**Directives**

* Directives are elements which change appearance or behaviour of DOM elements
* A **directive** is a custom HTML element that is used to extend the power of HTML.
* Directives are perhaps the most important bit of an Angular application, and if we think about it, the most-used Angular unit, the component, is actually a directive.
* An Angular component isn’t more than a directive with a template. When we say that components are the building blocks of Angular applications, we’re actually saying that directives are the building blocks of Angular applications.
* At the core, a directive is a function that executes whenever the Angular compiler finds it in the DOM. Angular directives are used to extend the power of the HTML by giving it new syntax.

**Component is also a type of directive with a template , styles and logic part .**

**The naming convention for components is name.component.ts**

Angular 2 has the following directives that get called as part of the BrowserModule module.

* ngif
* ngFor

If you view the app.module.ts file, you will see the following code and the BrowserModule module defined. By defining this module, you will have access to the 2 directives.

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

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

\*ngIf = 'expression'

### Syntax

\*ngFor = 'let variable of variablelist'

An Angular component isn’t more than a directive with a template. When we say that components are the building blocks of Angular applications, we’re actually saying that directives are the building blocks of Angular applications.

Each directive has a name — either one from the Angular predefined like ng-repeat, or a custom one which can be called anything. And each directive determines where it can be used: in an element, attribute, class or comment.

By default, from Angular versions 2 and onward, Angular directives are separated into three different types:

### Components

Components are just directives with templates. Under the hood, they use the directive API and give us a cleaner way to define them.

—directives with a template.

*Components* are the most common of the three directives.

The other two directive types don’t have templates. Instead, they’re specifically tailored to DOM manipulation.

### Structural directives

Adds or removes DOM elements to change DOM Layout

These are specifically tailored to create and destroy DOM elements.

*Structural Directives* change the structure of the view.

Two structural directives are [NgFor](https://angular.io/guide/template-syntax#ngFor) and [NgIf](https://angular.io/guide/template-syntax" \l "ngIf).

There r times when u want to show part of a page depending upon some condition

Showing or Hiding part of a page

Three Structural directives

ngFor > Repeater Dierctive that iteratres over ollection of data

ngIf > Adds or Removes an element from the Dom

ngSwitch > Displays one element out of multiple elements based on a ondition

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

//template: '<app-book-form> </app-book-form>',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'demo';

courses =[1,2];

}

.html

<div \*ngIf = "courses.length>0">

List of Courses

</div>

<div \*ngIf="courses.length==0">

No Courses available

</div>

We can use ng-template instead of if again

<div \*ngIf = "courses.length>0; else noCourses">

List of Courses

</div>

<ng-template ="#noCourses">

No Courses available

</ng-template>

<div \*ngIf = "courses.length>0; then CoursesList else noCourses">

</div>

<ng-template ="#CoursesList">

List of Courses

</ng-template>

<ng-template ="#noCourses">

No Courses available

</ng-template>

We can also use hidden attribute instead of ngIf

<div hidden>

List of Cousrses

</div>

<div>

No Courses

</div>

This will hide this div without any condition

Property binding : Bing this property with some DOM Element

We can also use this a s a property of the DOM object

<div [hidden]= "courses.length==0">

List of Cousrses

</div>

<div [hidden] ="courses.length > 0">

No Courses

</div>

Here, element is in the DOM , but its hidden

To work wih large element trees, use \*ngIF

ngSwitchCase

ul.nav.nav-pills tab

(li>a) \*2

<ul class="nav nav-pills">

<li [class.active]="viewMode=='map'"><a href="">Map View </a></li>

<li [class.active]="viewMode=='list'"><a href=""></a>List View </li>

</ul>

<div [ngSwitch] = "viewMode">

<div \*ngSwitchCase="'map'">

Map View Content

</div>

<div \*ngSwitchCase="'list'">

List View Content

</div>

<div \*ngSwitchDefault> Otherwise </div>

</div>

If you have to compare value of a field or property against multiple values , use ngSwitch

<ul class="nav nav-pills">

<li [class.active]="viewMode=='map'"><a (click)="viewMode='map'">Map View </a></li>

<li [class.active]="viewMode=='list'"><a (click)="viewMode='list'">List View </a></li>

</ul>

<div [ngSwitch] = "viewMode">

<div \*ngSwitchCase="'map'">

Map View Content

</div>

<div \*ngSwitchCase="'list'">

List View Content

</div>

<div \*ngSwitchDefault> Otherwise </div>

</div>

viewMode=”map”;

ngFor

Courses =[

{ id : 1, name :'ADCA'},

{ id : 2, name :'PGDC'},

{ id : 3, name :'DCA'},

{ id : 4, name :'CCA'},

<div \*ngFor = "let course of Courses">

{{course.id}}

{{course.name}}

</div>

<ul>

<li \*ngFor ="let course of Courses">

{{course.id}}

{{course.name}}

</li>

</ul>

Here idex, even , odd , first , lsat sre the exported values

<ul>

<li \*ngFor ="let course of Courses ; index as i">

{{i}} {{course.id}}

{{course.name}}

</li>

</ul>

<ul>

<li \*ngFor ="let course of Courses ; even as isEven">

{{i}} {{course.id}} <span \*ngIf="isEven"> EVEN </span>

{{course.name}}

</li>

</ul>

ngFor \*Change Detection

Angular has change detection mechanism. Whenever

You click a button, Angular Ajax request or a timer function completes, it performs change detection. It knows that this Courses list has a new item added .

