How To Create & Use Custom Directive In Angular

2 Comments / March 9, 2023 / 7 minutes of reading

⟵ ngFor TrackBy Angular Tutorial Angular Pipes ⟶

In this tutorial, we will show you how to create a Custom Directive in Angular. The Angular directives help us to extend or manipulate the DOM. We can change the appearance, behavior, or layout of a DOM element using the directives. We will build a four directive example s and show you how to

1. Create a custom directive using the `@Directive` decorator.
2. We will create both custom attribute directive & custom Structural directive.
3. How to setup selectors
4. Pass value to it using the `@input`.
5. How to respond to user inputs,
6. Manipulate the DOM element (Change the Appearance) etc.

Table of Contents

Angular Directives
Creating Custom Attribute Directive
Creating Custom Structural Directive
    Why you need to specify *
Custom Directive Examples
    Toggle Directive
    Tooltip Directive
References

Angular Directives

The Angular has three types of directives.

1. Components
2. Structural Directives
3. Attribute Directives

Components are directives with Template (or view). We know how to build Angular Components. Structural & Attribute directives do not have an associated view.

Structural directives change the DOM layout by adding and removing DOM elements. All structural Directives are preceded by the Asterix (`*`) symbol.

The Attribute directives can change the appearance or behavior of an element.

Creating Custom Attribute Directive

The Angular has several built-in attribute directives. Let us create a `ttClass` directive, which allows us to add class to an element. Similar to the Angular ngClass directive.

Create a new file and name it as `tt-class.directive.ts`. import the necessary libraries that we need.

```

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

```

Decorate the class with `@Directive`. Here we need to choose a selector (`ttClass`) for our directive. We name our directive as `ttClassDirective`.

```

@Directive({
  selector: '[ttClass]',
})
export class ttClassDirective implements OnInit {

```

Our directive needs to take the class name as the input. The `Input` decorator marks the property `ttClass` as the input property. It can receive the class name from the parent component.

We use the same name same as the select name `ttClass`. This will enable us to use the property binding syntax `<button [ttClass]="'blue'">` in the component.

You can also create more than `@Input` properties.

```

@Input() ttClass: string;

```

We attach the attribute directive to an element, which we call the parent element. To change the properties of the parent element, we need to get the reference. Angular injects the parent element when we ask for the instance of the `ElementRef` in its constructor.

```
constructor(private el: ElementRef) {
}
```

`ElementRef` is a wrapper for the Parent DOM element. We can access the DOM element via the property `nativeElement`. The `classList` method allows us to add the class to the element.

```
ngOnInit() {
  this.el.nativeElement.classList.add(this.ttClass);
}
```

The complete code is as shown below.

```
import { Directive, ElementRef, Input, OnInit } from '@angular/core'

@Directive({
  selector: '[ttClass]',
})
export class ttClassDirective implements OnInit {

  @Input() ttClass: string;

  constructor(private el: ElementRef) {
  }

  ngOnInit() {
    this.el.nativeElement.classList.add(this.ttClass);
  }

}
```

BEST ANGULAR BOOKS

BEST ANGULAR BOOKS

The Top 8 Best Angular Books, which helps you to get started with Angular

In the `app.component.css` and the CSS class `blue`

```
.blue {
  background-color: lightblue;
}
```

Finally in the component template attach our customer directive `ttClass` to the `button` element.

```
<button [ttClass]="'blue'">Click Me</button>
```

You can see from the image below, that `class='blue'` is inserted by Our Custom Directive.

The above is a simple imitation of ngClass. Have a look at the source code of ngClass

Creating Custom Structural Directive

Now, let us build a Custom Structural directive. Let us mimic the `ngIf` and create a custom directive, which we name it as `ttIf`. There is hardly any difference in creating a Attribute or structural directive.

We start of with creating a `tt-if.directive.ts` file and import the relevant modules.

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

Decorate the class with `@Directive` with the selector as (`ttIf`). We name our directive as `ttIfDirective`.

```
@Directive({
  selector: '[ttIf]'
})
export class ttIfDirective  {
```

A variable to hold our if condition.

```
_ttif: boolean;
```

Since, we are manipulating the DOM, we need `ViewContainerRef` and `TemplateRef` instances.

```
constructor(private _viewContainer: ViewContainerRef,
        private templateRef: TemplateRef<any>) {
}

```

Our directive needs to take the if condition as the input. The `Input` decorator marks the property `ttIf` as the input property. Note that we are using setter function, because we want the add or remove the content whenever the if condition changes.

