Angular Cheat Sheet – A Basic Guide to Angular

Read Courses

Angular is a client-side TypeScript-based, front-end web framework developed by the Angular Team at Google, that is mainly used to develop scalable **single-page web applications(SPAs)** for mobile & desktop. Angular is a great, reusable UI (User Interface) library for developers that helps in building attractive, steady, and utilitarian web pages and web applications. It is a continuously growing and expanding framework which provides better ways for developing web applications. It changes the static HTML to dynamic HTML. Its features like dynamic binding and dependency injection eliminate the need for code that we have to write otherwise. **TypeScript** is a Strict Super Set of JavaScript, which means anything that is implemented in JavaScript can be implemented using TypeScript.

Angular Cheat Sheet

The Angular Cheat Sheet will give quick ideas related to the topics like Basics, Lifecycle Hooks, Components & Modules, Directives, Decorators, Angular Forms, Pipes, Services, Routing & many more, which will provide you a gist of Angular with their basic implementation. The purpose of the Cheat Sheet is to provide you with the content at a glance with some quick accurate ready-to-use code snippets that will help you to build a fully-functional web application.

Table of Content:

Angular Introduction:

Angular is a component-based application design framework, build on Typescript, for creating a sophisticated & a scalable single-page web application that has a well-integrated collection of libraries that helps

Manager(NPM) & Angular CLI installed in the system. Please refer to the **Installation of Node.js on Windows/Linux/Mac** article for the detailed installation procedure.

Steps to Install & Create a new Angular Project

After successful installation of the node.js & npm, we need to install the Angular CLI, which is described below:

Open the terminal/Command prompt & type the below command:


```
npm install -g @angular/cli
```

For checking the version of the node.js & angular CLI installed, type the below command in the cmd:

```
node -v
ng -v or ng --version
```

Now, we will create the Angular application using the Angular CLI, as given below:

```
ng new Project-Name
```

You may navigate to any of the directories where you want to create the project.

Run the application: Navigate the folder where the project has been created & type the below command to run the app:

```
cd Project-Name
```

The *ng serve* command will launch the server & will open your browser to *http://localhost:4200/*.

Example 1: This example illustrates the basic Angular web app.

---

app.component.html

```
<div>
   <h1>{{name}}</h1>
   <h3>{{details}} </h3>
</div>
```

app.component.css

```
div {
text-align: center;
font-family: Arial, Helvetica, sans-serif;
}
h1 {
color: green;
}
```

app.component.ts

```
import { Component } from '@angular/core';
@Component({
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
})
export class AppComponent {
name = 'GeeksforGeeks';
details = 'A Computer Science portal for geeks';
}
```

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
@NgModule({
declarations: [
AppComponent
],
```

```
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Angular Fundamental

This section will cover the Components, Creation of Components, Lifecycle of the Component, and various directives provided by Angular & lastly, we will discuss the Decorator. The **Component** is the basic building block for developing UI for any Angular app, which contains the tree of Angular Component. The Directives are the subset for the components, where the Components are always associated with the template. Only a single Component would be instantiated for a given element in the template. The Components should belong to the *NgModule*, which helps to avail the component to another component or application. In order to do this, the component should be listed in the *declarations* field of the *NgModule* metadata.

Component Creation: The Angular application is composed of a different set of components, where every component has different roles & functionality that is designed to perform the specific task. The Components can be created with the following command:

```
ng generate component <component-name>
```

Please refer to the Components in Angular 8 article for further detailed descriptions.

Lifecycle Hooks of Component: Every Component has its own lifecycle, which will be started when Angular instantiates the particular Component class that renders the Component view along with the child view. It is basically timed methods that differ *when* and *why* they execute. These methods will be triggered with the change detection, i.e., depending upon the conditions, the corresponding cycle will be executed. Angular will constantly check when the data-bound properties changes & accordingly update both the view & the Component instances, as required. The order of the execution for these lifecy hooks is important. The Component instance will be destroyed & will

below:

Methods	Descriptions	Syntax

ngOnChanges()	This method will be triggered when the Angular set or reset the data-bound input properties & the property value for the current & previous objects that will be received by this method. This method will be called before the *ngOnInit() method* whenever more than one data-bound input properties change.	export class AppCompo
		implements OnChanges {
		@Input() geeks: string;
		lifecycleCount: number = 20;
		ngOnChanges() {
		this.lifecycleCount–;
		}
		}

ngOnInit()	This method is used to initialize the Component or Directive after the Angular sets the initial display of the data-bound properties & sets the directive or component's input properties. This method will only be called once after the first ngOnChanges() call. This method will still be called if the ngOnChanges() method is not called.	export class AppCompo
		implements OnInit {
		@Input() geeks: string;
		lifecycleCount: number = 20;
		ngOnInit() {
		this.lifecycleCount–;
		}
		}

