Оглавление

[What is an Angular Directive? 2](#_Toc128216077)

[Attribute Directives 2](#_Toc128216078)

[Structural Directives 2](#_Toc128216079)

[Components 3](#_Toc128216080)

# Angular Directive

Directives are the functions which will execute whenever Angular compiler finds it**.** Angular Directives enhance the capability of HTML elements by attaching custom behaviors to the DOM.

From the core concept, Angular directives are categorized into three categories.

1. **Attribute Directives**
2. **Structural Directives**
3. **Components**

## Attribute Directives

Attribute Directives are responsible for manipulating the appearance and behavior of DOM elements. We can use attribute directives to change the style of DOM elements. These directives are also used to hide or show particular DOM elements conditionally. Angular provides many built-in Attribute Directives like **NgStyle**, **NgClass**, etc. We can also create our own custom Attribute Directives for our desired functionality.

## Structural Directives

Structural Directives are responsible for changing the structure of the DOM. They work by adding or removing the elements from the DOM, unlike Attribute Directives which just change the element’s appearance and behavior.

You can easily differentiate between the Structural and Attribute Directive by looking at the syntax. The Structural Directive’s name always starts with an asterisk(\*) prefix, whereas Attribute Directive does not contain any prefix. The three most popular built-in Structural Directives Angular provides are **NgIf**, **NgFor**, and **NgSwitch**.

## Components

Components are directives with templates. The only difference between Components and the other two types of directives is the Template. Attribute and Structural Directives don't have Templates. So, we can say that the Component is a cleaner version of the Directive with a template, which is easier to use.

## Creating a Custom Attribute Directive

Creating a custom directive is just like creating an Angular component. To create a custom directive we have to replace @Component decorator with @Directive decorator.

So, let's get started with creating our first Custom Attribute directive. In this directive, we are going to highlight the selected DOM element by setting an element’s background color.

Create an app-highlight.directive.ts file in src/app folder and add the code snippet below.

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

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(private eleRef: ElementRef) {

eleRef.nativeElement.style.background = 'red';

}

}

Here, we are importing Directive and ElementRef from Angular core. The Directive provides the functionality of @Directive decorator in which we provide its property selector to appHighLight so that we can use this selector anywhere in the application. We are also importing the ElementRef which is responsible for accessing the DOM element.

Now to get appHighlight Directive to work, we need to add our Directive to the declarations array in the app.module.ts file.

import ...;

import { ChangeThemeDirective } from './app-highlight.directive';

@NgModule({

declarations: [

AppComponent,

ChangeThemeDirective

],

imports: [

...

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Now we are going to use our newly created custom directive. I am adding the appHightlight directive in the app.component.html but you can use it anywhere in the application.

<h1 appHightlight>Highlight Me !</h1>

The output of the above code snippet will look like this.

### Creating a Custom Structural Directive

In the previous section, we created our first Attribute directive. The same approach is used to create the structural directive as well.

So, let’s get started with creating our structural directive. In this directive, we are going to implement the \*appNot directive which will work just opposite of \*ngIf.

Now create a app-not.directive.ts file in the src/app folder and add the code below.

import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({

selector: '[appNot]'

})

export class AppNotDirective {

constructor(

private templateRef: TemplateRef<any>,

private viewContainer: ViewContainerRef) { }

@Input() set appNot(condition: boolean) {

if (!condition) {

this.viewContainer.createEmbeddedView(this.templateRef);

} else {

this.viewContainer.clear(); }

}

}

As you saw in the above code snippet, we are importing Directive, Input, TemplateRef and ViewContainerRef from @angular/core.

Directive provides the same functionality for the @Directive decorator. The Input decorator is used to communicate between the two components. It sends data from one component to the other using property binding.

TemplateRef represents the embedded template which is used to instantiate the embedded views. These embedded views are linked to the template which is to be rendered.

ViewContainerRef is a container where one or more views can be attached. We can use createEmbeddedView() function to attach the embedded templates in the container.

Now to get the appNot directive to work, we need to add our directive to the declarations array in the app.module.ts file.

import ...;

import { AppNotDirective } from './app-not.directive';

@NgModule({

declarations: [

AppComponent,

AppNotDirective

],

imports: [

...

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

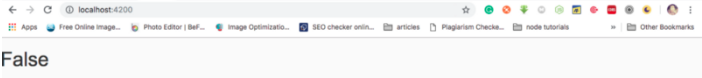
Now, it’s time to use our newly created structural directive.

I am adding the appNot directive in the app.component.html but you can use it anywhere in the application.

<h1 \*appNot="true">True</h1><h1 \*appNot="false">False</h1>

The \*appNot directive is designed in a way that it appends the template element into the DOM if the \*appNot value is false just opposite the \*ngIf directive.

The output of the above code snippet will look like this.



# Dependency injection in Angular

Imagine there is a class called HeroService that needs to act as a dependency in a component.

The first step is to add the @Injectable decorator to show that the class can be injected.

@Injectable()

class HeroService {}

The next step is to make it available in the DI by providing it. A dependency can be provided in multiple places:

At the Component level, using the providers field of the @Component decorator. In this case the HeroService becomes available to all instances of this component and other components and directives used in the template. For example:

@Component({

selector: 'hero-list',

template: '...',

providers: [HeroService]

})

class HeroListComponent {}

When you register a provider at the component level, you get a new instance of the service with each new instance of that component.

At the NgModule level, using the providers field of the @NgModule decorator. In this scenario, the HeroService is available to all components, directives, and pipes declared in this NgModule. For example:

@NgModule({

declarations: [HeroListComponent]

providers: [HeroService]

})

class HeroListModule {}

When you register a provider with a specific NgModule, the same instance of a service is available to all components in that NgModule. To understand all edge-cases, see Hierarchical injectors.

At the application root level, which allows injecting it into other classes in the application. This can be done by adding the providedIn: 'root' field to the @Injectable decorator:

@Injectable({

providedIn: 'root'

})

class HeroService {}

When you provide the service at the root level, Angular creates a single, shared instance of the HeroService and injects it into any class that asks for it. Registering the provider in the @Injectable metadata also allows Angular to optimize an app by removing the service from the compiled application if it isn't used, a process known as tree-shaking.

Injecting a dependency

The most common way to inject a dependency is to declare it in a class constructor. When Angular creates a new instance of a component, directive, or pipe class, it determines which services or other dependencies that class needs by looking at the constructor parameter types. For example, if the HeroListComponent needs the HeroService, the constructor can look like this:

@Component({ … })

class HeroListComponent {

constructor(private service: HeroService) {}

}

When Angular discovers that a component depends on a service, it first checks if the injector has any existing instances of that service. If a requested service instance doesn't yet exist, the injector creates one using the registered provider, and adds it to the injector before returning the service to Angular.

When all requested services have been resolved and returned, Angular can call the component's constructor with those services as arguments.

