Angular Decorators

Leave a Comment / 8 minutes of reading / March 9, 2023

← Angular Tutorial

Angular Observable Tutorial →

afterviewinit,afterviewchecked,aftercontentinit & aftercontentchecked

We use Angular Decorators to Provide metadata to a class declaration, method, accessor, property, or parameter in Angular. We use it at many places in our Angular App. We use it to decorate components, directives, Angular Modules, etc. In this article, let us learn what is Angular Decorator, Why it is needed, and How to create a custom Angular Decorator. We also learn list of built-in Decorators that Angular Supports.

Table of Contents

Angular Decorators

An Angular Decorator is a function, using which we attach metadata to a class declaration, method, accessor, property, or parameter.

We apply the decorator using the form **@expression**, where **expression** is the name of the decorator.

For Example, `@Component` is an Angular Decorator, which we attach to an Angular Component. When Angular sees the `@Component` decorator applied to a class, it treats the class as a component class. In the following example, it is the `@Component` decorator that makes `AppComponent` an Angular Component. Without the decorator, AppComponent is just like any other class.

```

@Component({
})
export class AppComponent {
  title = 'app';
}

```

The decorator is a Typescript feature and it is still not part of the Javascript. It is still in the Proposal stage.

To enable Angular Decorators, we need to add the `experimentalDecorators` to the `tsconfig.json` file. The `ng new` command automatically adds this for us.

```

{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true
  }
}

```

Why we need Angular Decorators

Angular uses the Decorators to Provide metadata to the class, method, or property.

For example, the `AppComponent` must know where to find its template. We do that by using the `templateUrl` parameter of the `@Component` directive. The `@Component` decorator also accepts the values for `styleUrls`, `encapsulation`, `changeDetection`, etc, which Angular uses to configure the component.

```

@Component({
  templateUrl: './app.component.html',

```

```
})
export class AppComponent {

```

To understand how a decorator works let us build a class decorator named `simpleDecorator`.

Creating a new Decorator in Angular

We use the decorator in the form **@expression**, where **expression** is the name of the decorator and it must be evaluate to a function.

In the following example, we create a function `simpleDecorator`. We will use it to decorate the `AppComponent` class. The function does not take any arguments.

```

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

@simpleDecorator
@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  name = 'Angular ' + VERSION.major;

```

```
  constructor() {
    console.log('Hello from Class constructor');
  }

  ngOnInit() {
    console.log((this as any).value1);
    console.log((this as any).value2);
  }
}

function simpleDecorator(target: any) {
  console.log('Hello from Decorator');

  Object.defineProperty(target.prototype, 'value1', {
    value: 100,
    writable: false
  });

  Object.defineProperty(target.prototype, 'value2', {
    value: 200,
    writable: false
  });
}



**** Console ***

Hello from Decorator
Hello from Class constructor
100
200


```

You can refer StackBlitz for the complete code.

As we said earlier, the `decorator` is a regular JavaScript function. Since we are using it using it on class, It gets the instance of the `AppComponent` as a argument (`target`)

```

function simpleDecorator(target: any) {
  console.log('Hello from Decorator');

```

```
5 //target is instance of AppComponent
6
```

Inside the function, we add two custom properties value1 & value2 to our AppComponent . Note that we use the defineProperty property to add a new property to the component class. Also, we add it to the prototype property of the class

```
1
2  Object.defineProperty(target.prototype, 'value1', {
3    value: 100,
4    writable: false
5  });
6
7  Object.defineProperty(target.prototype, 'value2', {
8    value: 200,
9    writable: false
10 });
11
```

Now, we can use it to Decorate our AppComponent

```
1
2 @simpleDecorator
3 @Component({
4   selector: 'my-app',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9
```

Inside the component, we can access the new properties using the keyword “this”.

```
1
2   ngOnInit() {
3     console.log((this as any).value1);
4     console.log((this as any).value2);
5   }
6
```

Decorator with Arguments

To Create a `Decorator` with arguments, we need to create a factory function, which returns the `Decorator` function. You can refer to the StackBlitz

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

@simpleDecorator({
  value1: '100',
  value2: '200'
})
@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  name = 'Angular ' + VERSION.major;

  constructor() {
    console.log('Hello from Class constructor');
  }

  ngOnInit() {
    console.log((this as any).value1);
    console.log((this as any).value2);
  }
}

function simpleDecorator(args) {
  console.log(args);

  return function(target: any) {
    console.log('Hello from Decorator');
    console.log(typeof target);
    console.log(target);

    Object.defineProperty(target.prototype, 'value1', {
      value: args.value1,
      writable: false
    });

    Object.defineProperty(target.prototype, 'value2', {
      value: args.value2,
      writable: false
```

```
    });
  };
}


```

The `simpleDecorator` takes the `args` as argument and returns the `decorator` function. The rest of the code is as same as above, except we use the `args` to populate the properties.

```

function simpleDecorator(args) {
  console.log(args);

  return function(target: any) {

```

We apply the `simpleDecorator` on the component as below

```

@simpleDecorator({
  value1: '100',
  value2: '200'
})
@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

```

List of All Angular Decorators

Angular provides several built in Decorators. We can classify them under four different categories

1. Class decorators.
2. Property decorators
3. Method decorators
4. Parameter decorators

The following is a complete list of Decorators in Angular.

Class Decorators

1. @NgModule
2. @Component
3. @Injectable
4. @Directive
5. @Pipe

Property decorators

1. @Input
2. @Output
3. @HostBinding
4. @ContentChild
5. @ContentChildren
6. @ViewChild

7. @ViewChildren

Method decorators

1. @HostListener

Parameter decorators

1. @Inject
2. @Self
3. @SkipSelf
4. @Optional
5. @Host

Class decorators

We apply class decorators to classes. `@NgModule`, `@Component`, `@Injectable`, `@Directive` & `@Pipe` are Class Decorators in Angular

@NgModule

@NgModule Decorator defines the class as Angular Module and adds the required metadata to it.