Methods	Descriptions	Syntax

ngDoCheck()	This method is utilized to check on depending on the changes done when Angular can not detect the modification made on its own. This method is called immediately after every changecomponents detected by the ngOnChanges() method & also called immediately after the initial execution of the ngOnInit() method.	ngDoCheck() {
		
		console.log(++this.lifecycleCc
		...
		}
		}

ngAfterContentInit()	This method is used to render the external data onto the component's view or render it into the view of that directive. It is called only once after the first ngDoCheck() call.	ngAfterContentInit() {
		...
		}

ngAfterContentChecked()	This method will be used once Angular detects the data that is rendered into the directives or component. It will be called after ngAfterContentInit() and every subsequent ngDoCheck().	ngAfterContentChecked() {
		...
		}

ngAfterViewInit()	This method will be used once Angular initializes the component's views and child views, or the	ngAfterViewInit() {
		...
		}

Methods	Descriptions	Syntax
	once after the first ngAfterContentChecked().	

ngAfterViewChecked()	This method will be used once Angular checks the component's views and child views, or the view that contains the directive. It will be called after the ngAfterViewInit() and every subsequent ngAfterContentChecked().	ngAfterViewChecked() {
		...
		}

ngOnDestroy()	This method will be used to clean up just before Angular destroys the directive or component. Unsubscribe Observables and detach event handlers to avoid memory leaks. It will be called immediately before Angular destroys the directive or component.	export class AppComponent
		destroy: boolean = true;
		DataDestroy() {
		this.destroy = !this.destroy;
		}

Modules: Every Angular application contains the root module, called NgModules, which acts as the container for the cohesive block of code, that is specifically designed to that application domain, a workflow, or a closely related set of capabilities. It consists of the components, services, & other code files, whose scope is decided by the NgModules. The NgModules in the AppModule, reside in the app.module.ts file. The **@NgModule()** class decorator is used to define the NgModule. It is a function that accepts one metadata object, whose properties describe the module. Some of the importa properties are described below:

Properties	Descriptions
declarations	The components, directives, and pipes will belong to this NgModule.
exports	It is a subset of declarations, which will be visible & usable should in the component templates of other NgModules.
imports	It depicts the other modules whose exported classes are required by the component templates declared in this NgModule
providers	It is the Creator of services, that is contributed to the global collection of services & becomes accessible in all parts of the application.
bootstrap	The root component hosts all other application views. The bootstrap property should be set by the root NgModule only.

The data can be shared between the parent component and one or more child components by implementing the *@Input()* and *@Output() decorators*, which provide the child component a way to communicate with its parent component. The **@Input() decorator** helps to update the data by the parent component in the child component, while the **@Output() decorator** helps to send the data by the child component to the parent component.

Angular Template The **Template** is a blueprint of the Angular application for developing the enhanced user interface, written in HTML & special syntax can be used within a template. Basically, each Template represents the section of the code on an HTML page that will be rendered in the browser with a lot more functionality. While generating the application through the Angular CLI, the *app.component.html* will be the default template that will contain the HTML code. The *Template Syntax* helps to control the UX/UI by coordinating the data between the class and the template. Angular provides several template syntax, which is described below:

Templates	Descriptions	Syntax
Text	Interpolation is a one-way	

Templates	Descriptions	Syntax
on	Angular, that can be used to display a component property in the respective view template with double curly braces syntax. Basically, it helps to transmit the properties mentioned in the component class to be reflected in its template.	
Template Statement	It is a method or property that can be used in the HTML code, in order to respond to specific user events. It also helps the user to engage through actions, like submitting the form data or displaying the dynamic content, etc.	<element type="button" (click)="ChangeData()">Delete</element>

Templates	Descriptions	Syntax
Binding	The Binding helps to create the connection between the HTML template & the model, along with synchronizing the view with the model, in order to inform the model while an event or user action happens in the view, or vice-versa. It always focuses on 2 parts, i.e., the target that will receive the bound value & the template expression that will produce a value from the model.	<element [(expression)]=”ClassName”> </element>
Pipes	It is used to transform the strings, currency amounts, dates, and other data, without affecting the actual content.	{{ Expression | data }}

We will learn about the various concepts of Binding & Pipes in more detail, in the underneath section.

