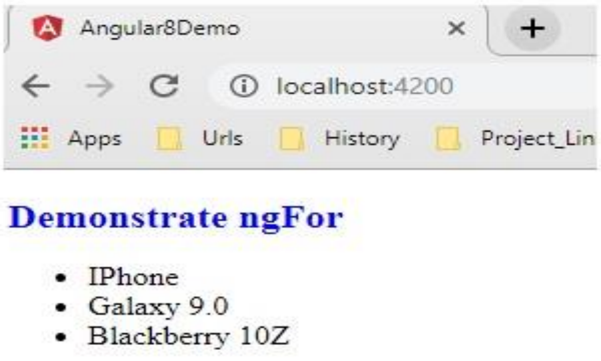
Now in this demo, we will explain how to use ngFor in our angular application.
app.component.ts

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

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

  productList: Array<string> = ['IPhone','Galaxy 9.0','Blackberry 10Z'];

  constructor() { }
}
```

app.component.html

```
<div>
    <h2>Demonstrate ngFor</h2>
    <ul>
        <li *ngFor="let item of productList">
            {{item}}
        </li>
    </ul>
</div>
```

Now check the output in the browser.



```
Demonstrate ngFor

• IPhone
• Galaxy 9.0
• Blackberry 10Z
```

Demo 4: ngSwitch

Now in this demo, we will explain how to use ngSwitch in our angular application.
app.component.ts

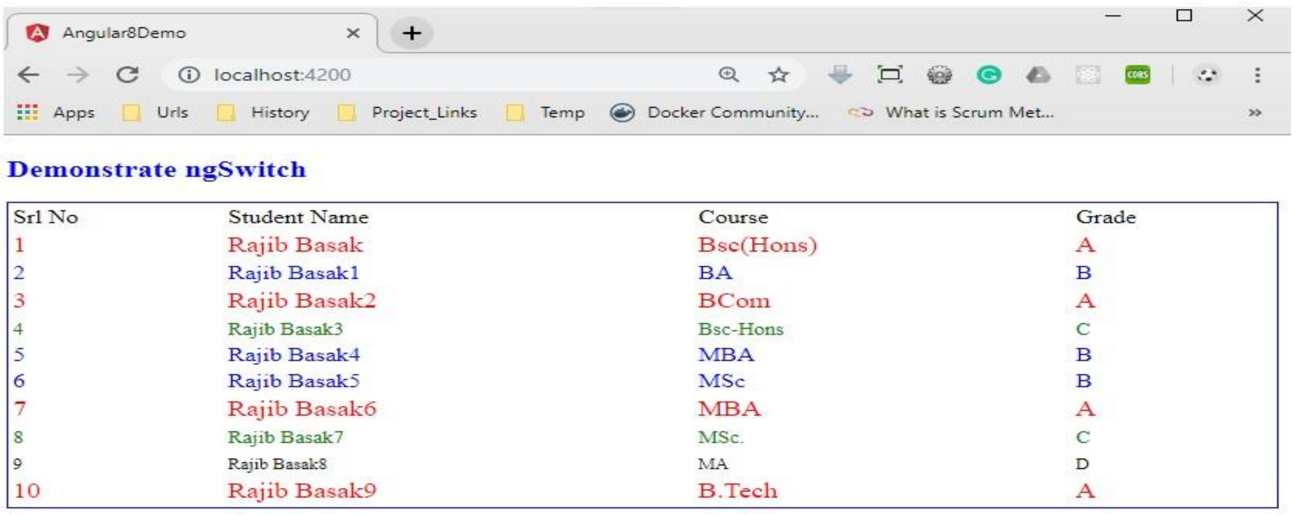
```
import { Component, OnInit } from '@angular/core';

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

  studentList: Array<any> = new Array<any>();

    constructor() { }
    ngOnInit() {
        this.studentList = [
            { SrlNo: 1, Name: 'Rajib Basak', Course: 'Bsc(Hons)', Grade: 'A' },
            { SrlNo: 2, Name: 'Rajib Basak1', Course: 'BA', Grade: 'B' },
            { SrlNo: 3, Name: 'Rajib Basak2', Course: 'BCom', Grade: 'A' },
            { SrlNo: 4, Name: 'Rajib Basak3', Course: 'Bsc-Hons', Grade: 'C' },
            { SrlNo: 5, Name: 'Rajib Basak4', Course: 'MBA', Grade: 'B' },
            { SrlNo: 6, Name: 'Rajib Basak5', Course: 'MSc', Grade: 'B' },
            { SrlNo: 7, Name: 'Rajib Basak6', Course: 'MBA', Grade: 'A' },
            { SrlNo: 8, Name: 'Rajib Basak7', Course: 'MSc.', Grade: 'C' },
            { SrlNo: 9, Name: 'Rajib Basak8', Course: 'MA', Grade: 'D' },
            { SrlNo: 10, Name: 'Rajib Basak9', Course: 'B.Tech', Grade: 'A' }
        ];
    }
}
```

app.component.html

```
<div>
    <h2>Demonstrate ngSwitch</h2>
    <table style="width:100%;border:solid;border-color:blue;border-width:thin;">
        <thead>
            <tr >
                <td>Srl No</td>
                <td>Student Name</td>
                <td>Course</td>
                <td>Grade</td>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="let student of studentList;" [ngSwitch]="student.Grade">
                <td>
                    <span *ngSwitchCase="'A'" [ngStyle]="{'font-size':'18px','color':'red'}">{{student.SrlNo}}</span>
                    <span *ngSwitchCase="'B'" [ngStyle]="{'font-size':'16px','color':'blue'}">{{student.SrlNo}}</span>
```

```
          <span *ngSwitchCase="'C'" [ngStyle]="{'font-size':'14px','color':'green'}">{{student.SrlNo}}</span>
          <span *ngSwitchDefault [ngStyle]="{'font-size':'12px','color':'black'}">{{student.SrlNo}}</span>
        </td>
        <td>
          <span *ngSwitchCase="'A'" [ngStyle]="{'font-size':'18px','color':'red'}">{{student.Name}}</span>
          <span *ngSwitchCase="'B'" [ngStyle]="{'font-size':'16px','color':'blue'}">{{student.Name}}</span>
          <span *ngSwitchCase="'C'" [ngStyle]="{'font-size':'14px','color':'green'}">{{student.Name}}</span>
          <span *ngSwitchDefault [ngStyle]="{'font-size':'12px','color':'black'}">{{student.Name}}</span>
        </td>
        <td>
          <span *ngSwitchCase="'A'" [ngStyle]="{'font-size':'18px','color':'red'}">{{student.Course}}</span>
          <span *ngSwitchCase="'B'" [ngStyle]="{'font-size':'16px','color':'blue'}">{{student.Course}}</span>
          <span *ngSwitchCase="'C'" [ngStyle]="{'font-size':'14px','color':'green'}">{{student.Course}}</span>
          <span *ngSwitchDefault [ngStyle]="{'font-size':'12px','color':'black'}">{{student.Course}}</span>
        </td>
        <td>
          <span *ngSwitchCase="'A'" [ngStyle]="{'font-size':'18px','color':'red'}">{{student.Grade}}</span>
          <span *ngSwitchCase="'B'" [ngStyle]="{'font-size':'16px','color':'blue'}">{{student.Grade}}</span>
          <span *ngSwitchCase="'C'" [ngStyle]="{'font-size':'14px','color':'green'}">{{student.Grade}}</span>
          <span *ngSwitchDefault [ngStyle]="{'font-size':'12px','color':'black'}">{{student.Grade}}</span>
        </td>
      </tr>
    </tbody>
  </table>
</div>
```

Now check the output in the browser.



Demonstrate ngSwitch

Srl No	Student Name	Course	Grade
1	Rajib Basak	Bsc(Hons)	A
2	Rajib Basak1	BA	B
3	Rajib Basak2	BCom	A
4	Rajib Basak3	Bsc-Hons	C
5	Rajib Basak4	MBA	B
6	Rajib Basak5	MSc	B
7	Rajib Basak6	MBA	A
8	Rajib Basak7	MSc.	C
9	Rajib Basak8	MA	D
10	Rajib Basak9	B.Tech	A

Demo 5: Custom Directive – Color Change

Now, in this demo, we will create a custom attribute based directives which will change the color of the selected text on mouseover. For that, we need to first create the directive below.

app.directive.ts

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

@Directive({
    selector: '[colorchange]'
})
export class ColorChangeDirective {
    private _defaulColor = 'red';
    @Input('colorchange') highlightColor: string;

     constructor(private el: ElementRef, private render: Renderer) {
     }

     @HostListener('mouseenter') onMouseEnter() {
         console.log(this.highlightColor);
         this.changecolor(this.highlightColor || this._defaulColor);
     }

     @HostListener('mouseleave') onMouseLeave() {
         console.log(this.highlightColor);
         this.changecolor(null);
     }

    private changecolor(color: string) {
        this.render.setElementStyle(this.el.nativeElement, 'color', color);
    }
}
```

Now, we need to use this custom directive in our app-root component as below.
app.component.ts

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

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

  public message: string = 'Sample Demostration of Attribute Directives using Custom Directives';
  public color: string = 'blue';