<button (click)="AddCourse()" > Add Course </button>

<ul>

<li \*ngFor ="let course of Courses ; even as isEven">

{{i}} {{course.id}} <span \*ngIf="isEven"> EVEN </span>

{{course.name}}

</li>

</ul>

AddCourse()

{

this.Courses.push( { id: 5, name : 'DCST'});

}

OnRemove(course)

{

let index = this.Courses.indexOf(course);

this.Courses.splice(index,1);

}

<button (click)="AddCourse()" > Add Course </button>

<ul>

<li \*ngFor ="let course of Courses ; even as isEven">

{{i}} {{course.id}} <span \*ngIf="isEven"> EVEN </span>

{{course.name}}

<button (click)="OnRemove(course)"> Remove Course </button>

</li>

</ul>

Some attribute directives — like hidden, which shows or hides an element — basically maintain the DOM as it is. But the structural Angular directives are much less DOM friendly, as they add or completely remove elements from the DOM. So, when using these, we have to be extra careful, since we’re actually changing the HTML structure.

### Attribute directives

Attribute directives manipulate the DOM by changing its behavior and appearance.

* An **Attribute** directive changes the appearance or behavior of a DOM element.
* change the appearance or behavior of an element, component, or another directive.

We use attribute directives to apply conditional style to elements, show or hide elements or dynamically change the behavior of a component according to a changing property.

ngStyle > Sets inline styles dynamicxally based on state of the component

div [ngStyle]="{'color' :'green'}"> THIS IS A ATTRIBUTE DIRECTIVE </div>

Add these classes in style.cs

.myClass1

{

color: yellow;

background-color : orange;

font-size : 24px;

}

.myClass2

{

color: red;

background-color : green;

font-size : 24px;

}

<div [ngClass]="{myClass1 : true }"> THIS IS NGCLASS EXAMPLE </div>

ngClass > Adds or Removes css classes

## Using the Existing Angular Directives

The ngClass directive is a good example of an existing Angular attribute directive:

<p [ngClass]="{'blue'=true, 'yellow'=false}">

Angular Directives Are Cool!

</p>

<style>

.blue{color: blue}

.yellow{color: yellow}

</style>

So, by using the [ngClass directive](https://angular.io/api/common/NgClass) on the example below, we’re actually adding the blue class to our paragraph, and explicitly not adding the yellow one. Since we’re changing the appearance of a class, and not changing the actual HTML structure, this is clearly an attribute directive.

The real power of Angular directives comes with the ability to create our own. Angular provides a clean and simple API for creating custom directives, and that’s what we’ll be looking at in the following sections.

### Creating an attribute directive

Creating a directive is similar to creating a component. But in this case, we use the @Directive decorator. For our example, we’ll be creating a directive called “my-error-directive”, which will highlight in red the background of an element to indicate an error.

Let’s start by creating a file called app.myerrordirective.ts on the src/appfolder and adding the following code to it:

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

@Directive({

selector:'[my-error]'

})

export class MyErrorDirective{

constructor(elr:ElementRef){

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

}

}

After importing the Directive from @angular/core we can then use it. First, we need a selector, which gives a name to the directive. In this case, we call it my-error.

Best practice dictates that we always use a prefix when naming our Angular directives. This way, we’re sure to avoid conflicts with any standard HTML attributes. We also shouldn’t use the ng prefix. That one’s used by Angular, and we don’t want to confuse our custom created Angular directives with Angular predefined ones. In this example, our prefix is my-.

We then created a class, MyErrorDirective. To access any element of our DOM, we need to use [ElementRef](https://angular.io/api/core/ElementRef). Since it also belongs to the @angular/core package, it’s a simple matter of importing it together with the Directive and using it.

We then added the code to actually highlight the constructor of our class.