Directives: The Directive is generally a built-in class that includes the additional behavior to elements in the Angular Applications. There are 2 types of directives:

- **Attribute Directive:** The Attribute Directive is used to modify the appearance or change the behavior of DOM elements & the components in Angular. In this case, the element, component, or another directive can be styled. The list of most commonly used attribute directives is:

Directives	Descriptions	Syntax
ngClass	This directive is used to specify the CSS classes on HTML elements & it is used to dynamically bind classes on an HTML element. The value can either be a string, an object, or an array. It must contain more than one class name, which is separated by space, in the case of a string.	<element [ng-class]=”expression”> Contents... </element>
ngStyle	This directive is used to manipulate the styles for the HTML elements. It is a set of more than one style property, that is specified with colon-separated key-value pairs.	<element [ngStyle] = “typescript_property”>
ngModel	This directive is used to create a top-level form group Instance, and it binds the form to the given form value.	<Input [(NgModel)= ‘name’]>

- **Structural Directive:** The Structural Directive is used to change the structure of the element or the component, i.e. it is used for modifying the structure of the DOM by applying or removing the elements from the DOM. The set of most commonly used built-in structural directives is:

Directives	Descriptions	Syntax
NgIf	This directive is used to remove or recreate a portion of an HTML element based on an expression. If the expression inside it is false then the element is removed and if it is true then the element is added to the DOM.	<element *ngIf=’condition’> </element>
NgFor	This directive is used to render each element for the given collection each element can be displayed on the page.	<element *ngFor=’condition’> </element>

Directives	Descriptions	Syntax
		</element>

Decorators: The Decorators are the function that is called with the prefix @ symbol, immediately followed by the class, methods, or property. The Service, filter, or directive are allowed to be modified by the Decorators before they are utilized. It facilitates the metadata for configuration that decides how the components, function or class are to be processed, instantiated & utilized during the runtime. All the Decorators can be categorized in 2 ways & each type of Decorator contains its own subset of the decorators, which are described below:

- **Class Decorators:** This decorator is facilitated by Angular having some class, which permits us to specify whether the particular class is a component or a module, along with permitting us to define this effect without having any code inside of it. For eg., the @Component that provides the metadata as part of the class & @NgModule clicks the widely used classes. Some other Class decorators are:

Class Decorator	Descriptions	Syntax
import	This is usually used to import the Directives from the @angular/core.	import { NgModule } from '@angular/core';
@Directive	It is used to define the class as the Directive & provides its metadata of it.	@Directive({ ... })
@Pipe	This is used to define the class as the Pipe, along with providing its metadata of it.	@Pipe({ ... })
@Injectable	This is used to declare the class to be injected. When this is injected somewhere, the compiler	@Injectable({

Class Decorator	Descriptions	Syntax
	without having this decorator.	

- **Class Field Decorator:** This is another category of Decorator, which can contain the following types:

Class Field Decorators	Descriptions	Syntax

Property decorator	This decorator can be utilized to decorate the particular properties declared inside the class. With this decorator, we can easily detect the purpose of using any particular property of the class. For eg., @Input(), @Output(), ReadOnly(), @Override() are most used property decorators.	class GFG {
		@ReadOnly('check')
		name: string = 'DSA';
		}

Method Decorators	This decorator is mainly used to decorate the method defined inside the class having some functionality. The most commonly used Method Decorators are @HostListener where the host element property is linked with the directive method, & @HostBinding, which helps to bind the Host element property with the Component property.	export class AppComponent {
		@HostListener('click', ['$event'])
		onHostClick(event: Event) {
		code...
		}
		}

Parameter Decorator	This decorator is permitted to decorate the parameter in the class constructors. The commonly used decorator of this type is @Inject, which is used for injecting the services in the Angular Class.	export class GFGExample{
		
		constructor(@Inject(gfgService) gfgServ)
		console.log(gfgServ);
		}

Example 2: This example describes the basic implementation of the Directives & the Decorators in Angular.