}
```

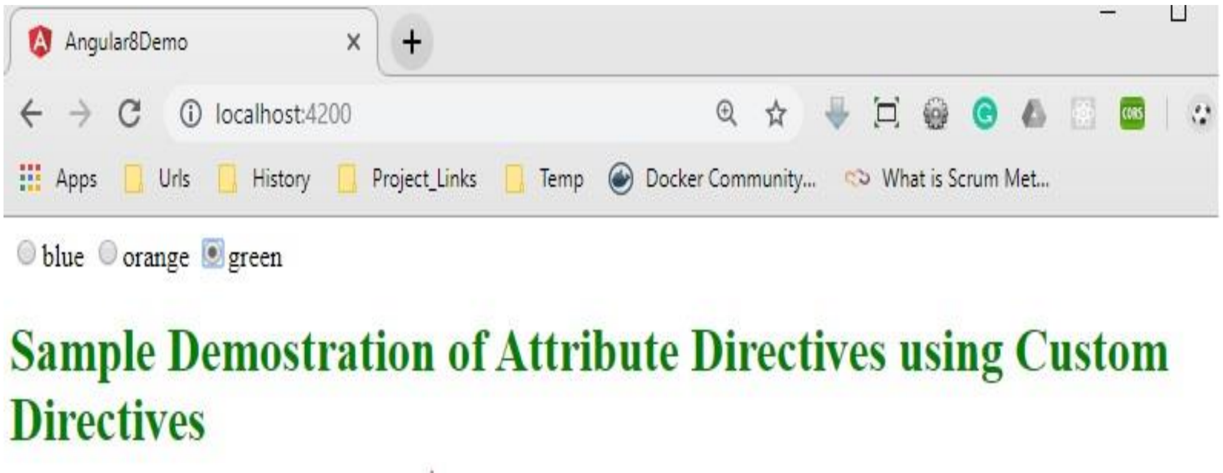
app.component.html

```
<div>
    <input type="radio" name="colors" (click)="color='blue'">blue
    <input type="radio" name="colors" (click)="color='orange'">orange
    <input type="radio" name="colors" (click)="color='green'">green
</div>
<h1 [colorchange]="color">{{message}}</h1>
```

Now, include the above created custom directive in our AppModule as below –

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

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

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

Now, run the output in the browser as below -



blue orange [✓ green]

Sample Demostration of Attribute Directives using Custom Directives

What is Pipe?

When we want to develop any application, we always start the application with a simple task: retrieve data, transform data, and then display the data in front of the user through the user interface. Retrieval of data from any type of data source totally depends on data service providers like web services, Web API, etc. So, once data arrives, we can push those raw data values directly to our user interface for viewing by the user. But sometimes, this is not exactly what happens. For example, in most use cases, users prefer to see a date in a simple format like 15/02/2017 rather than the raw string format Wed Feb 15 2017 00:00:00 GMT-0700 (Pacific Daylight Time). So, it is clear from the above example that some values require editing before being viewed in the user interface. Also, that same type of transformation might be required by us in many different user interfaces. So, in this scenario, we think about some style type properties that we can create centrally and apply whenever we require it. So, for this purpose, the Angular framework introduced Angular pipes, a definite way to write display – value transformations that we can declare in our HTML.

In Angular 8.0, pipes are typescript classes that implement a single function interface, accept an input value with an optional parameter array, and return a transformed value. To perform the value transformation, we can implement any type of business logic as per our requirement. That pipe can be used in any UI to transform that particular type of data as per the desired result.

Basic Concept of Pipes

Basically, pipes provide a sophisticated and handsome way to perform the tasks within the templates. Pipes make our code clean and structured. In Angular, Pipes accept values from the DOM elements and then return a value according to the business logic implemented within the pipes. So, pipes are one of the great features through which can transform our data into the UI and display. But we need to understand one thing very clearly, that pipes do not automatically update our model value. It basically performs the transformation of data and returns to the component. If we need to update the model data after transformation through pipes, then we need to update our model data manually. Expect these, we can use pipes for any of the below reason –

- If we want to retrieve the position of the elements
- If we want to track the user inputs in input type elements
- If we want to restrict the user to input some value in the input controls

Why Pipe Required?

In any application, Pipe can be needed to use as per the following reason –

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

Types of Pipe

In Angular 8, we can categories the pipes in two types i.e. Pure Pipes and Impure Pipes.

Pure Pipes:- Pure pipes in angular are those pipes which always accepts some arguments as input value and return some value as the output according to the input values. Some examples of the pure pipes are – decimal pipes, date pipes, etc. When we use these types of pipes in Angular, we provide input value with related configuration value to the pipes and pipes return us the formatted value as an output.

Impure Pipes:- Impure pipes in angular are those pipes which also accepts the input values, but return the different types of the value set according to the state of the input value. An example of the impure pipes is async pipes. These pipes always store the internal state and return different types of value as the output according to the internal state and logic.

Filter vs Pipe

The concept of the filter is mainly used in the Angular 1.x version. From Angular 2 onwards, Google deprecated the Filter concept and introduced the new concept called Pipe. Now, as per the functionality, filter, and pipes, both are working like the same. But still, there are some differences as below –

1. Filters are acting just like helpers similar to function where we can pass input and other parameters and it will return us a formatted output value. In the case of a pipe, it works as an operator. It also accepts input value and modifies that value to return the desired output.
2. Filters can not handle directly any async type operations. We need to set those values manually. But Pipe can handle async operations on its own. For these types of operations, we need to use async type pipes.

Basic Pipes

Most of the pipes provided by Angular 8 will be familiar with us if we already worked in the previous Angular version. Actually, pipes do not provide any new features in Angular 8. In Angular 8, we can use logic in the template. We can define any function within the pipe class to implement any special type data conversion or business login and then execute that particular function from the HTML template to obtain the desired result. The syntax of the Pipe in the HTML template begins with the input value and then followed the pipe symbol (|) and then need to provide the pipe name. The parameters of that pipe can be sent separately by a colon (:). The order of execution of a pipe is right to left. In General, Pipe is working within the HTML only. The most commonly used built-in pipes are:

- Currency
- Date
- Uppercase
- Lowercase
- JSON
- Decimal
- Percent
- Async

Syntax of Pipes

{{myValue | myPipe:param1:param2 | mySecondPipe:param1}}

Custom Pipes

Now we can define custom pipes in Angular 8 as per our custom business logic. To configure custom pipes, we need to use a pipes object. For this, we need to define a custom pipe with the @Pipe decorator and use it by adding a pipes property to the @View decorator with the pipe class name. We use the transform method to do any logic necessary to convert the value that is being passed in as an input value. We can get a hold of the arguments array as the second parameter and pass in as many as we like from the template. The @Pipe decorator contains the below two properties –

1. name: It contains the name of the pipe which needs to be used in the DOM elements.
2. pure: It accepts the Boolean value. It identifies the pipe is a pure or impure pipe. The default value of the pure property is true i.e. it always considers the custom pipe is a pure type pipe. It means Angular Framework will execute a pure pipe only when it detects a pure change in the input value. Pure change data can be a primitive (means data contains only single values) or non-primitive (means data contains such data type which accepts a group of values). If we need to make the pipe as impure then we need to pass false as the value against this property. In the case of the Impure pipe, the

Angular framework will execute the pipes on every component change detection cycle. In this case, the pipe is often called as often as every keystroke or mouse-move.

When we define any pipe class using @Pipe decorator, we need to implement the PipeTransforms interface which mainly used to accept the input values (optional values) and return the transform values to the DOM.

What is Viewchild?

Basically, Viewchild is one of the new features introduced in the Angular framework. Angular is basically depended on component-based architecture. So when we try to develop any web page or UI, it is most obvious that that page or UI must be a component that basically contains multiple numbers of different functional components as a child component. So in simple words, it is basically a parent component – child component-based architecture. In this scenario, there are some situations occurred when a parent component needs to interact with the child component. We can establish connections between parent and child components in many ways. One of the ways is ViewChild decorator. ViewChild decorator can be used if we need to access the instance of a child component or a directive from the parent component class. So when the need to invoke any method of the child component from the parent component, it can inject the child component as a Viewchild within the parent component. In cases where you'd want to access multiple children, you'd use ViewChildren instead.

To implement ViewChild, we need to use @ViewChild decorator in the parent component. The @ViewChild decorator provides access to the class of child components from the parent component. The @ViewChild decorator is basically a function or method which accepts the name of a component class as its input and finds its selector in the template of the related component to bind to. The basic syntax of the ViewChild decorator in Angular 8 as below –

@ViewChild(ChildComponent, { static:true}) _calculator: ChildComponent;

So, as per the above example, ViewChild decorator contains the following metadata:-

1. selector: It contains the directive or component name which need to use as view child
2. static: It accepts the Boolean value. We need to pass true as a value if we want to resolve the query result before change detection occurs. So, when we pass the value as true, then the Angular framework tries to locate that element at the time of component initialization i.e. in the ngOnInit method. If we pass the value as false, then Angular will try to find the element after the initializing of the view i.e. in the ngAfterViewInit method.

Demo 1: Basic Pipes Demo

In this example, we will demonstrate how to use the inbuilt pipes in Angular 8.

app.component.ts

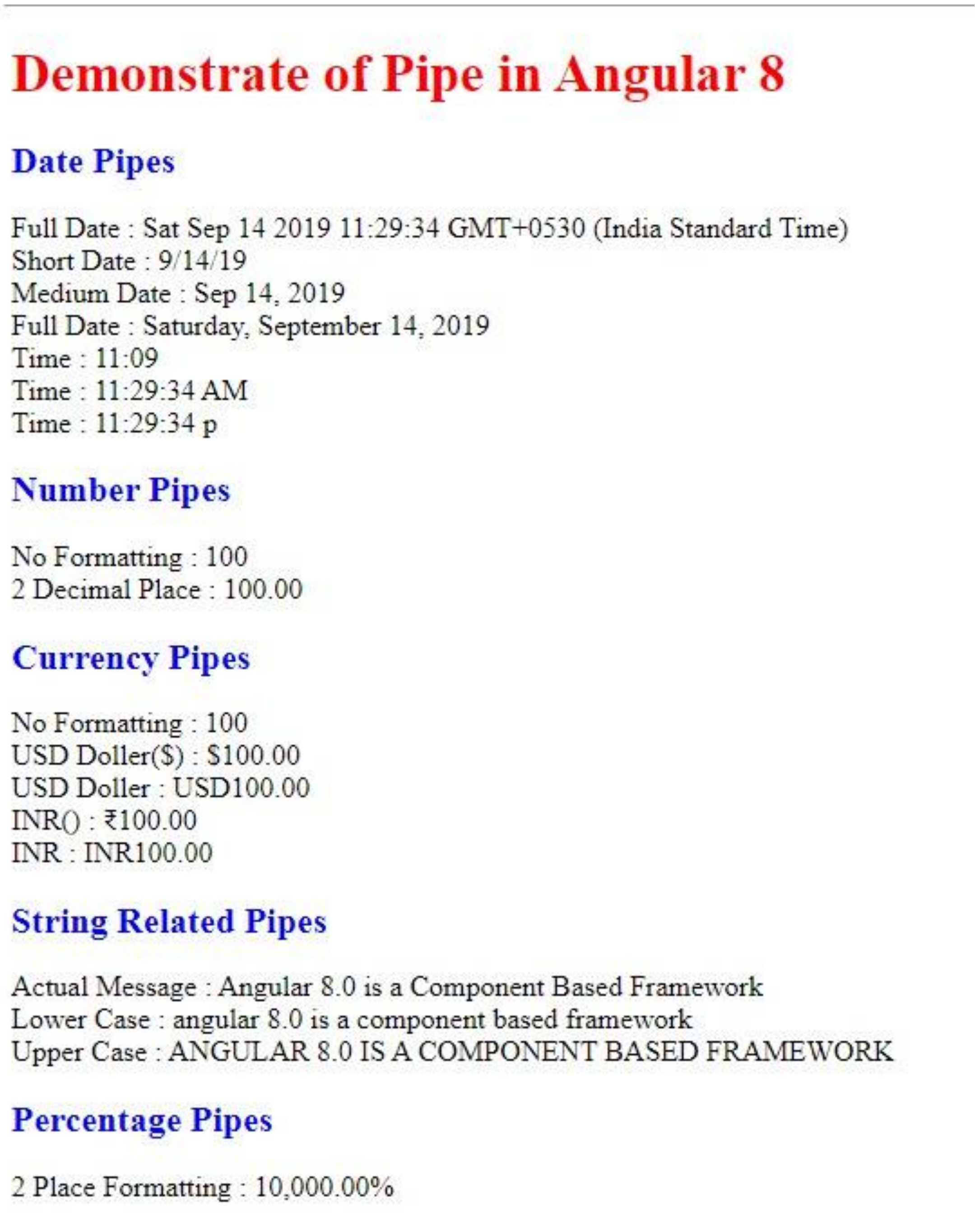
```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls : ['./custom.css']
})
export class AppComponent implements OnInit {
  public todayDate: Date;
    public amount: number;
    public message: string;

    constructor() { }

    ngOnInit(): void {
      this.todayDate = new Date();
      this.amount = 100;
      this.message = "Angular 8.0 is a Component Based Framework";
    }
  }
```

app.component.html

```
<div>
    <h1>Demonstrate of Pipe in Angular 8</h1>
    <h2>Date Pipes</h2>
    Full Date : {{todayDate}}<br />
    Short Date : {{todayDate | date:'shortDate'}}<br />
    Medium Date : {{todayDate | date:'mediumDate'}}<br />
    Full Date : {{todayDate | date:'fullDate'}}<br />
    Time : {{todayDate | date:'HH:MM'}}<br />
    Time : {{todayDate | date:'hh:mm:ss a'}}<br />
      Time : {{todayDate | date:'hh:mm:ss p'}}<br />

      <h2>Number Pipes</h2>
      No Formatting : {{amount}}<br />
      2 Decimal Place : {{amount |number:'2.2-2'}}

      <h2>Currency Pipes</h2>
      No Formatting : {{amount}}<br />
      USD Doller($) : {{amount |currency:'USD':true}}<br />
      USD Doller : {{amount |currency:'USD':false}}<br />
      INR() : {{amount |currency:'INR':true}}<br />
      INR : {{amount |currency:'INR':false}}<br />

      <h2>String Related Pipes</h2>
      Actual Message : {{message}}<br />
```

```
        Lower Case : {{message | lowercase}}<br />
        Upper Case : {{message | uppercase}}<br />

        <h2> Percentage Pipes</h2>
        2 Place Formatting : {{amount | percent :'.2'}}<br /><br />
    </div>
```

Now check the output into the browse –



```
Demonstrate of Pipe in Angular 8

Date Pipes

Full Date : Sat Sep 14 2019 11:29:34 GMT+0530 (India Standard Time)
Short Date : 9/14/19
Medium Date : Sep 14, 2019
Full Date : Saturday, September 14, 2019
Time : 11:09
Time : 11:29:34 AM
Time : 11:29:34 p

Number Pipes

No Formatting : 100
2 Decimal Place : 100.00

Currency Pipes

No Formatting : 100
USD Doller($) : $100.00
USD Doller : USD100.00
INR() : ₹100.00
INR : INR100.00

String Related Pipes

Actual Message : Angular 8.0 is a Component Based Framework
Lower Case : angular 8.0 is a component based framework
Upper Case : ANGULAR 8.0 IS A COMPONENT BASED FRAMEWORK

Percentage Pipes

2 Place Formatting : 10,000.00%
```

Demo 2: Custom Pipe – Proper Case

Now, its time to define some custom pipe as per our requirement. Since in the above example, we see that in general Angular 8 framework provides us to UpperCase and LowerCase pipes against any string type value as in-build pipes. But, it does not provide any proper case type pipes. So, let's define a pipe that will transfer any string value as proper case and return the result. For that purpose, add the below typescript class files –

propercase.pipe.ts

```
import { Pipe, PipeTransform } from "@angular/core"

@Pipe({
    name: 'propercase'
})

export class ProperCasePipe implements PipeTransform {
    transform(value: string, reverse: boolean): string {
        if (typeof (value) == 'string') {
             let intermediate = reverse == false ? value.toUpperCase() : value.toLowerCase();
             return (reverse == false ? intermediate[0].toLowerCase() :
                 intermediate[0].toUpperCase()) + intermediate.substr(1);
          }
          else {
              return value;
          }
      }
 }
```

Now, include the proper case pipe in the AppModule as below –

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

import { AppComponent } from './app.component';
import { ColorChangeDirective } from './app.directive';
import { ProperCasePipe } from './propercase.pipe';

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

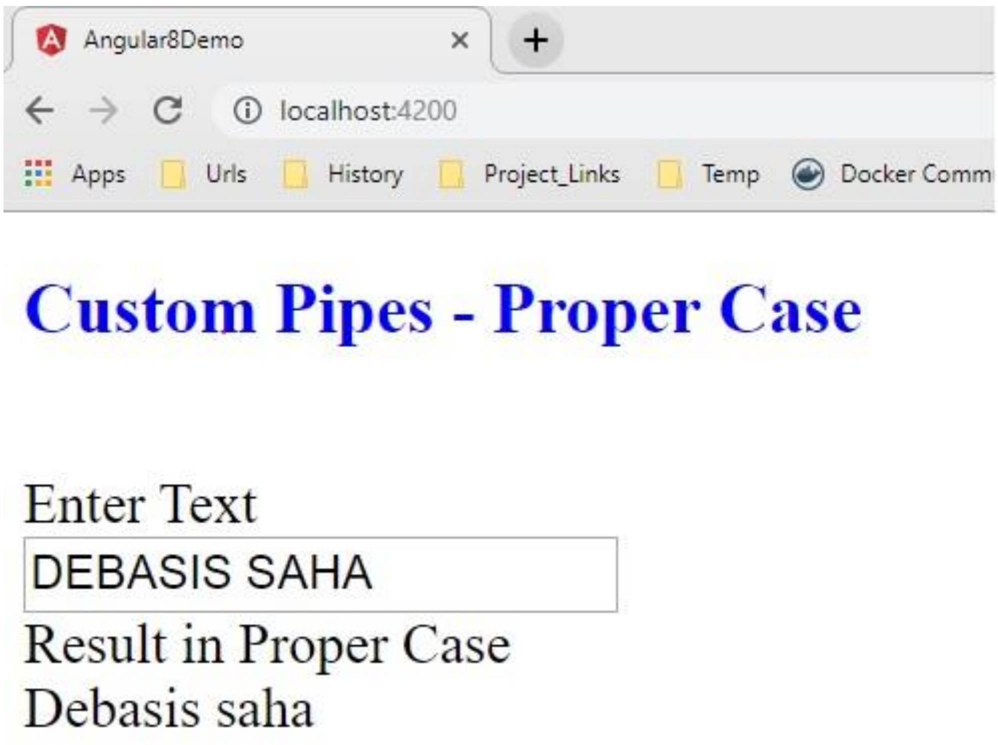
app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls : ['./custom.css']
})
export class AppComponent implements OnInit {
  public message: string;

  constructor() { }

  ngOnInit(): void {
    this.message = "This is a Custom Pipe";
  }
}
```

app.component.html

```
<div>
    <div class="form-horizontal">
        <h2 class="aligncenter">Custom Pipes - Proper Case</h2><br />
        <div class="row">
            <div class="col-xs-12 col-sm-2 col-md-2">
                <span>Enter Text</span>
            </div>
            <div class="col-xs-12 col-sm-4 col-md-4">
                <input type="text" id="txtFName" placeholder="Enter Text" [(ngModel)]="message" />
            </div>
        </div>
        <div class="row">
            <div class="col-xs-12 col-sm-2 col-md-2">
                <span>Result in Proper Case</span>
            </div>
            <div class="col-xs-12 col-sm-4 col-md-4">
                <span>{{message | propercase}}</span>
            </div>
        </div>
    </div>
</div>
```

Now run the output into the browser -



Custom Pipes - Proper Case

Enter Text
DEBASIS SAHA
Result in Proper Case
Debasis saha

Demo 3: Viewchild

Now, in this demo, we will discuss how to use ViewChild decorator within a component. For that purpose, first, need to create a child component that will be used as a ViewChild in the parent or root component.

child.component.ts

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

@Component({
    selector: 'child',
    templateUrl: 'child.component.html'
})

export class ChildComponent implements OnInit {
    private firstNumber: number = 0;
      private secondNumber: number = 0;
      private result: number = 0;

    @Output() private addNumber: EventEmitter<number> = new EventEmitter<number>();
    @Output() private subtractNumber: EventEmitter<number> = new EventEmitter<number>();
    @Output() private multiplyNumber: EventEmitter<number> = new EventEmitter<number>();
    @Output() private divideNumber: EventEmitter<number> = new EventEmitter<number>();

    constructor() { }

    ngOnInit(): void {
    }

    private add(): void {
        this.result = this.firstNumber + this.secondNumber;
        this.addNumber.emit(this.result);
    }

    private subtract(): void {
        this.result = this.firstNumber - this.secondNumber;
        this.subtractNumber.emit(this.result);
    }

    private multiply(): void {
        this.result = this.firstNumber * this.secondNumber;
        this.multiplyNumber.emit(this.result);
    }

    private divide(): void {
        this.result = this.firstNumber / this.secondNumber;
        this.divideNumber.emit(this.result);
    }

    public clear(): void {
        this.firstNumber = 0;
        this.secondNumber = 0;
        this.result = 0;
    }
}
```

child.component.html

```
<div class="ibox-content">
    <div class="row">
        <div class="col-md-4">
            Enter First Number
        </div>
        <div class="col-md-8">
            <input type="number" [(ngModel)]="firstNumber" />
```

```
          </div>
      </div>
        <div class="row">
            <div class="col-md-4">
                Enter Second Number
            </div>
            <div class="col-md-8">
                <input type="number"  [(ngModel)]="secondNumber" />
            </div>
        </div>
        <div class="row">
            <div class="col-md-4">
            </div>
            <div class="col-md-8">
               <input type="button" value="+" (click)="add()" />