To be able to use this newly created directive, we need to add it to the declarations on the app.module.ts file:

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

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

import { MyErrorDirective } from './app.myerrordirective';

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

@NgModule({

imports: [ BrowserModule ],

declarations: [ AppComponent, MyErrorDirective ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

Finally, we want to make use of the directive we just created. To do that, let’s navigate to the app.component.ts file and add the following:

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

@Component({

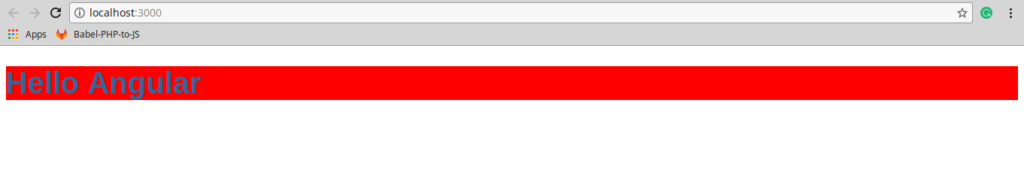
selector: 'my-app',

template: `<h1 my-error>Hello {{name}}</h1>`,

})

export class AppComponent { name = 'Angular'; }

The final result looks similar to this:



### Creating a structural directive

In the previous section, we saw how to create an attribute directive using Angular. The approach for creating a structural behavior is exactly the same. We create a new file with the code for our directive, then we add it to the declarations, and finally, we use it in our component.

For our structural directive, we’ll implement a copy of the ngIf directive. This way, we’ll not only be implementing a directive, but also taking a look at how Angular directives handle things behind the scenes.

Let’s start with our app.mycustomifdirective.ts file:

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

@Directive({

selector: '[myCustomIf]'

})

export class MyCustomIfDirective {

constructor(

private templateRef: TemplateRef<any>,

private viewContainer: ViewContainerRef) { }

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

if (condition) {

this.viewContainer.createEmbeddedView(this.templateRef);

} else {

this.viewContainer.clear();

}

}

}

As we can see, we’re using a couple of different imports for this one, mainly: Input, TemplateRef and ViewContainerRef. The Input decorator is used to pass data to the component. The TemplateRef one is used to instantiate **embedded views**. An embedded view represents a part of a layout to be rendered, and it’s linked to a template. Finally, the ViewContainerRef is a container where one or more Views can be attached. Together, these components work as follows:

Directives get access to the view container by injecting a ViewContainerRef. Embedded views are created and attached to a view container by calling the ViewContainerRef’s createEmbeddedView method and passing in the template. We want to use the template our directive is attached to so we pass in the injected TemplateRef. — from Rangle.io’s [*Angular 2 Training*](https://angular-2-training-book.rangle.io/handout/advanced-angular/directives/view_containers_and_embedded_views.html)

Next, we add it to our declarators:

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

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

import { MyErrorDirective } from './app.myerrordirective';

import { MyCustomIfDirective } from './app.mycustomifdirective';

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

@NgModule({

imports: [ BrowserModule ],

declarations: [ AppComponent, MyErrorDirective, MyCustomIfDirective ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

And we use it in our component:

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

@Component({

selector: 'my-app',

template: `<h1 my-error>Hello {{name}}</h1>

<h2 \*myCustomIf="condition">Hello {{name}}</h2>

<button (click)="condition = !condition">Click</button>`,

})

export class AppComponent {

name = 'Angular';

condition = false;

}

The kind of approach provided by structural directives can be very useful, such as when we have to show different information for different users based on their permissions. For example, a site administrator should be able to see and edit everything, while a regular user shouldn’t. If we loaded private information into the DOM using an attribute directive, the regular user and all users for that matter would have access to it.

### Angular Directives: Attribute vs Structural

There’s a simple rule that can help us choose the right one. Basically, if the element that has the directive will still be useful in the DOM when the DOM is not visible, then we should definitely keep it. In this case, we use an attribute directive like hidden. But if the element has no use, then we should remove it. However, we have to be careful to avoid some common pitfalls. We have to avoid the pitfall of always hiding elements just because it’s easier. This will make the DOM much more complex and probably have an impact on overall performance. The pitfall of always removing and recreating elements should also be avoided. It’s definitely cleaner, but comes at the expense of performance.

All in all, each case should be carefully analyzed, because the ideal solution is always the one that has the least overall impact on your application structure, behavior and performance. That solution might be either attribute directives, structural directives or, in the most common scenario, a compromise between both of them.

Attribute Directives

An Attribute directive changes the appearance or behavior of a DOM element.

**Custom Directives**

<h1 RedColor> This is a text </h1>

We need to tell that here, RedColor is not a HTML Directive , Example of HTML Directives are style, class , etc.

Second, what action it will perform

So, create a Custom Directive

Ng g directive RedColor

Check in app.module.ts file. Its added there

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

@Directive({

selector: '[RedColor]'

})

export class RedColorDirective {

constructor() { }

}

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

@Directive({

selector: '[RedColor]'

})

export class RedColorDirective {

constructor(element : ElementRef) {

element.nativeElement.style.color = "Red";

}

}