app.component.html

```
<!-- ngClass & ngIf -->
<div *ngIf="true" [ngClass]="'gfgclass'">
This text will be displayed
<!-- ngStyle -->
<h1 [ngStyle]="{'color':'#00FF00'}">
   GFG Structural Directive Example
</h1>
</div>
<hr>
<!-- ngFor -->
<div *ngFor="let item of items">
   <p> {{item}} </p>
</div>
<hr>
<!-- ngSwitch -->
<div [ngSwitch]="'one'">
   <div *ngSwitchCase="'one'">One is Displayed</div>
   <div *ngSwitchCase="'two'">Two is Displayed</div>
   <div *ngSwitchDefault>Default Option is Displayed</div>
</div>
<hr>
<h1>
   GeeksforGeeks
</h1>
<hr>
<h2>GeeksforGeeks</h2>
<!-- ngModel -->
<input [(ngModel)]="gfg" #ctrl="ngModel" required>
<p>Value: {{ gfg }}</p>
<button (click)="setValue()">Set value</button>
```

app.component.ts

```
import { Component, VERSION } from "@angular/core";
@Component({
selector: "my-app",
templateUrl: "./app.component.html",
styleUrls: ["./app.component.css"],
})
export class AppComponent {
items = ["GfG 1", "GfG 2", "GfG 3", "GfG 4"];
gfg: string = "";
setValue() {
```

app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule }
from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { AppComponent } from "./app.component";
@NgModule({
imports: [BrowserModule, FormsModule],
declarations: [AppComponent],
bootstrap: [AppComponent],
})
export class AppModule {}
```

Data Bindings

Data Binding is a technique to automatically synchronize the data between the model and view components. Angular implements data-binding that treats the model as the single source of truth in your application & for all the time, the view is a projection of the model. Unlike React, Angular supports two-way binding. In this way, we can make the code more loosely coupled. Angular broadly categorizes the data flow sequence in 2 ways:

- **One-way Binding:** This type of binding is unidirectional, i.e. this binds the data flow from the component class to the view of the same Component or vice-versa. There are various techniques through which the data flow can be bound from component to view or vice-versa. If the *data flow from component to view*, then this task can be accomplished with the help of *String Interpolation* & *Property Binding.*

Binding	Description	Syntax
Attribute Binding	It is used to set the values for attributes directly, which helps to improve accessibility, styling it dynamically, along with the ability to manage multiple CSS classes for styling simultaneously.	<element [attr.attribute-you-are-targeting]=”expression”> </element>

Binding	Description	Syntax
Binding	HTML element's class attribute, in order to set the styles dynamically.	
Event Binding	Event Binding in Angular is used to handle the events raised by the user actions like button clicks, mouse movements, keystrokes, etc. Basically, it listens for and responds to user actions. When the DOM event happens at an element(e.g. click, key down, keyup), it calls the specified method in the particular component. With Event Binding, we can bind data from DOM to the component and hence can use that data for the rest of the process.	<element (event) = function()>
Property Binding	Property Binding in Angular is a one-way data-binding technique where we can bind a property of a DOM element to a field which is a defined property in our component TypeScript code. Basically, it is used to set the values for the properties of the HTML elements or the Directives. Actually, Angular internally converts string interpolation into property binding.	[class]="variable_name"

- **Two-way Binding:** The flow of data is bidirectional i.e. when the data in the model changes, the changes are reflected in the view and when the data in the view changes it is reflected in the model. Two-way data binding is achieved by using the ng-model directive. The *ng-model directive* transfers data from the view to the model and from the model to the view. Basically, it is shorthand for a combination of *property binding* and *event binding*. According to the direction of data flow, Angular provides 3 categories to bind the data:
  - From data source to view target (One-way Binding)

view (Two-way Binding)

Depending upon the direction of data flow, Angular provides the following concepts that help to bind the data flow either from source to view, view to source, or two-way sequence of view to the source to view, which are described below:

Binding	Description	Syntax
Two-way Binding	This binding provides a way to share the data. It listens for events and updates values simultaneously between child & parent components.	<app-component [(expression)]=”DataValue”> </app-component>

The different binding types with their category, are described below:

Binding Types	Category
Interpolation, Property, Attribute, Class, Style	One-way from the data source to view-target
Event	One-way from view target to data source
Two-way	Two-way

Except for the Text interpolation, the Binding has the target name that is declared on the left-hand side of the equal sign, where the target can be property or the event, that is surrounded by a square bracket ([ ]) characters, parenthesis (( )) characters, or both ([( )]) characters. The prefix of the binding punctuation of [], (), [()] specifies the direction of data flow, i.e.,

- The square bracket (**[]**) can be used to bind from source to view.
- The parenthesis (**()**) can be used to bind from view to a source.
- The **[()]** can be used to bind in a two-way sequence of view to the source to view.

Please refer to the Difference between One-way Binding and Two-way Binding article for detailed differences between them

Example 3: This example describes the basic implementation of the Data Binding In Angular.

---

app.component.html