                <input type="button" value="-" (click)="subtract()" />

                <input type="button" value="X" (click)="multiply()" />

                <input type="button" value="/" (click)="divide()" />
            </div>
        </div>
    </div>
```

Now, include this child component in the app.module.ts file.

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

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

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

Now, use the above child component in the root component as below –

app.component.ts

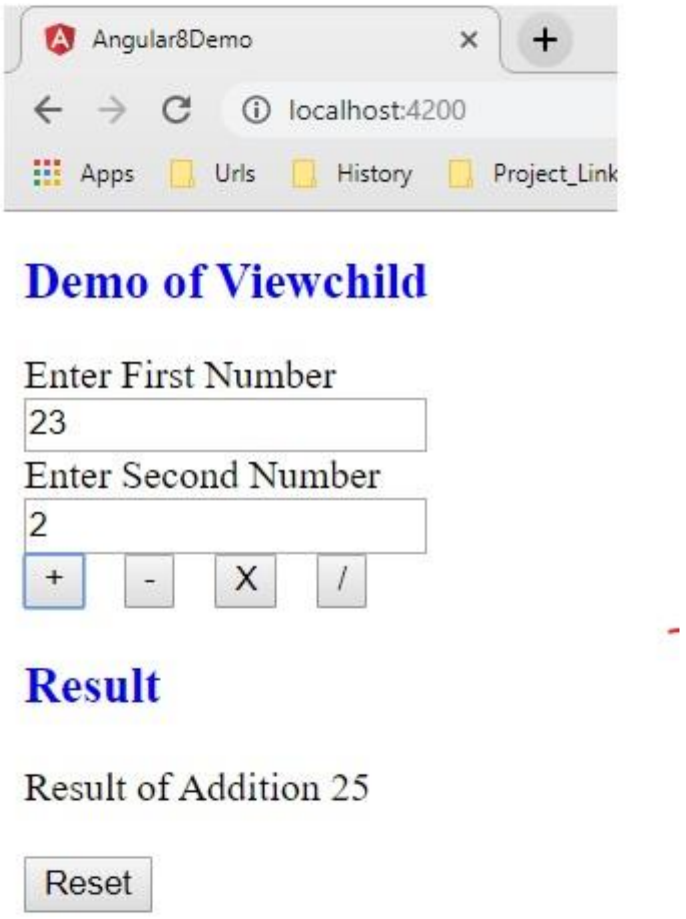
```typescript
import { Component,ViewChild } from '@angular/core';
import { ChildComponent } from './child.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls : ['./custom.css']
})
export class AppComponent {
    private result: string = '';

      @ViewChild(ChildComponent, { static:true}) private _calculator: ChildComponent;

      constructor() {
      }

      private add(value: number): void {
          this.result = 'Result of Addition ' + value;
      }

      private subtract(value: number): void {
          this.result = 'Result of Subtraction ' + value;
      }

      private multiply(value: number): void {
          this.result = 'Result of Multiply ' + value;
      }

      private divide(value: number): void {
          this.result = 'Result of Division ' + value;
      }

      private reset(): void {
          this.result = '';
          this._calculator.clear();
      }
 }
```

app.component.html

```html
<h2>Demo of Viewchild</h2>
<div>
    <child (addNumber)="add($event)" (subtractNumber)="subtract($event)" (multiplyNumber)="multiply($event)"
            (divideNumber)="divide($event)" #calculator></child>
```

```
</div>
<h3>Result</h3>
<span>{{result}}</span>
<br />
<br />
 <input type="button" value="Reset" (click)="reset()" />
```

Now, check the output in the browser –



Demo of Viewchild

Enter First Number

23

Enter Second Number

2

+ - X /

Result

Result of Addition 25

Reset

About Angular Forms

When we want to start developing any web-based application, we always find that a large number of UI or interfaces are very much form dependent. This is very much true for any enterprise-type application also, since most of the interfaces of that application are a large form which contains multiple tabs, dialogs, grid, etc. And also these types of forms always contain non-trivial business validation logic. Since in a component-based framework, we need to split a form into multiple small and reusable pieces of code. These components can be used across the entire application which basically provides many architectural benefits including flexibility and design changes.

Angular 8 deals with the forms of objects through ngModel. The instance two-way data binding technique of ng-model in the Angular framework is really a life-saver, because it provides automatic sync between the form and the view model objects. For using the forms module in Angular, we need to inject the FormModule in our application module.

Types of Angular Forms

As a modern full-fledged UI Framework, Angular has its own full-fledged libraries for developing complex form-based UI. The current Angular framework has two types of form-building strategies as,

1. Template Driven Form
2. Model Drive Form or Reactive Form

Both the above technology belongs to the @angular/forms packages and are totally based on the form-control classes. But in spite of that, both the techniques are different from each other in respect to their own philosophy, programming style, and technique.

In the below section, we will discuss in detail items related to the above two types of Angular Forms.

Template Driven Forms

In Angular Framework, Template-driven are those forms where we can write logic, validations, controls, etc. in the HTML template part of the component. Basically, the Template is totally responsible for setting the form elements in the UI, for implementing the validation using form control, etc. With the help of Template, we can also provide the Form Group within the HTML template. Template-driven forms are perfect for the simple scenario-based interface where we can easily use the two-way data binding of Angular. For Template-driven form, Angular provides some form-specific directives which we can use to bind the form input data without model variable. Due to this form-specific directive, we can add extra functionality and behavior to a plain HTML form. In the end, the Template itself takes care of binding the values with the model and the form validation.

Benefits of Template Driven Forms

In Angular Framework, we can obtain some benefit for using Template-Driven form as below,

- It is much easier to use
- This technique works perfectly in simple scenarios
- It totally depends on two-way data binding techniques i.e. ngModel syntax.
- It requires a minimum of code in the component part since most of the work is done in the HTML template part.

- It automatically tracks the form element and its control.
- Despite the above benefits, it has some drawbacks like –
- Template-driven form techniques fail when we want to design some complex form in the UI section
- We can’t perform any Unit Testing based on the Template Driven Form.

Model-Driven Forms

Model-Driven form technique is often called a Reactive Form technique in Angular. It provides a model-driven approach to the UI to handling form inputs towards the component variable. Using this technique, we can create and update a simple form of control, progress to using multiple controls within a form group to validate the form control values, etc.

Model-driven or Reactive forms normally use an explicit and immutable approach to manage the state of a form control at any given point of time. If any change occurrs in the form control, then it will return a new state through which Angular can maintain the integrity of the model variables between changes. Basically, reactive forms are normally based on observable streams. So, every form's input controls and values provide the input value as a stream which is accessed by the reactive form synchronously. With the help of reactive forms, we can perform a straightforward process to perform unit testing on the form controls. This can be done very easily since we can be assured that form data is consistent and predictable whenever requested.

Reactive forms or Model-driven forms are different compared to template-driven forms. Reactive forms always provide more predictability with synchronous access towards the data model variable, provide immutability with observables operators and also keep track of changes using observable streams.

Benefits of Model-Driven Forms

In Angular Framework, we can obtain some benefit for using Model-Driven form or Reactive Form techniques as below,

- In Reactive Forms, form definition including logic related coding mainly maintained within the TypeScript part of the component. Since using this technique, we create form controls programmatically using FormGroup or FormBuilder class. In HTML template, HTML form tags are only used to put a reference of TypeScript based form-control class.
- It provides us programmatic and full control of the form value updates and form validations.
- In this technique, we can create a dynamic structure-based form at run time.
- We can implement custom form validation.
- Since the entire form-based part is in typescript class or component, it is much easier to write unit tests in reactive forms.
- Despite the above benefits, it has some drawback like –
- This technique requires much more coding, especially in the TypeScript part.
- It is a little bit complex to understand and maintain the code.

Template-Driven Form vs Reactive Form

The comparison between Template-Driven Form and Reactive Forms (Model-Driven Forms) are as below,

Template-Driven Form	Reactive Form
Template-Driven Form is less explicit, and it is mainly created by Directives.	Reactive Form is more explicit and normally created within the Component class.
It supports the unstructured data model	It always supports the structured data model.
It uses directives for implementing Form validations	It uses the function for implementing Form Validations
When form control value changes, it provides an asynchronous mechanism to update form controls.	When form control value changes, it provides synchronous mechanism to update form controls.

Form Controls

FormControl class is mainly used to assign any form related fields in the Angular component. This class is also used in the FormBuilder class method. It is used mainly for ease of access. With the help of references of the FormControl in place of FormControl class, we can gain access to the inputs in the template without using the Form itself. Similarly, we can use any instance of FormControl to access its parent group by using its root property. When we define a form of control, it requires two properties: an initial value and a list of validators.

Reactive Form Validation

Angular Framework provides many validators to validate the form control input values in any application. These validations can be imported along with the related dependencies for procedural forms. In general, the common practice for using Form validations are using .valid and .untouched to determine if we need to raise an error message. As an inbuilt validator, we can use hasError() method on the form element to validate the data.

Reactive Form Custom Validation

As per the built-in validators, it is very useful if we can create our own custom validator for our purpose. Angular Framework always allows us to do just that, with minimal effort. A simple function takes the FormControl instance and returns null if everything is fine. If the test fails, it returns an object with an arbitrarily named property. In this case, we need to use the property name .hasError() for the test.

```
<div [hidden]="!password.hasError('hasSpecialChars')">
      Your password must have Special Characters like @,#,$, etc
!
</div>
```

Demo 1 - Template Driven Form

Now, in this demo, we will demonstrate how to define a template-driven form in Angular.

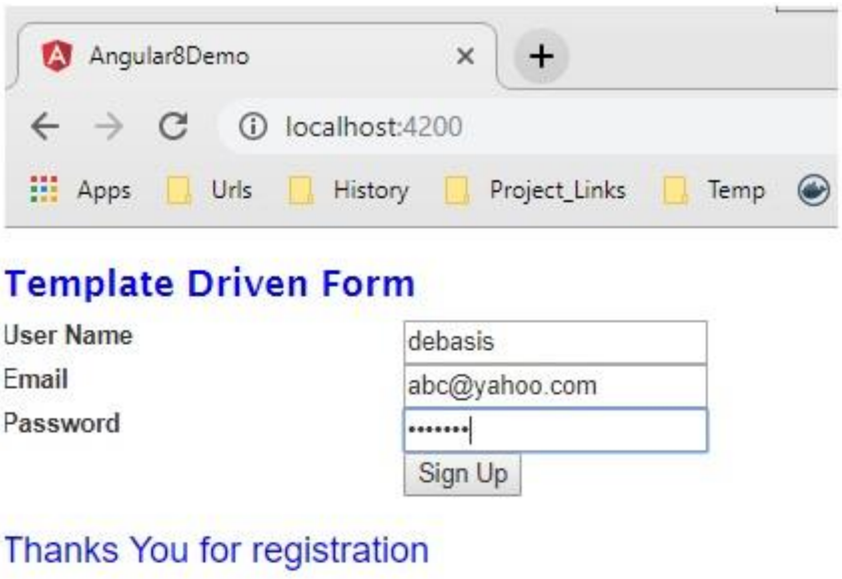
app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls : ['./custom.css']
})
export class AppComponent implements OnInit {
      private formData: any = {};
      private showMessage: boolean = false;

      constructor() {
      }

      ngOnInit(): void {
      }

      registerUser(formdata: NgForm) {
          this.formData = formdata.value;
          this.showMessage = true;
      }
 }
```

app.component.html

```
<h2>Template Driven Form</h2>
<div>
    <form #signupForm="ngForm" (ngSubmit)="registerUser(signupForm)">
        <table style="width:60%;" cellpadding="5" cellspacing="5">
            <tr>
                <td style="width :40%;">
                    <label for="username">User Name</label>
                </td>
                <td style="width :60%;">
                     <input type="text" name="username" id="username" [(ngModel)]="username" required>
                   </td>
               </tr>
               <tr>
                   <td style="width :40%;">
                       <label for="email">Email</label>
                   </td>
                   <td style="width :60%;">
                       <input type="text" name="email" id="email" [(ngModel)]="email" required>
                   </td>
               </tr>
               <tr>
                   <td style="width :40%;">
                       <label for="password">Password</label>
                   </td>
                   <td style="width :60%;">
                       <input type="password" name="password" id="password" [(ngModel)]="password" required>
                   </td>
               </tr>
               <tr>
                   <td style="width :40%;"></td>
                   <td style="width :60%;">
                       <button type="submit">Sign Up</button>
                   </td>
               </tr>
           </table>
       </form>
       <div *ngIf="showMessage">
           <h3>Thanks You {{formData.username}} for registration</h3>
       </div>
   </div>
```

Now check the output in the browser,



Template Driven Form

User Name	debasis
Email	abc@yahoo.com
Password	•••••••
	Sign Up

Thanks You for registration

Demo 2 - Model-Driven Form

Now in this demo, we will discuss how to develop a Reactive Form in Angular 8 Framework. For that, we first need to develop a login form component as below –

app.component.ts

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { Validators, FormBuilder, FormControl, FormGroup  } from
 '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls : ['./custom.css']
})
export class AppComponent implements OnInit {
   private formData: any = {};

   username = new FormControl('', [
       Validators.required,
       Validators.minLength(5)
   ]);

   password = new FormControl('', [
       Validators.required,
       hasExclamationMark
   ]);

   loginForm: FormGroup = this.builder.group({
       username: this.username,
       password: this.password
   });

   private showMessage: boolean = false;

   constructor(private builder: FormBuilder) {
   }

   ngOnInit(): void {
   }

   registerUser() {
       this.formData = this.loginForm.value;
       this.showMessage = true;
   }
 }

 function hasExclamationMark(input: FormControl) {
   const hasExclamation = input.value.indexOf('!') >= 0;
   return hasExclamation ? null : { needsExclamation: true };
 }
```

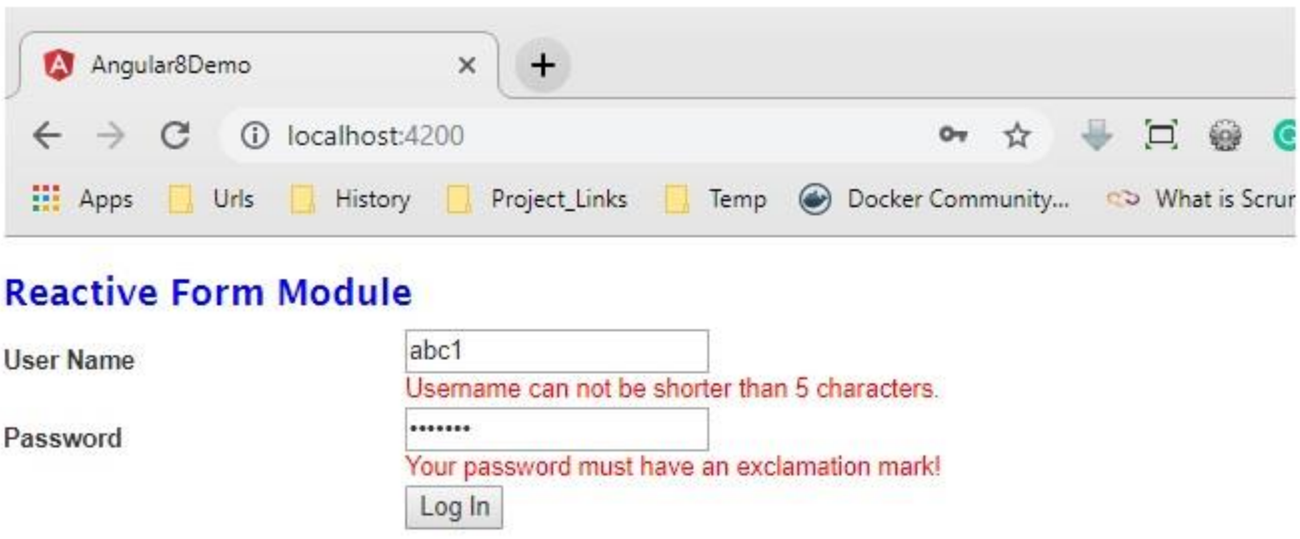
app.component.html