We use the same name same as the select name `ttIf`. This will enable us to use the property binding syntax `<div *ttIf="show">` in the template.

```
@Input()
set ttIf(condition) {
  this._ttif = condition
  this._updateView();
}

```

This is where all the magic happens. We use the `createEmbeddedView` method of the `ViewContainerRef` to insert the template if the condition is `true`. The `clear` removes the template from the DOM.

```
_updateView() {
  if (this._ttif) {
    this._viewContainer.createEmbeddedView(this.templateRef);
  }
  else {
    this._viewContainer.clear();
  }

```

That it. Remember to `ttIfDirective` in the declaration array of the `app.module.ts`. The complete code is as shown below.

```

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

@Directive({
  selector: '[ttIf]'
})
export class ttIfDirective  {

  _ttif: boolean;

  constructor(private _viewContainer: ViewContainerRef,
          private templateRef: TemplateRef<any>) {
  }


  @Input()
  set ttIf(condition) {
    this._ttif = condition
    this._updateView();
  }

  _updateView() {
    if (this._ttif) {
      this._viewContainer.createEmbeddedView(this.templateRef);
    }
    else {
      this._viewContainer.clear();
    }
  }

}

```

Component class

```

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title: string = "Custom Directives in Angular";
  show=true;
}
```

```
}
```

Template

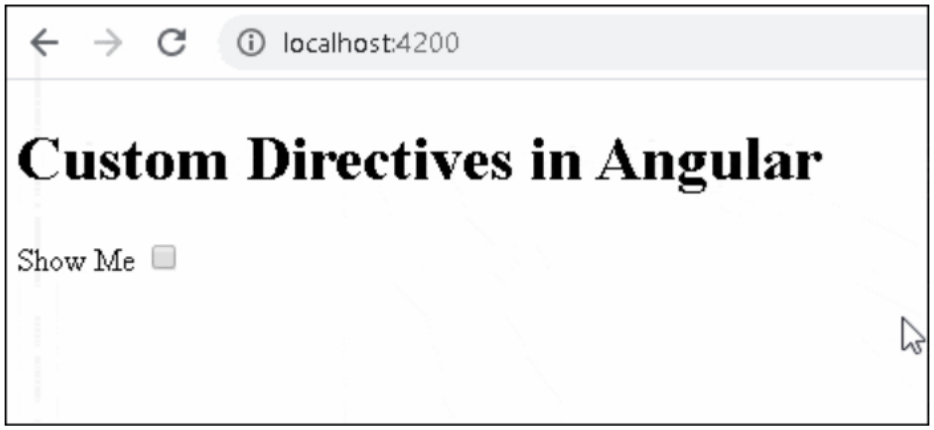
```
<h1> {{title}} </h1>

Show Me
<input type="checkbox" [(ngModel)]="show">

<div *ttIf="show">
 Using the ttIf directive
</div>

<div *ngIf="show">
 Using the ngIf directive
</div>
```

Run the app and compare the `ngIf` & our custom directive `ttIf` side by side.



Why you need to specify *

Remove the * from our newly created `ttIf` directive. And you will get the error message

```
ERROR NullInjectorError: StaticInjectorError(AppModule)[NgIf -> TemplateRef]:
StaticInjectorError(Platform: core)[NgIf -> TemplateRef]:
NullInjectorError: No provider for TemplateRef!
```

We use the `*` notation to tell Angular that we have a structural directive and we will be manipulating the DOM. It basically tells angular to inject the `TemplateRef`. To inject the `templateRef`, the Angular needs to locate the template. The `*` tells the Angular to locate the template and inject its reference as `templateRef`

Custom Directive Examples

The following two more Custom Directive Examples. `Toggle` & `Tooltip` directives

Toggle Directive

The following directive adds or removes the CSS class `toggle` from the Parent element. We do that by listening to the click event on the host element or parent element.