```
1
2  @NgModule({
3    providers?: Provider[],
4    declarations?: Array<Type<any> | any[]>,
5    imports?: Array<Type<any> | ModuleWithProviders<{}> | any[]>,
6    exports?: Array<Type<any> | any[]>,
7    bootstrap?: Array<Type<any> | any[]>,
8    schemas?: Array<SchemaMetadata | any[]>,
9    id?: string,
10   jit?: true
11 })
```

```
12
```

@Component

The Angular recognizes the class as Angular Component only if we decorate it with the @Component Decorator.

```
1
2  @Component({
3    changeDetection?: ChangeDetectionStrategy,
4    viewProviders?: Provider[],
5    moduleId?: string,
6
7    templateUrl?: string,
8    template?: string,
9    styleUrls?: string[],
10   styles?: string[],
11   animations?: any[],
12   encapsulation?: ViewEncapsulation,
13   interpolation?: [string, string],
14   preserveWhitespaces?: boolean,
15 })
16
17
```

@Injectable

Injectable decorator has two purposes.

One it instructs the Angular that this class needs a dependency. The Angular compiler will generate the necessary metadata to create the class's dependencies

Second, using the `providedIn` we inform the Dependency Injection system how to provide the service.

```
1
2 @Injectable({
3   providedIn?: Type<any> | 'root' | 'platform' | 'any' | null
4 })
5
```

@Directive

@Directive Decorator marks a class as an Angular directive. The directives help us to change the appearance, behavior, or layout of a DOM element.

```
1
2 @Directive({
3   selector?: string,
4   inputs?: string[],
5   outputs?: string[],
6   providers?: Provider[],
7   exportAs?: string,
8   queries?: { [key: string]: any;},
9   host?: {[key: string]: string; },
10  jit?: true
11 })
12
```

@Pipe

Decorator that marks a class as Angular Pipe and supplies configuration metadata.

```
1
2 @Pipe({
3   name: string,
4   pure?: boolean
```

```
})

```

Property Decorators

Property Decorators are applied to the properties of the class.

@Input

Input decorator marks the property as the input property. I.e it can receive data from the parent component. The parent component uses the property binding to bind it to a component property. Whenever the value in the parent component changes angular updates the value in the child component.

```
Input(bindingPropertyName?: string)

```

@Output

Output decorates the property as the output property. We initialize it as an EventEmitter . The child component raises the event and passes the data as the argument to the event. The parent component listens to events using event binding and reads the data.

```
Output(bindingPropertyName?: string)

```

@ContentChild & @ContentChildren

The ContentChild & ContentChildren are decorators, which we use to Query and get the reference to the Projected Content in the DOM. Projected content is the content that this component receives from a parent component.

```
ContentChild(
  selector: string | Function | Type<any> | InjectionToken<unknown>,
  opts?: {read?: any; static?: boolean;}
)
```

```
ContentChildren(
  selector: string | Function | Type<any> | InjectionToken<unknown>,
  opts?: { descendants?: boolean;    read?: any;}
)
```

@ViewChild & @ViewChildren

The ViewChild or ViewChildren decorators are used to Query and get the reference of the DOM element in the Component. ViewChild returns the first matching element and ViewChildren returns all the matching elements as QueryList of items. We can use these references to manipulate element properties in the component.

```
ViewChild(
  selector: string | Function | Type<any> | InjectionToken<unknown>,
  opts?: {
 read?: any; static?: boolean;}
)
```

```
ViewChildren(
  selector: string | Function | Type<any> | InjectionToken<unknown>,
  opts?: {read?: any;}
)
```

@HostBinding

The HostBinding allows us to bind to a property of the host element. The host is an element on which we attach our component or directive. This feature allows us to manipulate the host styles

```
1
2 @HostBinding(hostPropertyName?: string)
3
```

Method decorators

Method Decorators are applied to the methods of the class.

@HostListener

The HostListener listens to host events. The host is an element on which we attach our component or directive. Using HostListener we can respond whenever the user performs some action on the host element.

```
1
2 @HostListener(eventName: string, args?: string[])
3
```

Parameter decorators

Parameter Decorators are applied to the constructor parameter of the class.

@Inject

The `@Inject()` is a constructor parameter decorator, which tells angular to Inject the parameter with the dependency provided in the given token. It is a manual way of injecting the dependency

```
1
2 Inject(token:any)
3
```

@Host

The @host is a Parameter decorator that tells the DI framework to resolve the dependency in the view by checking injectors of child elements, and stop when reaching the host element of the current component.

@Self

The `@Self` decorator instructs Angular to look for the dependency only in the local injector. The local injector is the injector that is part of the current component or directive.

@SkipSelf

The `@SkipSelf` decorator instructs Angular to look for the dependency in the Parent Injector and upwards.

@Optional

@Optional marks the dependency as Optional. If the dependency is not found, then it returns `null` instead of throwing an error