```
<h2>Reactive Form Module</h2>
<div>
    <form [formGroup]="loginForm" (ngSubmit)="registerUser()">
        <table style="width:60%;" cellpadding="5" cellspacing="5">
            <tr>
                <td style="width :40%;">
                    <label for="username">User Name</label>
                </td>
                <td style="width :60%;">
                    <input type="text" name="username" id="username" [formControl]="username">
                    <div [hidden]="username.valid || username.untouched" class="error">
                        <div [hidden]="!username.hasError('minlength')">
                            Username can not be shorter than 5 characters.
                        </div>
                        <div [hidden]="!username.hasError('required')">
                            Username is required.
                        </div>
                    </div>
                </td>
            </tr>
            <tr>
                <td style="width :40%;">
                    <label for="password">Password</label>
                </td>
                <td style="width :60%;">
                    <input type="password" name="password" id="password" [formControl]="password">
                    <div [hidden]="password.valid || password.untouched" class="error">
                        <div [hidden]="!password.hasError('required')">
                            The password is required.
                        </div>
                        <div [hidden]="!password.hasError('needsExclamation')">
                            Your password must have an exclamation mark!
                        </div>
                    </div>
                </td>
            </tr>
            <tr>
                <td style="width :40%;"></td>
                <td style="width :60%;">
                    <button type="submit" [disabled]="!loginForm.valid">Log In</button>
                </td>
            </tr>
        </table>
    </form>
    <div *ngIf="showMessage">
        <h3>Thanks You {{formData.username}} for registration</h3>
    </div>
</div>
```

Now, for using the reactive form we need to inject ReactiveFormModule in our app.module.ts file as below –

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule, NO_ERRORS_SCHEMA } from '@angular/core';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

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

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

Now check the output in the browser,



Reactive Form Module

User Name: abc1

Username can not be shorter than 5 characters.

Password: •••••••

Your password must have an exclamation mark!

Log In

What is Angular Service?

In Angular Framework, services are always singleton objects which normally get instantiated only once during the lifetime of any application or module. Every Angular Service contains several methods that always maintains the data throughout the life of an application. It is a mechanism to share responsibilities within one or multiple components. As per the Angular Framework, we can develop any application using a nested relationship-based component. Once our components are nested, we need to manipulate some data within the different components. In this case, a service is the best way to handle this. Service is the best place where we can take data from other sources or write down some calculations. Similarly, services can be shared between multiple components as needed.

Angular has greatly simplified the concept of services since Angular 1.x. In Angular 1x, there were services, factories, providers, delegates, values, etc., and it was not always clear when to use which one. So, for that reason, Angular 8 simply changed the concept of Angular. There are simply two steps for creating services in Angular:

1. Create a class with @Injectable decorator.
2. Register the class with the provider or inject the class by using dependency injection.

In Angular, a service is used when a common functionality or business logic needs to be provided, written, or needs to be shared in a different name. Actually, a service is a totally reusable object. Assuming that our Angular application contains some of the components performing the logging for error tracking purposes, you will end up with an error log method in each of these components. As per the standard practice, it is a bad approach since we used an error log method multiple times in the application. If you want to change the semantics of error logging, then you will need to change the code in all these components, which will impact the whole application. So use a common service component for the error log features is always good. In this way, we can remove the error log method from all components and placed that code within a service class. Then components can use the instance of that service class to invoke the method. In Angular Framework, Service injection is one way of performing dependency injection.

Benefits of using Service

Angular services are single objects that normally get instantiated only once during the lifetime of the Angular application. This Angular service maintains data throughout the life of an application. It means data does not get replaced or refreshed and is available all the time. The main objective of the Angular service is to use shared business logic, models, or data and functions with multiple different components of an Angular application.

The main objective of using an Angular service is the Separation of Concern. An Angular service is basically a stateless object, and we can define some useful functions within an Angular service. These functions can be invoked from any component of the application elements like Components, Directives, etc. This will help us to divide the entire application into multiple small, different, logical units so that those units can be reusable.

How to define a Service

We can create a user-defined custom service as per our requirement. To create a service, we need to follow the below steps:

1. First, we need to create a TypeScript file with proper naming.
2. Next, create a TypeScript class with the proper name that will represent the service after a while.
3. Use the @Injectable decorator at the beginning of the class name that was imported from the @angular/core packages. Basically, the purpose of the @Injectable is that user-defined service and any of its dependents can be automatically injected by the other components.
4. Although, for design readability, Angular recommends that you always define the @Injectable decorator whenever you create any service.
5. Now, use the Export keyword against the class objects so that this service can be injectable or reused on any other components.

NOTE
When we create our custom service available to the whole application through the use of a provider’s metadata, this provider’s metadata must be defined in the app.module.ts(main application module file) file. If you provide the service in the main module file, then it is visible to the whole application. If you provide it in any component, then only that component can use the service. By providing the service at the module level, Angular creates only one instance of the CustomService class, which can be used by all the components in an application.

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

@Injectable()
export class AlertService
{
  constructor()
  { }

  publish showAlert(message: string)
   {
     alert(message);
   }
}
```

@Injectable()

@Injectable is actually a decorator in Angular Framework. Decorators are a proposed extension in JavaScript. In short, a decorator provides the ability for programmers to modify or use methods, classes, properties, and parameters. In angular, every Injectable class actually behaves just like a normal class. That’s why the Injectable class does not have any special lifecycle in the Angular framework. So, when we create an object of an Injectable class, the constructor of that class simply executed just like ngOnInit() of the component class. But, in the case of Injectable class, there is no chance to define destructor because, in JavaScript, there is no concept of the destructor. So, in simple work, the Injectable service class can’t be the destroyer. If we want to remove the instance of the service class, then we need to remove the reference point of dependency injection related to that class.

@Injectable decorator indicates Angular Framework that this particular class can be used with the dependency injector. @Injectable decorator is not strictly required if the class has other Angular decorators like a Component decorator, directive decorator, etc. on it or does not have any dependencies.

Most important is that any class needs to define with @Injectable() decorator so that it can be injected into the application.

```
@Injectable()
export class SampleService
{
  constructor()
  {
    console.log('Sample service is created');
  }
}
```

Similar to the .NET MVC Framework, Angular 8 also provides support for the Dependency Injection or DI. We can use the component constructor to inject the instances of the service class. Angular 8 provides the provider metadata in both Module level and Component level which can be used to perform Automatic Dependency Injection of any Injectable Service at the runtime.

What is Dependency Injection?

Dependency Injection always is one of the main benefits of Angular Framework. Due to these benefits, Angular Framework is receiving much more appreciation and acceptation among developers which can not achieve by other related client-side frameworks. With the help of this feature, we can inject any types of dependency like service, external utility module in our application module. For doing this, we do not even want to know how those dependency modules or services have been developed.

So, Angular Framework has its own mechanism for the Dependency Injection system. In this system, every angular module has its related own injector metadata values. According to that, the injector of each module is responsible for creating the dependent object reference point and then it will inject in the module we required. Actually, every dependency behaves like key-pair values where token act as a key and the instance of the object which needs to be injected act as a value. But in spite of this cool mechanism, there are some problems in the existing Dependency mechanism as below -

- *Internal cache*
  In Angular, every dependency object created as a singleton object. So, when we inject any service class as a dependent object, then the instance of that service class created once for the entire application lifecycle.

- *Namespace collision*
  In Angular, we can't be injected two different service classes from two different modules with the same name. As an example, suppose we create a service called user service for our own module and we inject that service. Now, suppose we import an EmployeeModule which contains the same name service called UserService and we also want to inject that. Then that can't be done since the same name service already injected in the application.

- *Built into the framework*
  In Angular Framework, Dependency Injection is totally tightly coupled with Angular Modules. We can't decouple the Dependency Injection from the Angular module as a standalone system.

In Angular Framework, Dependency Injection contains three sections like Injector, Provider & Dependency.

1. *Injector*
   The main purpose of the using Injector section is the expose an object or APIs which basically helps us to create instances of the dependent class or services.

2. *Provider*
   A Provider is basically acting as an Instructor or Commander. It basically provides instruction to the injector about the process of creating instances of the dependent objects. The provider always taken the token value as input and then map that token value with the newly created instances of the class objects.

3. *Dependency*
   Dependency is the processor type that basically identifies the nature of the created objects.

So as per the above discussion, we can perform the below tasks using the Dependency Injection in Angular Framework,

- We can create instances of the service classes in the constructor level using the provider metadata.

- We can use providers to resolve the dependencies between module-level providers and component level providers.
- The dependency injection process creates the instances of the dependent class and provides the reference of those objects when we required in the code.
- All the instances of the dependency injected objects are created as a Singletone object.

What is Provider?

So now, one question arises after the above discussion: what are these providers that injectors register at each level? A provider is an object or class that Angular uses to provide something we want to use:

- A class provider always generates or provides an instance of a class
- A factory provider generates or provides whatever returns when we run a specified function
- A value provider does not need to take up action to provide the result, it just returns a value

Sample of a Class

```
export class testClass {
  public message: string = "Hello from Service Class";
  public count: number;
  constructor() {
    this.count=1;
  }
}
```

Okay, that's the class. Now, we need to use the Injectable() decorator so that Angular Framework can register that class a provider and we can use the instance of that class within the application. We'll create a component that will serve as the root component of our application. Adding the testClass provider to this component is straightforward:

- Import testClass
- Add it to the @Component() decorator property

Add an argument of type "testClass" to the constructor

```
import { Component } from "@angular/core";
import { testClass} from "./service/testClass";

@Component({
  module : module.id,
  selector : 'test-prog',
  template : '<h1>Hello {{_message}}</h1>',
  providers : [testClass]
})

 export class TestComponent
 {
   private _message:string="";
   constructor(private _testClass : testClass)
   {
     this._message = this._testClass.message;
   }
 }
```

Under the covers, when Angular instantiates the component, the DI system creates an injector for the component that registers the testClass provider. Angular then sees the testClass type specified in the constructor’s argument list, looks up the newly registered testClass provider, and uses it to generate an instance that it assigns to “_testClass”. The process of looking up the testClass provider and generating an instance to assign to “_testClass” is all Angular.

Inject a Service into a Module

We can inject the Angular Service in Application Level or Module Level. The provider's property of NgModule decorator gives us the ability to inject a list number of Angular services into the module level. It will provide a single instance of the Angular Service across the entire application and can share the same value within different components.

```
@NgModule({
  imports: [ BrowserModule, FormsModule ],
  declarations: [ AppComponent, ParentComponent, ChildComponent ],
  bootstrap: [ AppComponent ],
  providers: [ SimpleService, EmailService ] ①
  })
class AppModule { }
```

Inject a Service into a Component

We can also inject the Angular Service into the Component Level. Similarly like NgModule, providers' property of Component decorator gives us the ability to inject a list number of Angular services within that particular component. It will provide a single instance of the Angular Service across the entire component along with the child component.

```
@Component({
  selector: 'parent',
  template: `...`,
  providers: [ EmailService ]
  })
  class ParentComponent {
  constructor(private service: EmailService) { }
  }
```

Demo 1 - Basic Service

Now in this demo, we will demonstrate how to use Injectable Service in Angular. For this purpose, we will develop an entry form related to the Student Information and on submit button click, we will pass those data into service so that it will store that data.

app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { StudentService } from './app.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls : ['./custom.css']
})
export class AppComponent implements OnInit {
    private _model: any = {};
    private _source: Array<any>;

    constructor(private _service: StudentService) {
        this._source = this._service.returnStudentData();
    }

    ngOnInit(): void {
    }

    private submit(): void {
        if (this.validate()) {
            this._service.addStudentData(this._model);
            this.reset();
        }
    }

    private reset(): void {
        this._model = {};
    }

    private validate(): boolean {
        let status: boolean = true;
        if (typeof (this._model.name) === "undefined") {
            alert('Name is Blank');
            status = false;
            return;
        }
        else if (typeof (this._model.age) === "undefined") {
            alert('Age is Blank');
            status = false;
            return;
        }
```

```
      else if (typeof (this._model.city) === "undefined") {
          alert('City is Blank');
          status = false;
          return;
      }
      else if (typeof (this._model.dob) === "undefined") {
          alert('dob is Blank');
          status = false;
          return;
      }
      return status;
  }
}
```

app.component.html

```
<div style="padding-left: 20px">
    <h2>Student Form</h2>
    <table style="width:80%;">
        <tr>
            <td>Student Name</td>
            <td><input type="text" [(ngModel)]="_model.name" /></td>
        </tr>
        <tr>
            <td>Age</td>
             <td><input type="number" [(ngModel)]="_model.age" /></td>
         </tr>
         <tr>
             <td>City</td>
             <td><input type="text" [(ngModel)]="_model.city" /></td>
         </tr>
         <tr>
             <td>Student DOB</td>
             <td><input type="date" [(ngModel)]="_model.dob" /></td>
         </tr>
         <tr>
             <td></td>
             <td>
                 <input type="button" value="Submit" (click)="submit()" />
                 <input type="button" value="Reset" (click)="reset()" />
             </td>
         </tr>
     </table>
     <h3>Student Details</h3>
     <div class="ibox-content">
         <div class="ibox-table">
             <div class="table-responsive">
                 <table class="responsive-table table-striped table-bordered table-hover">
                     <thead>
                         <tr>
                             <th style="width:40%;">
                                 <span>Student's Name</span>
                             </th>
                             <th style="width:15%;">
                                 <span>Age</span>
                             </th>
                             <th style="width:25%;">
                                 <span>City</span>
                             </th>
                             <th style="width:20%;">
                                 <span>Date of Birth</span>
                             </th>
                         </tr>
                     </thead>
                     <tbody>
                         <tr *ngFor="let item of _source; let i =index">
                            <td><span>{{item.name}}</span></td>
                             <td><span>{{item.age}}</span></td>
                             <td><span>{{item.city}}</span></td>
                             <td><span>{{item.dob}}</span></td>
                         </tr>
                     </tbody>
                 </table>
             </div>
         </div>
     </div>
 </div>
```

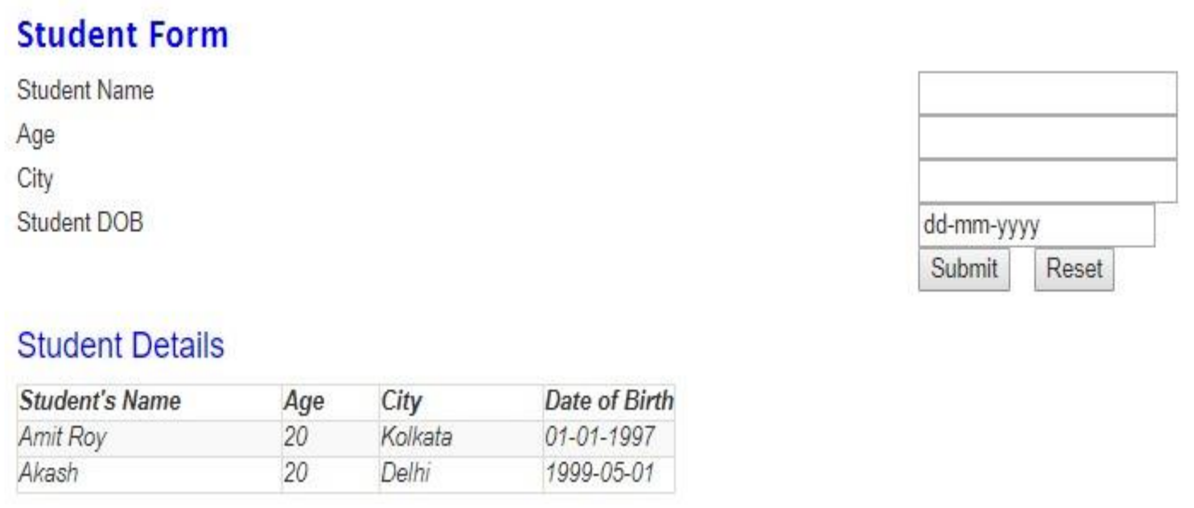
app.service.ts

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

@Injectable()
export class StudentService {
    private _studentList: Array<any> = [];

    constructor() {
        this._studentList = [{name:'Amit Roy', age:20, city:'Kolkata', dob:'01-01-1997'}];
    }

     returnStudentData(): Array<any> {
         return this._studentList;
     }

     addStudentData(item: any): void {
         this._studentList.push(item);
     }
}
```