Angular makes this easy to listen to the events from the parent or host element using the `@HostListener` function decorator. We use it to decorate the function (`onClick` method in the example). It accepts the name of the event as the argument and invokes the decorated method whenever the user raises the event.

```

@HostListener('click')
private onClick() {

```

The complete code of the `ttToggleDirective` is as follows.

```

import { Directive, ElementRef, Renderer2, Input, HostListener, HostBinding } from '@angular/c

@Directive({
  selector: '[ttToggle]',
})
export class ttToggleDirective {

```

```
  private elementSelected = false;

  constructor(private el: ElementRef) {
  }

  ngOnInit() {
  }

  @HostListener('click')
  private onClick() {
    this.elementSelected = !this.elementSelected;
    if (this.elementSelected) {
      this.el.nativeElement.classList.add('toggle')
    } else {
      this.el.nativeElement.classList.remove('toggle')
    }
  }

}

```

Add the following CSS Class

```

.toggle {
  background-color: yellow
}

```

Use it as follows.

```

<button ttToggle>Click To Toggle</button>

```

Tooltip Directive

The tooltip directive shows the tip whenever the user hovers over it. The directive uses the HostListener to listen to the `mouseenter` and `mouseleave` events.

The `showHint` method adds a `span` element into the DOM and sets its top & left position just below the host element. The `removeHint` removes it from the DOM.

```

import { Directive, ElementRef, Renderer2, Input, HostListener } from '@angular/core'

@Directive({
  selector: '[ttToolTip]',
})
export class ttTooltipDirective {

  @Input() toolTip: string;

  elToolTip: any;

  constructor(private elementRef: ElementRef,
          private renderer: Renderer2) {
  }

  @HostListener('mouseenter')
  onMouseEnter() {
    if (!this.elToolTip) { this.showHint(); }
  }

  @HostListener('mouseleave')
  onMouseLeave() {
    if (this.elToolTip) { this.removeHint(); }
  }

  ngOnInit() {
  }

  removeHint() {
    this.renderer.removeClass(this.elToolTip, 'tooltip');
    this.renderer.removeChild(document.body, this.elToolTip);
    this.elToolTip = null;
  }
```

```
  showHint() {

    this.elToolTip = this.renderer.createElement('span');
    const text = this.renderer.createText(this.toolTip);
    this.renderer.appendChild(this.elToolTip, text);

    this.renderer.appendChild(document.body, this.elToolTip);
    this.renderer.addClass(this.elToolTip, 'tooltip');

    let hostPos = this.elementRef.nativeElement.getBoundingClientRect();
    let tooltipPos= this.elToolTip.getBoundingClientRect();

    let top = hostPos.bottom+10 ;
    let left = hostPos.left;

    this.renderer.setStyle(this.elToolTip, 'top', `${top}px`);
    this.renderer.setStyle(this.elToolTip, 'left', `${left}px`);
  }
}
```

Add the following CSS Class

```
.tooltip {
  display: inline-block;
  border-bottom: 1px dotted black;
  position: absolute;
}
```

Use it as follows.

```
<button ttToolTip toolTip="Tip of the day">Show Tip</button>
```

References

1. @Directive decorator
2. ngClass Source Code
3. Directives Source Code

Read More

1. Angular Tutorial
2. Angular Directives
3. ngFor
4. ngSwitch
5. ngIf
6. ngClass
7. ngStyle
8. ngFor Trackby
9. Custom Directive
10. Angular @input, @output & EventEmitter
11. HostListener & HostBinding

← ngFor TrackBy Angular Tutorial Angular Pipes →

2 thoughts on “How to Create & Use Custom Directive In Angular”

KIRAN NERELLA

AUGUST 25, 2022 AT 2:43 PM

For custom structural directive we need to include that in the declarations array of app.module.ts.

Reply

JULIAN HONMA

NOVEMBER 12, 2020 AT 8:18 PM

There is one missing step at the end of *Creating Custom Attribute Directive* (the author mentions it in the next section though).
Simply creating the directive won't work, it needs to be declared in `app.module.ts` in the `@NgModule` > `declarations` like so:

```
@NgModule({
declarations: [
AppComponent,
ttClassDirective
],
...
})
```

You might need to save the html file again to see the changes on your preview.

(Other subject)
I had an issue where the tooltip wouldn't appear. After debugging, I found that I had left a link to `bootstrap.min.css` in `app.component.html` which has its own `.tooltip` class.
Renaming our custom tooltip class from `tooltip` to `tt-tooltip` fixed the issue.

Reply

Leave a Comment

Your email address will not be published. Required fields are marked *

Type here..

Name*

Email*

Website

Post Comment »

This site uses Akismet to reduce spam. Learn how your comment data is processed.

Copyright © 2023 TekTutorialsHub |