```
<!--Interpolation -->
<h1>{{ title }}</h1>
<hr>
<!--Property Binding -->
<input style="color:green;
   margin-top: 40px;
   margin-left: 100px;"
   [value]='title'>
<hr>
<!--Event Binding -->
<h1>GeeksforGeeks</h1>
<input (click)="gfg($event)"
   value="Geeks">
<hr>
<!--Two-way Data Binding -->
<div style="text-align: center">
   <h1 style="color: green">
      GeeksforGeeks
   </h1>
   <h3>Two-way Data Binding</h3>
   <input type="text"
   placeholder="Enter text"
   [(ngModel)]="val" />
   <br />{{ val }}
</div>
```

app.component.ts

```
import { Component } from "@angular/core";
@Component({
selector: "my-app",
templateUrl: "./app.component.html",
styleUrls: ["./app.component.css"],
})
export class AppComponent {
title = "GeeksforGeeks";
gfg(event) {
console.log(event.toElement.value);
}
val: string;
```

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule }
from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
@NgModule({
imports:      [ BrowserModule, FormsModule ],
declarations: [ AppComponent],
bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Angular Forms

The **Angular Form** is an integral part of any web application, which may contain the collection of controls for an input field, buttons, checkboxes, etc, along with having the validation check that enables to validate data entered by the user in an appropriate form. There are 2 different approaches that are facilitated by Angular, to handle the user input through the forms, i.e., the reactive form & the template-driven form.

- **Template-Driven Approach**: The conventional form tag is used to make the form. The Form object representation for the tag will be interpreted & created by Angular. The various form controls can be included with the help of ngModel & the grouping for multiple control can be done by using the ngControlGroup module. For validating the form field, basic HTML validation can be used. In simple words, it completely depends on the directive, in order to create & manipulate the basic object model.
- **Reactive Forms:** This form facilitates the model-driven approach, in order to handle the various form inputs whose values vary with time. This form is more robust, scalable, reusable & testable, as it provides direct & access explicitly to the basic form's object model.

Example 4: This example describes the basic implementation of the Angular Forms.

app.component.html

```
<input [(ngModel)]="gfg"
#ctrl="ngModel" required>
<p>Value: {{ gfg }}</p>
<button (click)="setValue()">
Set value
</button>
```

app.component.ts

```
import { Component, Inject } from "@angular/core";
import { PLATFORM_ID } from "@angular/core";
import { isPlatformWorkerApp } from "@angular/common";
@Component({
selector: "my-app",
templateUrl: "./app.component.html",
styleUrls: ["./app.component.css"],
})
export class AppComponent {
gfg: string = "";
setValue() {
this.gfg = "GeeksforGeeks";
}
}
```

app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule }
from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { AppComponent } from "./app.component";
@NgModule({
imports: [BrowserModule, FormsModule],
declarations: [AppComponent],
bootstrap: [AppComponent],
})
export class AppModule {}
```

Angular Pipes

Pipes in Angular can be used to transform the strings, currency amounts, dates, and other data, without affecting the actual content. This is a simple function that can be used with the interpolation(template expression), in order to acc

can be declared only once & can be used throughout the application. Angular facilitates 2 kinds of Pipes, i.e., built-in pipes & custom pipes.

- **Built-in Pipes:** These are the predefined pipes provided by Angular. The list of built-in Pipes provided by Angular is described below:

Pipe	Descriptions	Syntax
DatePipe	This pipe is used to format the date to the specified format.	{{ value | date }}
DecimalPipe	This pipe is used to format a value according to digit options and locale rules.	{{ value | number}}
PercentPipe	This pipe is used to Transform a number into a percentage string.	{{ value | percent [: digitsInfo [: locale]] }}
UpperCasePipe	This pipe is used to Transform all the text to uppercase.	{{ value | uppercase }}
LowerCasePipe	This pipe is used to Transform all the text to lowercase.	{{ value | lowercase }}
CurrencyPipe	This pipe is used to transform a number to a currency string, formatted according to locale rules that determine group sizing and separator, decimal-point character.	{{ value | currency }}

- **Custom Pipe**: The Custom pipes can be used for various use cases like formatting the phone number, highlighting the search result keyword, returning the square of a number, etc. For generating the custom pipes, we can follow 2 ways:

into the module file.
- By using Angular CLI, it will set up all the necessary configurations in the component & module files automatically.