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule, NO_ERRORS_SCHEMA } from '@angular/core';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

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

@NgModule({
  declarations: [
     AppComponent
   ],
   imports: [
     BrowserModule, FormsModule, ReactiveFormsModule
   ],
   providers: [StudentService],
   bootstrap: [AppComponent],
   schemas: [NO_ERRORS_SCHEMA]
 })
 export class AppModule { }
```

Now check the browser for the output,



Student's Name	Age	City	Date of Birth
Amit Roy	20	Kolkata	01-01-1997
Akash	20	Delhi	1999-05-01

Demo 2 - Inject Service into a Module

Now, in this demo, we will discuss the module level injection of any Angular service. For that purpose, we will create two-component as a parent child-related i.e. Parent Component and Child Component. And then use the selector of the parent component in our Root component. For showing the module level service instance, we use two instances of Parent component within the Root component. The code samples as below,

parent.component.ts

```
import { Component, OnInit } from '@angular/core';
import { DemoService } from './app.service';

@Component({
  selector: 'parent',
  templateUrl: './parent.component.html',
  styleUrls : ['./custom.css']
})
export class ParentComponent implements OnInit {

     constructor(private demoService:DemoService){

     }

     ngOnInit(){
     }
 }
```

parent.component.html

```
<div class="parent">
    <p>Parent Component</p>
```

```html
    <div class="form-group">
        <input type="text" class="form-control" name="value" [(ngModel)]="demoService.message">
    </div>
    <child></child>
</div>
```

child.component.ts

```typescript
import { Component, OnInit } from '@angular/core';
import { DemoService } from './app.service';

@Component({
  selector: 'child',
  templateUrl: './child.component.html',
  styleUrls : ['./custom.css']
})
export class ChildComponent implements OnInit {
     constructor(private demoService:DemoService){
     }

     ngOnInit(){
     }
 }
```

child.component.html

```html
<div class="child">
    <p>Child Component</p>
    {{ demoService.message }}
</div>
```

app.component.ts

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

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

   ngOnInit(){

   }
 }
```

app.component.html

```
<div style="padding-left: 20px;padding-top: 20px; width: 500px;">
    <div class="row">
        <div class="col-xs-6">
            <h2>Parent Componet - 1</h2>
            <parent></parent>
        </div>
        <div class="col-xs-6">
            <h2>Parent Componet - 2</h2>
            <parent></parent>
         </div>
     </div>
 </div>
```

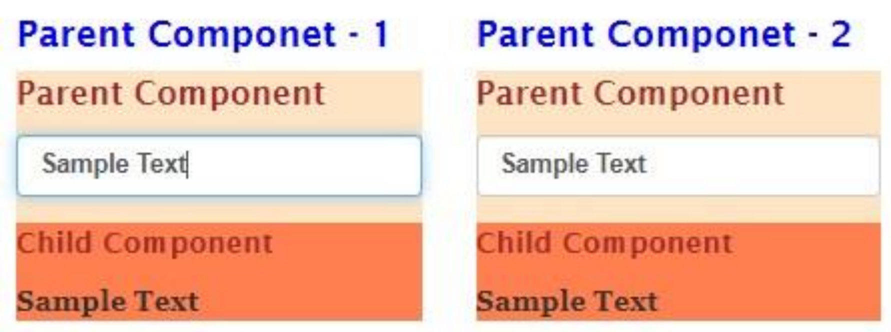
custom.css

```
.parent{
    background-color: bisque;
    font-family: Arial, Helvetica, sans-serif;
    font-weight: bolder;
    font-size: large;
}

.child{
    background-color:coral;
     font-family: Georgia, 'Times New Roman', Times, serif;
     font-weight: bold;
     font-size: medium;
 }
```

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule, NO_ERRORS_SCHEMA } from '@angular/core';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { ParentComponent } from './parent.component';
import { ChildComponent } from './child.component';
import { DemoService } from './app.service';

 @NgModule({
   declarations: [
     AppComponent,ParentComponent,ChildComponent
   ],
   imports: [
     BrowserModule, FormsModule, ReactiveFormsModule
   ],
   providers: [DemoService],
   bootstrap: [AppComponent],
   schemas: [NO_ERRORS_SCHEMA]
 })
 export class AppModule { }
```

Now check the output into the browser,



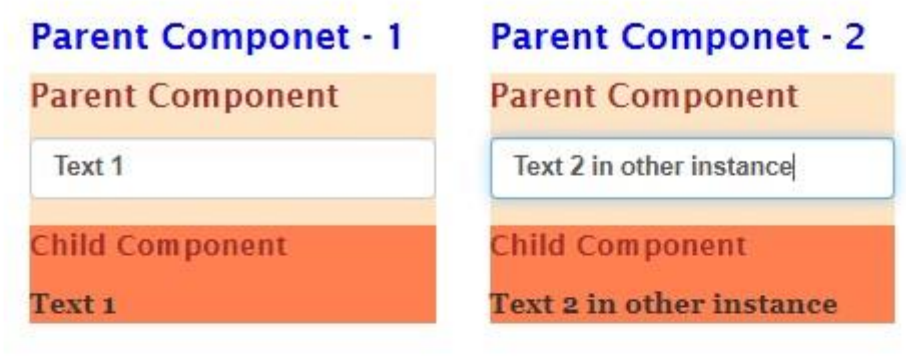
In the above example, we see that if we input some text in any one parent component input box, it will automatically update the related nested child component along with another parent component also. This is occurred because of the DemoService is injected into the Module level. So it creates a single-tone instance of that service and that instance is available across the entire application.

Demo 3 - Inject Service into a Component

In the previous sample, we see how to inject any Angular service into Module level and how if share the same data entered into one component to all components across the module. Now, in this demo, we will demonstrate what will happen if we inject any service into the Component level. For that purpose, we just make the below changes in the parent.component.ts file,

```
import { Component, OnInit } from '@angular/core';
import { DemoService } from './app.service';

@Component({
  selector: 'parent',
  templateUrl: './parent.component.html',
  styleUrls : ['./custom.css'],
  providers:[DemoService]
})
export class ParentComponent implements OnInit {
     constructor(private demoService:DemoService){
     }

     ngOnInit(){
     }
 }
```

Now, check the browser for the output,



In the above example, we clearly see that when we type any input in the parent 1 component, it will not automatically be shared by another instance of the parent component. Due is occurred due to the Component level injection of the DemoService.

Basic Idea of Ajax Call

Ajax stands for Asynchronous JavaScript and XML. Actually, Ajax is not any programming language or framework or tool. Ajax is basically is a concept to client-side script that communicates between the server and the client machine to perform any type of operations. The best definition as per Ajax in the overview of SPA based application will be “Ajax is a method to exchange or pull data from a server ”. In a basic concept, Ajax always refers to the use of XmlHttpRequest objects to interact with a web server dynamically via JavaScript library. There are several benefits we can achieve by using Ajax in any web-based application like –

1. Ajax is used to perform callback operations, which make a perfect quick round trip to and from the server to send or retrieve data without post the entire page to the server. In this way, we can minimize the network utilization and application performance can be a boost.
2. Ajax always allows us to make asynchronous calls to a web server. In this way, the client browser avoids waiting for all data to arrive before allowing the user to act once more.
3. Since Ajax call, complete page postback is not required, so any Ajax-enabled web application will always be more responsive and user-friendly.
4. With the help of Ajax, we can improve the speed, performance, and usability of a web application.

About HttpClient

In Angular 6, Google first introduced a new easier way to operate with http requests with the new library called HttpClient. Angular provides the new library name so that it can avoid the new breaking changes made with the existing Http library. With the help of this new Http library, we can take advantage of the new functionality like the ability to listen for progress events, interceptors monitor to modify requests or response. Actually, the HTTP module always uses RxJS Observable-based API internally. For this reason, when we pass multiple ajax calls to the server using the HTTP module, it will return all responses as an observable object which we need to subscribe to one way or another. When we working using observable objects we need to remember some key points like,

- To receive the value as a response against any HTTP calls we need to subscribe to the observables. If we forget to subscribe, then nothing will happen.
- If we subscribe multiples times to the observables, then multiple HTTP requests will be triggered.
- By nature or type, every observable are single-value streams. So, if the HTTP request is successful then these observables will emit only one value and then complete.
- If the HTTP request fails, then these Observables emit an error.

To use the HttpClient in an Angular application, we need to import the HttpClientModule module into the root module of the application to use the Http Service.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule, NO_ERRORS_SCHEMA } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';

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

@NgModule({
  declarations: [
    AppComponent
   ],
   imports: [
     BrowserModule,HttpClientModule
   ],
   bootstrap: [AppComponent],
   schemas: [NO_ERRORS_SCHEMA]
 })
 export class AppModule { }
```

What are Observables?

Observable is one of the existing features used in the Angular Framework. But basically, it is not the Angular specific features, rather it is a proposed standard for managing any type of async data or operation which will be considered in the ES7 version. With the help of Observables, we can perform the continuous operations for multiple communication through which multiple values of data can be passed. So, by using Observables we can deal with array-like data operations to parse, modify and maintain data. That’s why the Angular framework use observables extensively.

Observables are actually a new primitive type that basically acts as a blueprint for how we want to create streams, subscribe to them, react to new values and combine streams together to build new streams.

What are Promises?

In our application, when we execute any task synchronously, we wait for it to finish before moves to the next line of code. But if we execute the same task asynchronously, the program moves to the next line of code before the task finished its job, maybe finishing off that task may require a few seconds or more. So in a simple word, we can assume that synchronous programming like waiting in line or queue and asynchronously programming like taking a ticket and don't wait in a queue. Since we have a ticket we can do some other tasks and when that particular task completed it will notify us against that ticket. One of the easiest ways to program asynchronously is to use callbacks functions. We can pass to an asynchronous function a function as a parameter which will be invoked when the task is completed.

```
function doAsyncTask(cb) {
    setTimeout(() => {
    console.log("Async Task Calling By Callback Function");
    cb();
  }, 1000);
}

doAsyncTask(() => console.log("Callback Function Called"));
```

In the above example, doAsyncTask function will be fired an asynchronous task and returns immediately. When the async task completes, a function will be called. It is a callback function, as initialize by short name cb, because of it calls-you-back.

ES6 provides us an alternative mechanism built into the language called a promise. So, the promise is a placeholder for future value. It basically serves the same functionality as callbacks but it provides a nice and organized syntax which provides us an easier way to handle errors. We can create an instance of a promise by called new on the Promise class like below,

```
var promise = new Promise((resolve, reject) => {
});
```

We always pass an inner function with two arguments (resolve, reject). As we define the function, we can call or named this argument whenever we want, but the standard convention is to call them as resolve and reject. Actually, resolve and reject are in fact functions themselves. Within this inner function, we actually perform our asynchronous processing and then when we are ready we can call the resolve() as below,

```
var promise = new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Async Work Complete");
      resolve();
  }, 1000);
});
```

So, if we want to use the promise in our callback function example, it will be like this,

```
let error = true;
function doAsyncTask() {
  var promise = new Promise((resolve, reject) => {
    setTimeout(() => {
    console.log("Async Work Complete");
    if (error) {
      reject();
    }
    else
      {
        resolve();
      }
      }, 1000);
   });
   return promise;
 }
```

Observables vs Promises

When we use HttpClient for the Ajax calls in Angular Framework, we can use both implementations (i.e. observables or promises) which provides an easy way of API operation for handling requests or receiving a response. But still, there are some key differences that make Observables a superior alternative compared to the promises,

- Promises can only accept one value at a time unless we compose multiple promises execution plan (Eg: $q.all).
- Promises can't be canceled if required.

In Angular 8, it mainly introduces reactive programming concepts totally based on the observables for dealing with the asynchronous processing of data. In earlier versions of Angular (i.e. Angular 1.x), we normally use promises to handle the asynchronous processing related to the ajax call. But still, in Angular 8, we can still use the promises for the same purpose in place of observables. The main objective of reactive programming is the observable element which mainly used to the model or variable which can be observed. Basically, at the normal concept, promises and observables both seem to very similar to each other. Both observables and promises are used to execute asynchronous ajax call along with callback functions for both successful and error responses and also notify or inform us when the result is there.

How to Define Observables

So before we start discussing HTTP verbs, first we need to understand how we can define observables objects. So, to create an observable, we need to use the create method of the Observable object. We need to provide a function object as a parameter in which we need to initialize the observable processing. This function can return a function to cancel the observable.

```
var observable = Observable.create((observer) => {
  setTimeout(() => {
    observer.next('some event');
  }, 500);
});
```

Just like promises, observables can also provide multiple notifications with the help of different methods as below,

- next – It emits an event and it can be executed several times.
- error – It mainly used to throw an error. Once it is executed the stream will be a break. It means error callback will be called immediately and no more events or steps can be executed.
- complete – It is used to indicate that the observable as completed. After these events, no events will be executed.

With the help of observables, we can register the callback for previously described notifications. The subscribe method tackles this issue. It can accept three callback function as parameters,

- onNext – This callback will be called when an event is triggered.
- onError – This callback will be called when an error is thrown.
- onCompleted – This callback will be called when an observable complete.