The Custom Pipes can further be defined by specifying as the pure & impure pipes, which are described below:

- **Pure Pipes:** It is the pipes that execute when it detects a pure change in the input value. A pure change is when the change detection cycle detects a change to either a primitive input value (such as String, Number, Boolean, or Symbol) or object reference (such as Date, Array, Function, or Object).
- **Impure Pipes:** It is the pipes that execute when it detects an impure change in the input value. An impure change is when the change detection cycle detects a change to composite objects, such as adding an element to the existing array.

Please refer to the Explain pure and impure pipe in Angular article for further details.

Example 5: This example describes the basic implementation of the Angular Pipes.

---

app.component.html

```
<!-- DatePipe -->
<p>Date {{today | date}}</p>
<p>Time {{today | date:'h:mm a z'}}</p>
<!-- DemicalPipe -->
<p> Number: {{pi | number}} </p>
<p> Number with 4 digits: {{pi | number:'4.1-5'}} </p>
<!-- percentPipe -->
<div>
   <p>1st: {{gfg | percent}}</p>
   <p>2nd: {{geeks | percent:'4.3-5'}}</p>
</div>
<!-- Custom Pipe -->
<h2>
   Product code without using
   custom pipe is {{productCode}}
</h2>
<h2>
   Product code using custom pipe
   is {{productCode | arbitrary:'-'}}
</h2>
```

app.component.ts

```
import { Component } from "@angular/core";
@Component({
selector: "my-app",
templateUrl: "./app.component.html",
})
export class AppComponent {
today: number = Date.now();
pi: number = 3.14159;
geeks: number = 0.4945;
gfg: number = 2.343564;
productCode = "200-300";
}
```

app.module.ts

```
import { BrowserModule }
from "@angular/platform-browser";
import { NgModule } from "@angular/core";
import { AppComponent } from "./app.component";
import { ArbitraryPipe } from "./arbitrary.pipe";
@NgModule({
declarations: [AppComponent,
ArbitraryPipe],
imports: [BrowserModule],
providers: [],
bootstrap: [AppComponent],
})
export class AppModule {}
```

arbitrary.pipe.ts

```
import { Pipe, PipeTransform } from "@angular/core";
@Pipe({
name: "arbitrary",
})
export class ArbitraryPipe implements PipeTransform {
transform(value: string, character: string): string {
return value.replace(character, " ");
}
}
```

a certain task or to perform the specific operation that needs to be accomplished by the application. In simple words, Service is a class having a narrow, well-defined purpose. In order to increase the modularity and reusability, Angular help us to distinguish the Component from the Services. The purpose of using the service is to organize & share the main application logic, data or functions, and models, in order to accomplish the requirement-specific task for the different components of Angular applications. For this, we usually implement it through Dependency Injection.

Features of Service:

- The Service is a typescript class that has a *@injectable decorator* in it, which notifies that the class is a service that can be injected into the components which utilize that specific service. The other Services can also be injected as dependencies.
- The Services can be used to contain the business logic, as it can be shared with multiple components.
- The Components are singleton, i.e., the single instance of the Service will be created only, & the same instance will be utilized by every building block in the Angular application.

Dependency Injection: Dependency Injection is part of Angular Framework, which is a coding pattern, where the dependencies are received by the class, from external sources, instead of creating them itself. It allows the classes along with the Angular Decorators, like, Components, Directives, Pipes, and Injectables, that help to configure the dependencies, as required. Angular facilitates the ability to inject the service into the Components, in order to provide access to that component to the service.

The *Injectable()* decorator describes the class as a service, which allows Angular to inject it into the Component as a dependency. The *injector* is the main technique, where it is used to create the dependencies & maintain the *container* of dependency instances, which will be reutilized when required. A *provider* is an object which notifies the injectors about the process to create or obtain the dependency.

Creating a Service: The Service can be created by using the below syntax:

Syntax:

Please refer to the Explain the steps for creating a services in Angular 2 article for the detailed description.

Example 6: This example describes the basic implementation of the Services & Dependency Injection in Angular.

---

app.component.html

```
<h2>GeeksforGeeks Course Lists:</h2>
<p *ngFor="let c of courses">{{ c }}</p>
```

app.component.ts

```
import { Component } from "@angular/core";
import { CoursesService } from "./courses.service";
@Component({
selector: "my-app",
templateUrl: "./app.component.html",
styleUrls: ["./app.component.css"],
})
export class AppComponent {
courses: string[];
constructor(private _coursesService: CoursesService) {}
ngOnInit() {
this.courses = this._coursesService.getdata();
}
}
```

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { CoursesService } from './courses.service';
@NgModule({
declarations: [AppComponent],
imports: [BrowserModule],
providers: [CoursesService], //FOR GLOBAL SERVICE
bootstrap: [AppComponent]
})
export class AppModule {}
```

courses.service.ts

```
export class CoursesService {
constructor() {}
getdata(): string[] {
return [
'Data Structures,
'Algorithms',
'Programming Language',
'Aptitude',
];
}
}
```

Routing

The **Router** helps to navigate between pages that are being triggered by the user's actions. The navigation happens when the user clicks on the link or enters the URL from the browser address bar. The link can contain the reference to the router on which the user will be redirected. We can also pass other parameters with a link through angular routing. For this, it helps to create a single-page application with multiple views and allows navigation between them. The **routes** are the JSON array where each object in the array defined the route. Angular allows for creating nested routing, which means sub-navigation or sub-page navigation between the available components.

Creation of Routing Modules: Routing can only be possible to create if we have at least 2 components. The following command can be used to create the routing:

```
ng generate module module_name --flat --module=a
```

Here,

- The *flat* flag depicts that this module file is required to be there inside the app folder & we don't require to separate the folder for this module.
- The *module* flag denotes that this module is going to be part of the main app module.

Router-Outlet: The Router-outlet component basically renders any component that the router module returns, i.e. it tells the router where to display the routed views. The below syntax is used to add the router component:

The *routerLink* directive takes a path & navigates the user to the corresponding component. In other words, it is similar to *href* in HTML, which takes the path & compared it to all the paths defined in the routing module & the component that matches will be displayed in the router outlet.

The *wildcard routes help* to restrict the users to navigate to a part of the application that does not exist, i.e. when none of the path matches then the wildcard route will be used & it should always be the last route in the routes array. The router helps to select the route any time the requested URL doesn't match any router paths. We can set up the wildcard route with the following command:

```
{ path: '**', component: <component-name> }
```

Here, the 2 asterisks (**) indicate this route's definition is a wildcard route.

The Router *CanActivate* interface that a class can implement to be a guard deciding if a route can be activated. The navigation only continues, if all guards return true, otherwise the navigation will be canceled. If UrlTree is returned by any guard, then the current navigation will be canceled & new navigation will be initiated to the UrlTree returned from the guard.

The Router *CanDeactivate* interface that a class can implement to be a guard deciding if a route can be deactivated. The navigation only continues, if all guards return true, otherwise the navigation will be canceled.

Example 7: This example describes the basic implementation of Routing in Angular.

---

app.component.html

```
<!DOCTYPE html>
<html>
   <head>
      <style>
         ul {
         list-style-type: none;
         margin: 0;
         padding: 0;
         overflow: hidden;
         background-color: #333;
         }
         li {
```

```
            display: block;
            color: white;
            text-align: center;
            padding: 14px 16px;
            text-decoration: none;
            }
            li a:hover {
            background-color: #04aa6d;
            }
            .active {
            background-color: #333;
            }
         </style>
      </head>
      <body>
         <ul>
            <li>
               <a class="active" href="#home"
                  routerLink="/">Home</a>
            </li>
            <li>
               <a href="#contact"
                  routerLink="/contactus">Contact</a>
            </li>
            <li>
               <a href="#about"
                  routerLink="/aboutus">About</a>
            </li>
         </ul>
         <div style="text-align: center;
            font-weight: bolder;
            font-size: 50px">
            <router-outlet></router-outlet>
         </div>
      </body>
   </html>
```

app-routing.module.ts

```
import { NgModule } from "@angular/core";
import { RouterModule, Routes } from "@angular/router";
import { AboutUsComponent }
from "./about-us/about-us.component";
import { OurCompanyComponent } from
"./about-us/our-company/our-company.component";
import { OurEmployeesComponent } from
"./about-us/our-employees/our-employees.component";
import { ContactUsComponent }
from "./contact-us/contact-us.component";
import { HomeComponent } from "./home/home.component";
```

```
component: HomeComponent,
},
{
path: "aboutus",
children: [
{
path: "",
component: AboutUsComponent,
},
{
path: "our_employees",
component: OurEmployeesComponent,
},
{
path: "our_company",
component: OurCompanyComponent,
},
],
},
{
path: "contactus",
component: ContactUsComponent,
},
];
@NgModule({
imports: [RouterModule.forRoot(routes)],
exports: [RouterModule],
})
export class AppRoutingModule {}
```

about-us.component.html