Http Verbs

In the HTTP protocol, a set of verbs has been described for each type of operation or URL. For any URL, we can perform a GET, PUT, POST, DELETE, HEAD, and OPTIONS request. In this section, we will discuss how to execute the API URL for the mentioned verbs. For that purpose, we assume that we perform the dependency injection for HttpClient in the constructor level of the component as below,

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

GET

To perform a GET request, we need to simply call the get function on our http client. This will returns an observable which needs to subscribe to receive the data or response from the server-side.

```
performGET() {
    console.log("GET");
    let url = `http://localhost/get`;
    this.http.get(url).subscribe(res => console.log(res.text()))
;
}
```

DELETE

To Perform a DELETE request we just need to call the delete function of the http client. The format or the function is quite similar to the get function as above. But we can pass query params within this function.

```
performDELETE() {
    console.log("DELETE");
    let url = `http://localhost/delete`;
    let search = new URLSearchParams();
    search.set('foo', 'moo');
    search.set('limit', 25);
    this.http.delete(url, {search}).subscribe(res => console.log
(res.json()));
}
```

POST

To perform a POST request, we call the post function of the http client. The format of the post function is the same as getting a function. But we normally perform the POST request to pass the data as well as to the server. So the in the second parameter in case of post, need to pass an object in place of query parameters which will be passed as the payload for the request.

```
performPOST() {
    console.log("POST");
    let url = `http://localhost/post`;
    this.http.post(url, {moo:"foo",goo:"loo"}).subscribe(res =>

    console.log(res.json()));
}
```

PUT

To perform a PUT request we just need to call the put function. It will work exactly the same as the post function.

```
performPUT() {
    console.log("PUT");
    let url = `http://localhost/put`;
    let search = new URLSearchParams();
    search.set('foo', 'moo');
    search.set('limit', 25);
    this.http.put(url, {moo:"foo",goo:"loo"}, {search}).subscrib
e(res =>
    console.log(res.json()));
}
```

Handling Errors

When we perform an ajax call from our Angular application,an error or exceptions on the server-side may occur. In that case, it will return that error or exception message to the angular application. So we need to know how we can receive that exception message so that we can display those messages in the front of the user or store in our error logs. So, whether we are handling the response as an Observable or as a Promise, we can always implement error handling function as the second param of the http method. In case of a promise, it looks like as below,

```
performGETAsPromiseError() {
    console.log("GET AS PROMISE ERROR");
    let url = `http://localhost/post`;
    this.http.get(url)
    .toPromise()
    .then(
      res => console.log(res.json()),
      msg => console.error(`Error: ${msg.status} ${msg.statusText}`) ①
    );
 }
```

In the case of observables, the same will look the below,

```
performGETAsError() {
    console.log("GET AS OBSERVABLES ERROR");
    let url = `http://localhost/post`;
    this.http.get(url).subscribe(
      res => console.log(res.json()),
      msg => console.error(`Error: ${msg.status} ${msg.statusText}`)
    );
}
```

Headers

In Ajax call, another important part is HTTP headers through which we can pass several configuration-related information to the server-side. By nature, HTTP headers are types of meta-data which normally attached by the browser when we call any HTTP request. Sometimes we need to pass some extra customer headers with our ajax request and we can do that very easily with the help of Angular http client. For this purpose, we first need to import two helper classes from the http module in our component or service.

```
import { HttpClient, HttpResponse, HttpHeaders } from '@angular/common/http';
```

We first need to create an instance of the HttpHeaders and then we can define our mentioned headers in that instance as below –

```
const httpOptions = {
        headers: new HttpHeaders({
          'Content-Type': 'application/json; charset=utf-8'
        })
      };
```

After it, we just need to pass the httpHeader to the proper HTTP verb function.

```
performPOST() {
    console.log("POST");
    let url = `http://localhost/post`;
    this.http.post(url, {moo:"foo",goo:"loo"},httpOptions).subscribe(res =>
    console.log(res.json()));
  }
```

Demo 1 - Http Core API Samples

Now in this demo, we will demonstrate how to use ajax call in Angular. For this purpose, we will develop a complete flow related to the employee where we can perform employee add, display existing records, edit data or delete employee data. In this sample demo, we create three different components as below –

- EmployeeListComponent – For display list employee of information
- AddEmployeeComponent – For Add New Employee Details
- UpdateEmployeeComponent – For update existing Employee Details

app.component.employeelist.ts

```
import { Component, OnInit, ViewChild, AfterViewInit } from '@angular/core';
import { HttpClient, HttpResponse, HttpHeaders } from '@angular/common/http';

@Component({
  selector: 'employee-list',
  templateUrl: 'app.component.employeelist.html'
})

export class EmployeeListComponent implements OnInit {

  public data: any = [];
  public showDetails: boolean = true;
  public showEmployee: boolean = false;
  public editEmployee: boolean = false;
  private _selectedData: any;
  private _deletedData: any;

  constructor(private http: HttpClient) {
  }

  ngOnInit(): void {

  }

  ngAfterViewInit(): void {
    this.loadData();
  }

  private loadData(): void {
    let self = this;
    this.http.get("http://localhost:81/SampleAPI/employee/getemployee")
      .subscribe((res: Response) => {
        self.data = res;
      });
  }

  private addEmployee(): void {
    this.showDetails = false;
    this.showEmployee = true;
  }

  private onHide(args: boolean): void {
    this.showDetails = !args;
    this.showEmployee = args;
    this.editEmployee = args;
    this.loadData();
  }

  private onUpdateData(item: any): void {
    this._selectedData = item;
    this._selectedData.DOB = new Date(this._selectedData.DOB);
    this._selectedData.DOJ = new Date(this._selectedData.DOJ);
    this.showDetails = false;
    this.editEmployee = true;
  }

  private onDeleteData(item: any): void {
    this._deletedData = item;
    if (confirm("Do you want to Delete Record Permanently?")) {
      let self = this;
      const httpOptions = {
        headers: new HttpHeaders({
          'Content-Type': 'application/json; charset=utf-8'
        })
      };
      this.http.post("http://localhost:81/SampleAPI/employee/DeleteEmployee", this._deletedData, httpOptions)
          .subscribe((res: Response) => {
              self.loadData();
          });
    }
  }
}
```

app.component.employeelist.html

```html
<div class="panel">
  <h3>HTTP Module Sample - Add and Fetch Data</h3>
  <div class="panel-body container-fluid" *ngIf="showDetails">
    <div class="row row-lg">
      <table class="table" style="width:100%;">
        <thead>
          <tr>
            <th class="cell-100">Srl No</th>
            <th class="cell-300">Alias</th>
            <th class="cell-300">Employee Name</th>
            <th class="cell-300">Date of Birth</th>
            <th class="cell-300">Join Date</th>
            <th class="cell-300">Department</th>
            <th class="cell-300">Designation</th>
            <th class="cell-300">Salary</th>
            <th class="cell-180"></th>
          </tr>
        </thead>
        <tbody>
          <tr *ngFor="let item of data">
            <td class="cell-100">{{item.Id}}</td>
            <td class="cell-300">{{item.Code}}</td>
            <td class="cell-300">{{item.Name}}</td>
            <td class="cell-300">{{item.DOB | date :'shortDate'}}</td>
            <td class="cell-300">{{item.DOJ | date :'mediumDate'}}</td>
            <td class="cell-300">{{item.Department}}</td>
            <td class="cell-300">{{item.Designation}}</td>
            <td class="cell-300">{{item.Salary |currency:'INR':true}}</td>
            <td class="cell-180">
              <a (click)="onUpdateData(item);" style="cursor:pointer;">
                <span class="badge badge-primary">Edit</span>
              </a>
              <a (click)="onDeleteData(item);" style="cursor:pointer;">
                <span class="badge badge-danger">Delete</span>
              </a>
            </td>
          </tr>
        </tbody>
      </table>
      <p>
```

```html
        <button class="btn btn-primary" (click)="addEmployee()">
          Add Employee
        </button>
      </p>
    </div>
  </div>
  <div  class="panel-body container-fluid" *ngIf="showEmployee">
    <employee-add (onHide)="onHide($event);"></employee-add>
  </div>
  <div  class="panel-body container-fluid" *ngIf="editEmployee">
    <employee-update [source]="_selectedData" (onHide)="onHide($event);"></employee-update>
  </div>
</div>
```

app.component.employeeadd.ts

```typescript
import { Component, OnInit, EventEmitter, Output } from '@angular/core';
import { HttpClient, HttpResponse, HttpHeaders } from '@angular/common/http';

@Component({
  selector: 'employee-add',
  templateUrl: 'app.component.employeeadd.html'
})

export class AddEmployeeComponent implements OnInit {

  public _model: any = {};
  @Output() private onHide: EventEmitter<boolean> = new EventEmitter<boolean>();

  constructor(private http: HttpClient) {
  }

  ngOnInit(): void {

  }

  public onCancel(): void {
    this._model = {};
    this.onHide.emit(false);
  }

  public submit(): void {
    if (this.validate()) {
      let self = this;
      const httpOptions = {
        headers: new HttpHeaders({
          'Content-Type': 'application/json; charset=utf-8'
        })
      };
      this.http.post("http://localhost:81/SampleAPI/employee/AddEmployee", this._model, httpOptions)
        .subscribe((res: Response) => {
          self.onCancel();
        });

    }
  }

  private reset(): void {
    this._model = {};
  }

  private validate(): boolean {
    let status: boolean = true;
    if (typeof (this._model.code) === "undefined") {
      alert('Alias is Blank');
      status = false;
      return;
    }
    else if (typeof (this._model.name) === "undefined") {
      alert('Name is Blank');
      status = false;
      return;
    }
    else if (typeof (this._model.dob) === "undefined") {
      alert('dob is Blank');
      status = false;
      return;
    }
    else if (typeof (this._model.doj) === "undefined") {
      alert('DOJ is Blank');
      status = false;
      return;
    }
    else if (typeof (this._model.department) === "undefined") {
      alert('Department is Blank');
      status = false;
      return;
    }
    else if (typeof (this._model.designation) === "undefined") {
      alert('Designation is Blank');
      status = false;
      return;
    }
    else if (typeof (this._model.salary) === "undefined") {
      alert('Salary is Blank');
      status = false;
      return;
    }
    return status;
  }
}
```

app.component.employeeadd.html

```
<div class="row row-lg">
  <h4>Provide Employee Details</h4>
  <table class="table">
    <tr>
      <td>Employee Code</td>
      <td><input type="text" [(ngModel)]="_model.code" /></td>
    </tr>
    <tr>
      <td>Employee Name</td>
      <td><input type="text" [(ngModel)]="_model.name" /></td>
    </tr>
    <tr>
      <td>Date of Birth</td>
      <td><input type="date" [(ngModel)]="_model.dob" /></td>
    </tr>
    <tr>
      <td>Date of Join</td>
      <td><input type="date" [(ngModel)]="_model.doj" /></td>
    </tr>
    <tr>
      <td>Department</td>
      <td><input type="text" [(ngModel)]="_model.department" /></td>
    </tr>
    <tr>
      <td>Designation</td>
      <td><input type="text" [(ngModel)]="_model.designation" /></td>
    </tr>
    <tr>
      <td>Salary</td>
      <td><input type="number" [(ngModel)]="_model.salary" /></td>
    </tr>
    <tr>
      <td></td>
      <td>
        <input type="button" value="Submit" (click)="submit()" />

        <input type="button" value="Cancel" (click)="onCancel()" />
      </td>
    </tr>
  </table>
</div>
```

app.component.employeeupdate.ts

```
import { Component, OnInit, EventEmitter, Output, Input } from '@angular/core';
import { HttpClient, HttpResponse, HttpHeaders } from '@angular/common/http';

@Component({
  selector: 'employee-update',
  templateUrl: 'app.component.employeeupdate.html'
})

export class UpdateEmployeeComponent implements OnInit {

  public _model: any = {};

  @Input()
  set source(data: any) {
    this._model = data;
  }
  get source() {
    return this._model;
  }

  @Output() private onHide: EventEmitter<boolean> = new EventEmitter<boolean>();

  constructor(private http: HttpClient) {
  }

  ngOnInit(): void {
    if (this._model == undefined) {
      this._model = this.source;
    }
  }

  public onCancel(): void {
    this._model = {};
    this.onHide.emit(false);
  }

  public onUpdate(): void {
    if (this.validate()) {
      let self = this;
      const httpOptions = {
        headers: new HttpHeaders({
          'Content-Type': 'application/json; charset=utf-8'
        })
      };
      this.http.put("http://localhost:81/SampleAPI/employee/UpdateEmployee", this._model, httpOptions)
        .subscribe((res: Response) => {
          self.onCancel();
        });

    }
  }

  private reset(): void {
    this._model = {};
  }

  private validate(): boolean {
    let status: boolean = true;
    if (typeof (this._model.Code) === "undefined") {
      alert('Alias is Blank');
      status = false;
      return;
    }
    else if (typeof (this._model.Name) === "undefined") {
      alert('Name is Blank');
      status = false;
      return;
    }
    else if (typeof (this._model.Department) === "undefined") {
      alert('Department is Blank');
      status = false;
      return;
    }
    else if (typeof (this._model.Designation) === "undefined") {
      alert('Designation is Blank');
      status = false;
      return;
    }
    else if (typeof (this._model.Salary) === "undefined") {
      alert('Salary is Blank');
      status = false;
      return;
    }
    return status;
  }

  private parseDate(dateString: string): Date {
    if (dateString) {
      return new Date(dateString);
    } else {
      return null;
    }
  }
}
```

app.component.employeeupdate.html

```
<div class="row row-lg">
  <h4>Provide Employee Details</h4>
  <table class="table">
    <tr>
      <td>Employee Code</td>
      <td><input type="text" [(ngModel)]="_model.Code" /></td>
    </tr>
    <tr>
      <td>Employee Name</td>
      <td><input type="text" [(ngModel)]="_model.Name" /></td>
    </tr>
    <tr>
      <td>Date of Birth</td>
      <td><input type="text" [(ngModel)]="_model.DOB" readonly /></td>
    </tr>
    <tr>
      <td>Date of Join</td>
      <td><input type="text" [(ngModel)]="_model.DOJ" readonly /></td>
    </tr>
    <tr>
      <td>Department</td>
      <td><input type="text" [(ngModel)]="_model.Department" /></td>
    </tr>
    <tr>
      <td>Designation</td>
      <td><input type="text" [(ngModel)]="_model.Designation" /></td>
    </tr>
    <tr>
      <td>Salary</td>
      <td><input type="number" [(ngModel)]="_model.Salary" /></td>
    </tr>
    <tr>
      <td></td>
      <td>
        <input type="button" value="Update" (click)="onUpdate()" />

        <input type="button" value="Cancel" (click)="onCancel()" />
      </td>
    </tr>
  </table>
</div>
```

app.component.ts

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

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

  ngOnInit(){
  }
}
```

app.component.html

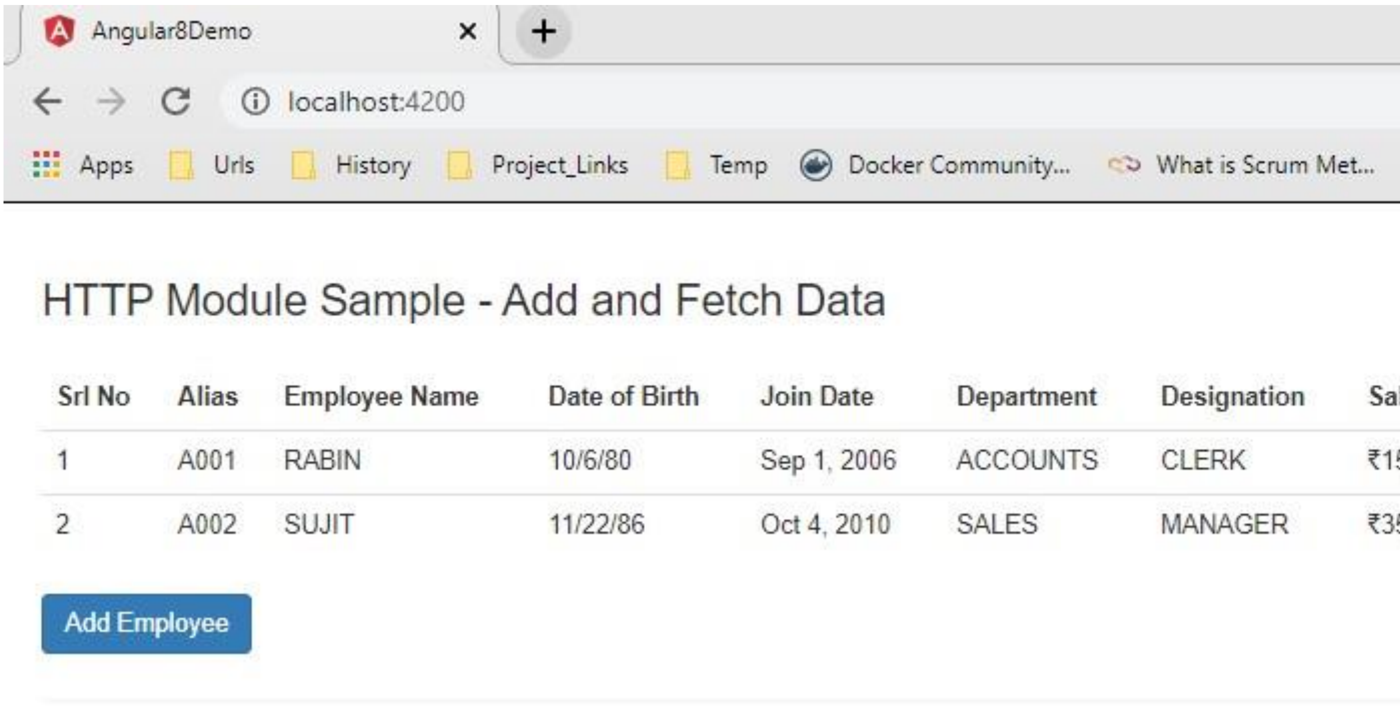
```
<div style="padding-left: 20px;padding-top: 20px; width: 1000px;">
    <div class="row">
        <div class="col-xs-12">
            <employee-list></employee-list>
        </div>
    </div>
</div>
```

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule, NO_ERRORS_SCHEMA } from '@angular/core';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';

import { AppComponent } from './app.component';
import { EmployeeListComponent } from './app.component.employeelist';
import { AddEmployeeComponent } from './app.component.employeeadd';
import { UpdateEmployeeComponent  } from './app.component.employeeupdate';

@NgModule({
  declarations: [
    AppComponent,EmployeeListComponent,AddEmployeeComponent, UpdateEmployeeComponent
  ],
  imports: [
    BrowserModule,HttpClientModule, FormsModule, ReactiveFormsModule
  ],
  bootstrap: [AppComponent],
  schemas: [NO_ERRORS_SCHEMA]
})
export class AppModule { }
```