```
<p>about-us works!</p>
<a class="btn btn-primary"
   routerLink="/aboutus/our_employees">
Our Employees
</a>
<br>
<a class="btn btn-primary"
   routerLink="/aboutus/our_company">
Our Company
</a>
```

our-company.component.html

```
<p>our-company works!</p>
<a class="btn btn-primary"
   routerLink="/aboutus">
```

our-employees.component.html

```
<p>our-employees works!</p>
<a class="btn btn-primary"
   routerLink="/aboutus">
Back
</a>
```

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { RouterModule, Router } from '@angular/router';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { ContactUsComponent }
from './contact-us/contact-us.component';
import { AboutUsComponent }
from './about-us/about-us.component';
@NgModule({
declarations: [
AppComponent,
HomeComponent,
ContactUsComponent,
AboutUsComponent
],
imports: [BrowserModule, FormsModule],
bootstrap: [AppComponent]
})
export class AppModule {}
```

Previous

Angular Cheat Sheet - A Basic Guide to Angular

Next

Angular Cheat Sheet - A Basic Guide to Angular

Similar Reads

jQuery Cheat Sheet – A Basic Guide to jQuery

HTML Cheat Sheet - A Basic Guide to HTML

CSS Cheat Sheet - A Basic Guide to CSS

JavaScript Cheat Sheet - A Basic Guide to JavaScript

Tkinter Cheat Sheet

Linux Commands Cheat Sheet

Nmap Cheat Sheet

Git Cheat Sheet

Computer Network - Cheat Sheet

Docker Cheat Sheet - Most Important Docker Commands

Complete Tutorials

HTML Tutorial

JavaScript Project Ideas with Source Code

Onsen UI

React Material UI

NuxtJS

Tutorials Archive

B bijaybhas...

Article Tags : GFG Sheets , Web-Dev Sheet , Web Technologies

Additional Information

Data Science & ML

Data Science With Python

Data Science For Beginner

Machine Learning Tutorial

ML Maths

Data Visualisation Tutorial

Pandas Tutorial

NumPy Tutorial

NLP Tutorial

Deep Learning Tutorial

HTML & CSS

HTML

CSS

Bootstrap

Tailwind CSS

SASS

LESS

Web Design

Python

Python Programming Examples

Django Tutorial

Python Projects

Python Tkinter

Web Scraping

OpenCV Python Tutorial

Python Interview Question

Computer Science

GATE CS Notes

Operating Systems

Computer Network

Database Management System

Software Engineering

Digital Logic Design

Engineering Maths

DevOps

Git

AWS

Docker

Kubernetes

Azure

GCP

DevOps Roadmap

Competitive Programming

Top DS or Algo for CP

Top 50 Tree

Top 50 Graph

Top 50 Array

Top 50 String

Top 50 DP

Top 15 Websites for CP

System Design

What is System Design

Monolithic and Distributed SD

High Level Design or HLD

Low Level Design or LLD

Crack System Design Round

JavaScript

TypeScript

ReactJS

NextJS

AngularJS

NodeJS

Web Browser

NCERT Solutions

Class 12

Class 11

Class 10

Class 9

Class 8

Complete Study Material

School Subjects

Mathematics

Physics

Chemistry

Biology

Social Science

English Grammar

Commerce

Accountancy

Business Studies

Indian Economics

Macroeconomics

Microeconimics

Statistics for Economics

Management & Finance

Management

HR Managament

Income Tax

Finance

Economics

UPSC Study Material

Polity Notes

Geography Notes

History Notes

Science and Technology Notes

Economy Notes

Ethics Notes

Previous Year Papers

SSC/ BANKING

SSC CGL Syllabus

SBI PO Syllabus

SBI Clerk Syllabus

IBPS PO Syllabus

IBPS Clerk Syllabus

SSC CGL Practice Papers

Colleges

Indian Colleges Admission & Campus Experiences

Top Engineering Colleges

Top BCA Colleges

Top MBA Colleges

Top Architecture College

Choose College For Graduation

Companies

IT Companies

Software Development Companies

Artificial Intelligence(AI) Companies

CyberSecurity Companies

Service Based Companies

Product Based Companies

PSUs for CS Engineers

Preparation Corner

Company Wise Preparation

Exams

JEE Mains

Internship Interviews

Competitive Programming

Aptitude Preparation

Puzzles

NEET

UGC NET

More Tutorials

Software Development

Software Testing

Product Management

SAP

SEO

Linux

Excel

Write & Earn

Write an Article

Improve an Article

Pick Topics to Write

Share your Experiences

Internships