Now run the demo in the browser,

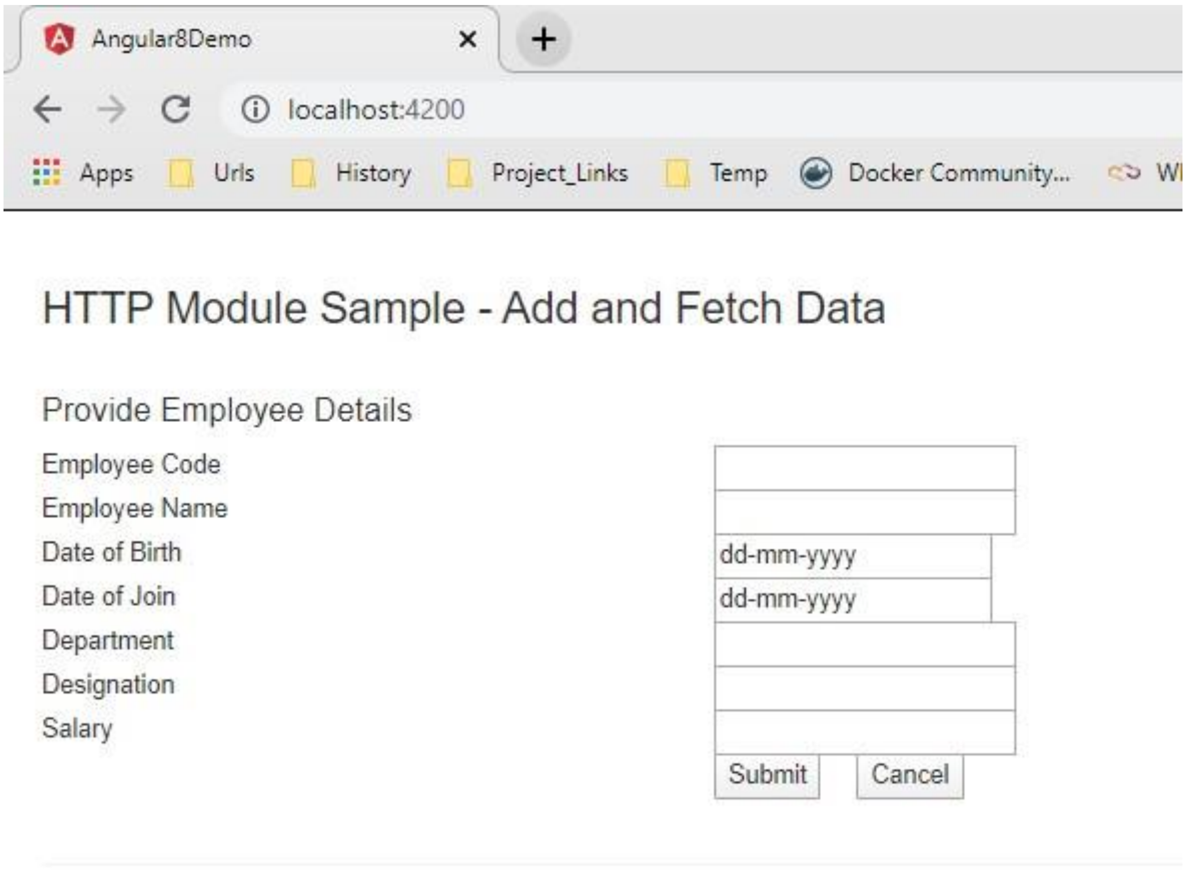
Employee List



HTTP Module Sample - Add and Fetch Data

Srl No	Alias	Employee Name	Date of Birth	Join Date	Department	Designation	Sa
1	A001	RABIN	10/6/80	Sep 1, 2006	ACCOUNTS	CLERK	₹1
2	A002	SUJIT	11/22/86	Oct 4, 2010	SALES	MANAGER	₹3

New Employee Add

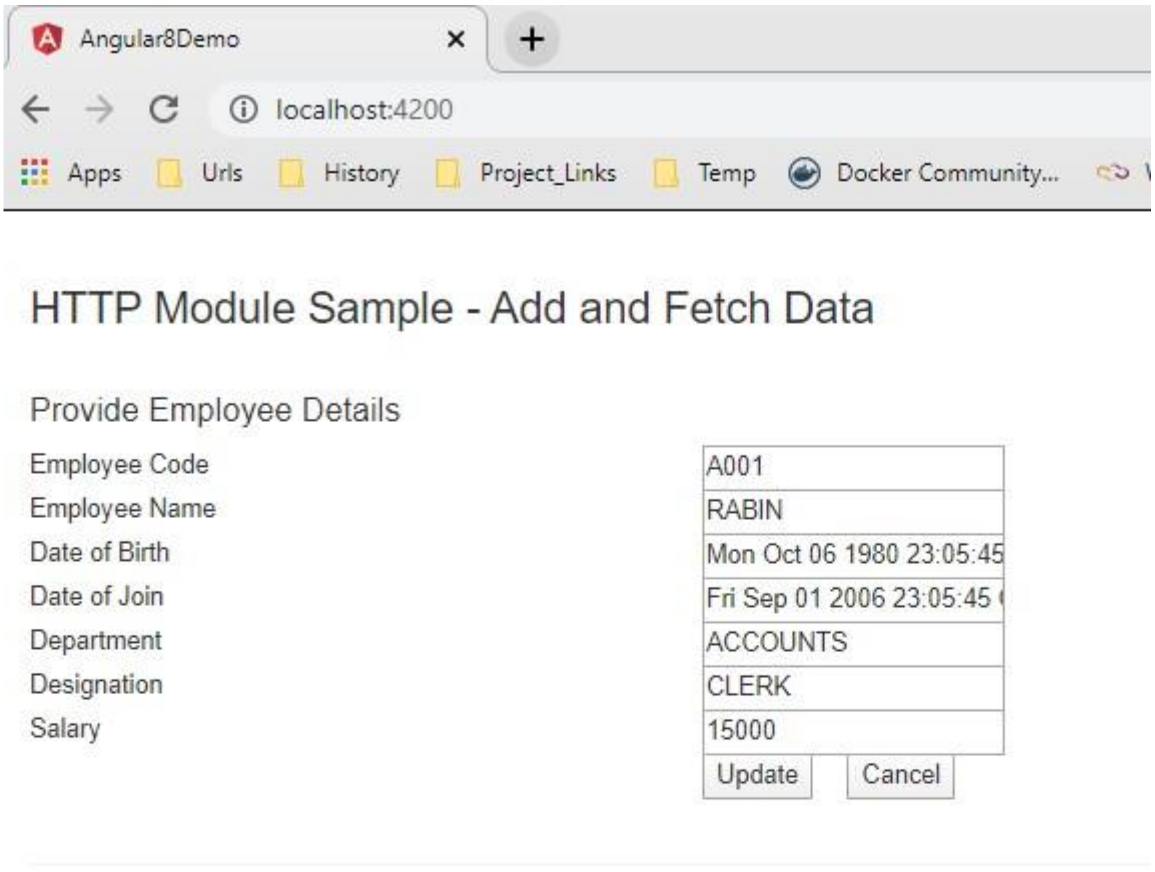


HTTP Module Sample - Add and Fetch Data

Provide Employee Details

Employee Code	
Employee Name	
Date of Birth	dd-mm-yyyy
Date of Join	dd-mm-yyyy
Department	
Designation	
Salary	

Update Employee Info



HTTP Module Sample - Add and Fetch Data

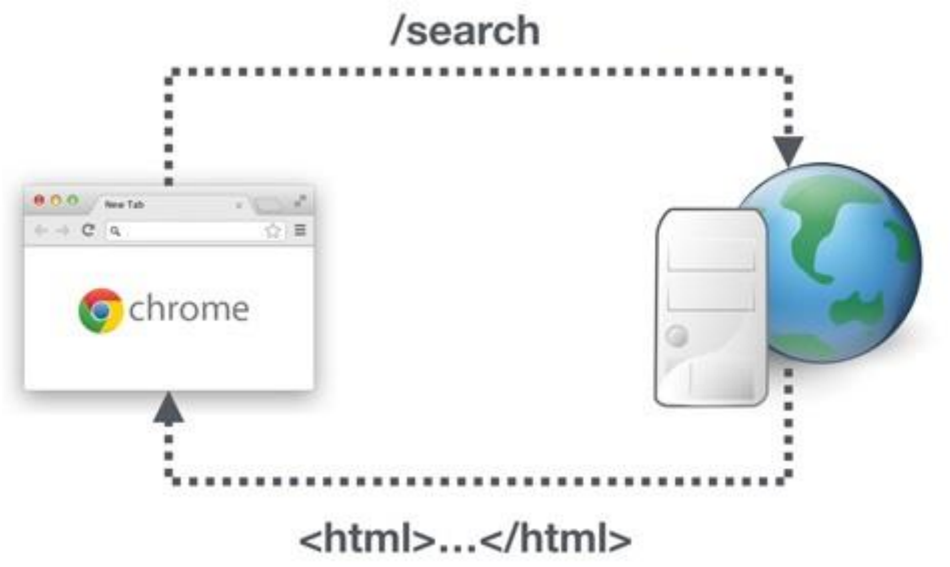
Provide Employee Details

Employee Code	A001
Employee Name	RABIN
Date of Birth	Mon Oct 06 1980 23:05:45
Date of Join	Fri Sep 01 2006 23:05:45
Department	ACCOUNTS
Designation	CLERK
Salary	15000

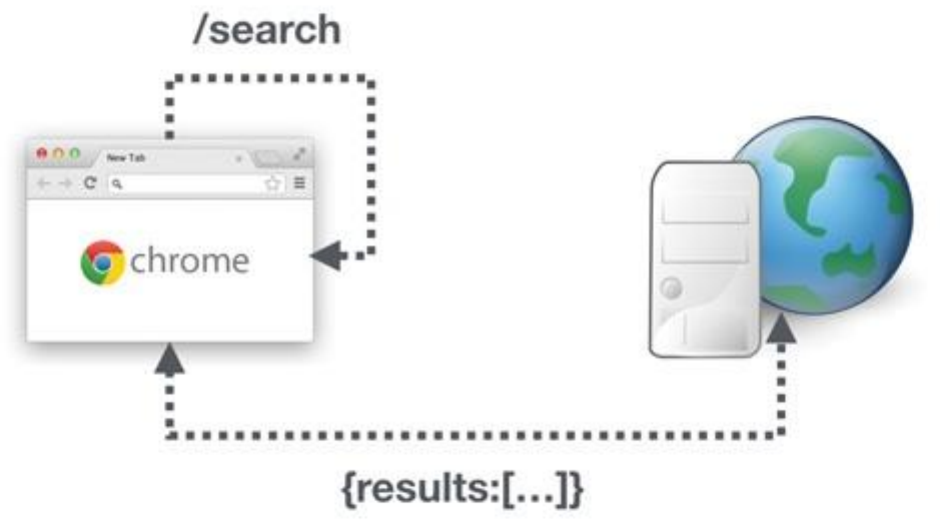
the concept of routing in an Angular application. In the basic concept, routing is a process through which we can transfer packets or data from one host to another through a network. In web application development, app routing is mainly used to map any specific URL with any specific objects. It is mainly used to access any specific UI in the application. In the past, the basic web application always used a different approach for routing in comparison to the modern one. In the older approach, we normally use the server-side routing approach. In this approach, the server normally generates the HTML related markup and sending to the client or browser. Users may fill some data into the form and again those data send to the server as a response. But, nowadays JavaScript framework or client-side framework becomes much more advanced compared previously. So, for generating HTML markup data, currently, we are not dependable on the server only. We can build that on the client-side and send it back to the browser. We just need to communicate with the server for the data part only. This type of routing concept is normally known as Client-Side routing which we will discuss in brief in this article.

Introduction to Client-Side Routing

Each and every web application always contains a URL that is normally shown in the address bar of the browser. Basically, this URL defines the current state of the application. As we known state means “all the stored information, at a given instant in time, to which program or application can access”. So in simple words, the state of an application is the current value of all the variables in the application. The Address in the URL can’t store a large volume of information, but using the URL we can bring the same state in the browser for the multiple users. As discussed in the previous section, in the traditional applications we normally user Server-Side Routing. In this process, the browser raises a request to the server to return some HTML markup data which it will display. The below image clearly has shown the process.



But we want to implement Client-Side Routing in our applications. In this process, when the URL changes in the browser address bar, our local application which is running in the browser (the client) needs to handle the changes. We do not want to send the request to the server. When we will navigate to a new site, the server returns the HTML, JavaScript, and CSS needed to render that application or page. After that, all the further changes to the URL will be handled locally by the client application. For retrieving the information related to the next UI which need to be shown, the client application will call one or more API request in the client application. At that time, a single HTML page will be returned from the server, then onwards all further modification of the page will be handled by the client and that's why it is called a Single Page Application (SPA).



The main advantage of the SPA based applications are,

1. It is much faster, since instead of making a time-consuming request to a long-distance server every time the URL changes, the client application updates the page much faster.
2. Less bandwidth or network traffic. We didn't need to send a large HTML page for every URL change request. Instead of that we just need to call a smaller API which returns just enough data to render the change in the page.
3. It is much more convenient. Since a single developer now can build most of the functionality of a site instead of splitting the effort between a front-end and a back-end or server-side developer.

Using the different modules in the Angular Framework, we can implement our web application as a SPA. For this purpose, Angular always followed a basic concept called Component Router.

Route Definition Objects

So, before going to discuss the Angular Router Modules, we first need to discuss route definition objects. For defining the router definition objects, we need to define the route type which is basically is an array of routes that defines the routing for the client application. Within this array, we can mention setup related to the expected paths, the related components which we want to use or open using the route. Each route can be defined with different attributes. Some of the common and most used attributes are,

- path – This attribute is used to mention the URL which will be shown in the browser when the application is redirecting on the specific route.
- component – This attribute is used to mention the name of the component which will be rendered when the application is on the specific route
- redirectTo – This attribute is an optional attribute. It needs to mention when we need to redirect any specific route if our main route is missing. The value of this attribute will be either component name or redirect attribute defined in the route.
- pathMatch – It is also an optional attribute property that defaults to 'prefix'. It determines whether to match full URLs or just the beginning. When defining a route with empty path string set pathMatch to 'full', otherwise it will match all paths.
- children – This attribute contains an array of route definitions objects representing the child routes of this route.

To use Routes, we need to define an array of route configurations as below,

```
const routes: Routes = [
  { path: 'component-one', component: ComponentOne },
  { path: 'component-two', component: ComponentTwo }
];
```

Concept of Router Modules

To implement routes in any web application, we need to import RouterModule in our application with the help of NgModules. RouterModule.forRoot takes the Routes array as an argument and returns a configured router module. The following sample shows how we import this module in an app.routes.ts file.

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

const routes: Routes = [
  { path: 'component-one', component: ComponentOne },
  { path: 'component-two', component: ComponentTwo }
];

export const routing = RouterModule.forRoot(routes);
```

After defining the router configuration, we need to import the router configuration into our app module files as below,

```
import { routing } from './app.routes';

@NgModule({
  imports: [ BrowserModule,routing],
  declarations: [AppComponent,ComponentOne,ComponentTwo],
  bootstrap: [ AppComponent ]
})

export class AppModule {
 }
```

When our application starts, it navigates to the empty route by default. So, we can configure the router in such a way that the router will be redirected to the mentioned route by default,

```
export const routes: Routes = [
  { path: '', redirectTo: 'component-one', pathMatch: 'full' },
  { path: 'component-one', component: ComponentOne },
  { path: 'component-two', component: ComponentTwo }
];
```

The pathMatch property, which is mainly required for redirects, instructs the router how it should match the URL provided in order to redirect to the specified route. Since pathMatch: full is mentioned, the router will be redirected to component-one when the entire URL matches ('').

In this way, when the application starts, the router system automatically loads the component-one component at the beginning as the default route.

Router Link

We can also control the navigation by using the ruterLink directive in the HTML template like as below –

```
<nav class="navbar navbar-light bg-faded">
  <a class="navbar-brand" [routerLink]="['component-one']">Component 1</a>
  <ul class="nav navbar-nav">
```

```
    <li class="nav-item active">
      <a class="nav-link" [routerLink]="['']">Home</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" [routerLink]="['component-
two']">Component 2</a>
    </li>
   </ul>
 </nav>
```

Alternatively, we can navigate to a route by calling the navigate function on the router:

```
this.router.navigate(['/component-one']);
```

An important feature of any navigation component is giving the user feedback about which link item they are currently viewing. Another way to describe this is by giving the user feedback about which route is currently active. To help in adding and removing classes depending on the currently active route Angular provides another directive called routerLinkActive. A routerLinkActive directive is associated with a route through a routerLink directive. It takes as input an array of classes which it will add to the element it's attached to if it's route is currently active, like so,

```
<a class="nav-link"
  [routerLink]="['home']"
  [routerLinkActive]="['active']">Home</a>
```

Add Route Component

In spite of defining each route component separately, we can use RouterOutlet directives which basically acts as a component placeholder in our application. Angular Framework dynamically injects the components for the active route within the <router-outlet></router-outlet> element.

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

In the above example, the component related to the route specified will be activated after the <router-outlet></router-outlet> element when the router link is clicked.

Demo 1 - Route Demo

For the demonstration of the route example in Angular, we first need to develop 3 components that are independent and can be used as a route component. For that purpose, we first create the below components – MobileComponent, TVComponent and ComputerComponent, and HomeComponent.

app.component.tv.ts

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

```

```
@Component({
    selector: 'app-tv',
    templateUrl: 'app.component.tv.html'
})

export class TvComponent implements OnInit {

    private data: Array<any> = [];

    constructor() {
        this.data = [{ name: 'LED TV 20"', company: 'Samsung', quantity: '10', price: '11000.00' },
        { name: 'LED TV 24"', company: 'Samsung', quantity: '50', price: '15000.00' },
        { name: 'LED TV 32"', company: 'LG', quantity: '10', price: '32000.00' },
        { name: 'LED TV 48"', company: 'SONY', quantity: '25', price: '28000.00' }];
    }

    ngOnInit(): void {
    }
}
```

app.component.tv.html

```
<div class="panel-body">
    <table class="table table-striped table-bordered">
        <thead>
            <tr>
                <th>Name</th>
                <th>Company</th>
                <th class="text-right">Quantity</th>
                <th class="text-right">Price</th>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="let item of data">
                <td>{{item.name}}</td>
                <td>{{item.company}}</td>
                <td class="text-right">{{item.quantity}}</td>
                <td class="text-right">{{item.price | currency}}</td>
            </tr>
        </tbody>
    </table>
</div>
```

app.component.mobile.ts

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

@Component({
    selector: 'app-mobile',
    templateUrl: 'app.component.mobile.html'
})

export class MobileComponent implements OnInit {

    private data: Array<any> = [];

    constructor() {
        this.data = [{ name: 'Galaxy Tab 3', company: 'Samsung', quantity: '10', price: '25000.00' },
        { name: 'Galaxy Tab 5', company: 'Samsung', quantity: '50', price: '55000.00' },
        { name: 'G4', company: 'LG', quantity: '10', price: '40000.00' },
        { name: 'Canvas 3', company: 'Micromax', quantity: '25', price: '18000.00' }];
    }

    ngOnInit(): void {
    }
 }
```

app.component.mobile.html

```
<div class="panel-body">
    <table class="table table-striped table-bordered">
        <thead>
            <tr>
                <th>Name</th>
                <th>Company</th>
                <th class="text-right">Quantity</th>
                <th class="text-right">Price</th>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="let item of data">
                <td>{{item.name}}</td>
                <td>{{item.company}}</td>
                <td class="text-right">{{item.quantity}}</td>
                <td class="text-right">{{item.price |currency:'INR':true}}</td>
            </tr>
        </tbody>
    </table>
</div>
```

app.component.computer.ts

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

@Component({
    selector: 'computer',
    templateUrl: 'app.component.computer.html'
})

export class ComputerComponent implements OnInit {

    private data: Array<any> = [];

    constructor() {
        this.data = [{ name: 'HP Pavilion 15"', company: 'HP', quantity: '10', price: '42000.00', specification: 'Intel Core i3 2 GB Ram 500 GB HDD with Windows 10' },
        { name: 'Lenovo Flex 2"', company: 'Lenovo', quantity: '20', price: '32000.00', specification: 'Intel Core i3 2 GB Ram 500 GB HDD with DOS OS' },
        { name: 'Lenovo Yova 500"', company: 'Lenovo', quantity: '20', price: '70000.00', specification: 'Intel Core i7 8 GB Ram 1TB HDD with Windows 8.1' }]
    }

    ngOnInit(): void {
    }
}
```

app.component.computer.html

```
<div class="panel-body">
    <table class="table table-striped table-bordered">
        <thead>
            <tr>
                <th>Name</th>
                <th>Company</th>
                <th class="text-right">Quantity</th>
                <th class="text-right">Price</th>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="let item of data">
                <td>{{item.name}}</td>
                <td>{{item.company}}</td>
                <td class="text-right">{{item.quantity}}</td>
                <td class="text-right">{{item.price | currency}}</td>
            </tr>
        </tbody>
    </table>
</div>
```

app.component.home.ts

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

@Component({
    selector: 'home',
    templateUrl: 'app.component.home.html'
})

export class HomeComponent implements OnInit {

    private message: string = '';
    constructor() {
        this.message = 'Click link to move other pages';
    }

    ngOnInit(): void {
    }
}
```

app.component.home.html

```
<div class="row">
    <div class="panel-body">
        Home Page
        <br />
        <h3 class="panel-heading"><span>{{message}}</span></h3>
    </div>
</div>
```

Now, we need to define the router link in the app-root component so that we can navigate between the different route components.

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Ang8RouteDemo';
}
```

app.component.html

```
<div class="toolbar" role="banner">
  <img
    width="40"
    alt="Angular Logo"
    src="data:image/svg+xml;base64,PHN2ZyB4bWxucz0iaHR0cDovL3d3dy53My5vcmcvMjAwMC9zdmciIHZpZXdCb3g9IjAgMCAyNTAgMjUwIj4KICAgIDxwYXRoIGZpbGw9IiNERDAwMzEiIGQ9Ik0xMjUgMzBMMzEuOSA2My4ybDE0LjIgMTIzLjFMMTI1IDIzMGw3OC45LTQzLjcgMTQuMi0xMjMuMXoiIC8+CiAgICA8cGF0aCBmaWxsPSIjQzMwMDJGIiBkPSJNMTI1IDMwdjIyLjItLjFWMjMwbDc4LjktNDMuNyAxNC4yLTEyMy4xTDEyNSAzMHoiIC8+CiAgICA8cGF0aCAgZmlsbD0iI0ZGRkZGRiIgZD0iTTEyNSA1Mi4xTDY2LjggMTgyLjZoMjEuN2wxMS43LTI5LjJoNDkuNGwxMS43IDI5LjJIMTgzTDEyNSA1Mi4xem0xNyA4My4zaC0zNGwxNy00MC45IDE3IDQwLjl6IiAvPgogIDwvc3ZnPg=="
  />
  <span>Welcome !! Angular 8 Route Demo</span>
    <div class="spacer"></div>
      <a aria-label="Angular on twitter" target="_blank" rel="noopener" href="https://twitter.com/angular" title="Twitter">

        <svg id="twitter-logo" height="24" data-name="Logo — FIXED" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 400 400">
          <defs>
            <style>
              .cls-1 {
                fill: none;
              }

              .cls-2 {
                fill: #ffffff;
              }
            </style>
          </defs>
          <rect class="cls-1" width="400" height="400" />
          <path class="cls-2" d="M153.62,301.59c94.34,0,145.94-78.16,145.94-145.94,0-2.22,0-4.43-.15-6.63A104.36,104.36,0,0,0,325,122.47a102.38,102.38,0,0,1-29.46,8.07,51.47,51.47,0,0,0,22.55-28.37,102.79,102.79,0,0,1-
```

```
32.57,12.45,51.34,51.34,0,0,0-
87.41,46.78A145.62,145.62,0,0,1,92.4,107.81a51.33,51.33,0,0,0,15
.88,68.47A50.91,50.91,0,0,1,85,169.86c0,.21,0,.43,0,.65a51.31,51
.31,0,0,0,41.15,50.28,51.21,51.21,0,0,1-
23.16.88,51.35,51.35,0,0,0,47.92,35.62,102.92,102.92,0,0,1-
63.7,22A104.41,104.41,0,0,1,75,278.55a145.21,145.21,0,0,0,78.62,
23"
            />
          </svg>

        </a>
</div>

<div class="content" role="main">
  <span><h2>{{ title }} app is running!</h2></span>
  <table style="width:40%;">
    <tr class="table-bordered">
        <td><a routerLink="/home" class="btn-
block" routerLinkActive="['active']">Home</a></td>
        <td><a routerLink="/mobile">Mobile</a></td>
        <td><a routerLink="/tv">TV</a></td>
        <td><a routerLink="/computer">Computers</a></td>
    </tr>
  </table>
  <br/>
  <div class="spacer">
    <router-outlet></router-outlet>
  </div>
</div>
```

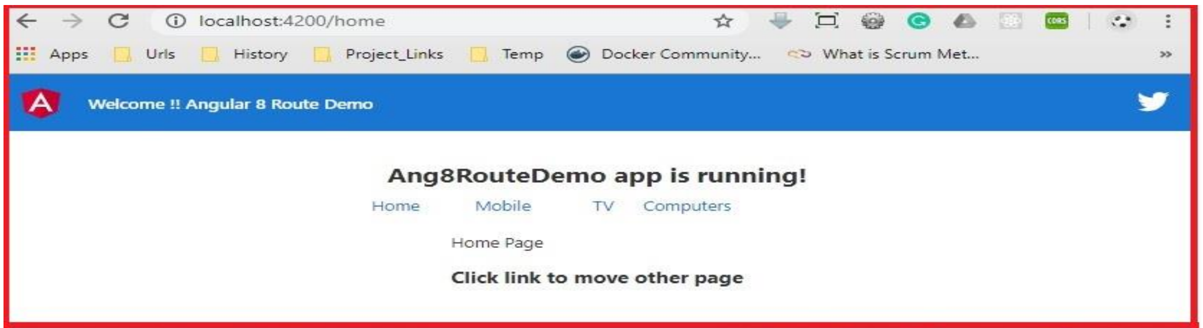
app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './app.component.home';
import { MobileComponent } from './app.component.mobile';
import { TvComponent } from './app.component.tv';
import { ComputerComponent } from './app.component.computer';

const routes: Routes = [
 { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'tv', component: TvComponent },
  { path: 'mobile', component: MobileComponent },
  { path: 'computer', component: ComputerComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

app.module.ts

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HomeComponent } from './app.component.home';
import { MobileComponent } from './app.component.mobile';
import { TvComponent } from './app.component.tv';
import { ComputerComponent } from './app.component.computer';

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

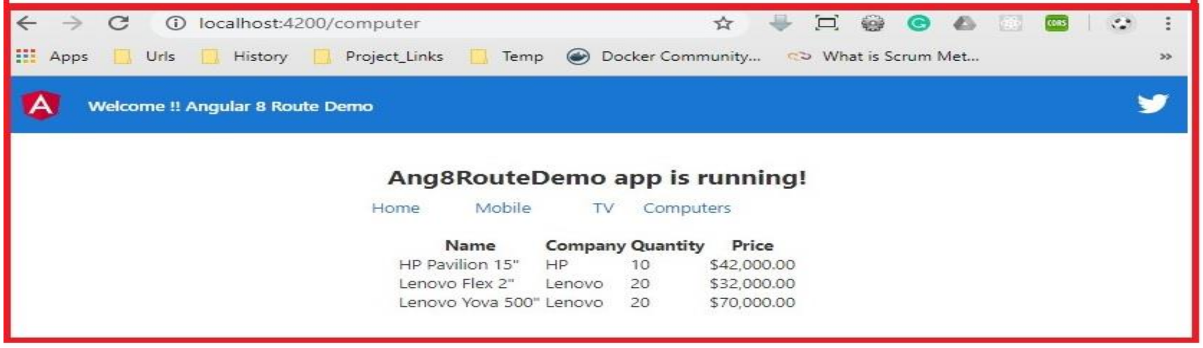
Now run the application in the browser to check the output,



Ang8RouteDemo app is running!

Home Page

Click link to move other page



Ang8RouteDemo app is running!

Name	Company	Quantity	Price
HP Pavilion 15"	HP	10	$42,000.00
Lenovo Flex 2"	Lenovo	20	$32,000.00
Lenovo Yova 500"	Lenovo	20	$70,000.00