Directives and Pipes

CS569 – Web Application Development II

Maharishi University of Management

Department of Computer Science

Assistant Professor Asaad Saad

Maharishi University of Management - Fairfield, Iowa

Attribute Directives & Structural Directives

Attribute Directives

They are applied like HTML attributes and impact the element they are attached. (DOM friendly)

Structural Directives

They change the structure of the DOM, not only the element on which they sit. They usually have a * before their name. (DOM un-friendly)

Components

Yes components are directives but with template.

Built-in Directives

Directives add behavior to their host elements.

The built-in directives are imported and made available to your components automatically when importing **`BrowserModule`**.

- **`ngIf`**
- **`ngSwitch`**
- **`ngStyle`**
- **`ngClass`**
- **`ngFor`**
- **`ngNonBindable`**

ngIf

The condition is determined by **the result of the expression** that you pass in to the directive.

If the result of the expression returns a **`false`** value, the element will be removed from the DOM.

```
<div *ngIf="false"></div> <!-- never displayed -->
<div *ngIf="a > b"></div> <!-- displayed if a is more than b -->
<div *ngIf="myFunc()"></div> <!-- displayed if myFunc returns true -->
```

The ***** indicates that this directive treats the component/tag as a **template**, ngIF is a **Structural Directive**

De-Sugaring of Structural Directives

Angular provides a nice syntax for structural directives (`*ngIf`). It transforms the * syntax into a less beautiful syntax behind the scenes.

```
<div *ngIf="false"></div>
```


Desugared

```
<ng-template [ngIf]="false">
        <div></div>
</ng-template>
```

The directive css selector name is `[ngIf]` but the * is just to tell angular to create the template for me

ngIf else

```
<div *ngIf="isValid; else other_content">
        content here ...
</div>

<ng-template #other_content>other content here...</ng-template>
```

ngStyle

To set a given DOM element CSS properties

```
<div [style.backgroundColor]="'yellow'">Yellow background</div>
```

Use it to set one css property.

Value is a literal string.

```
<span [style.fontSize.px]="16">
        Red text
</span>
```

Specify units

```
[style.fontSize.px] [style.fontSize.em] [style.fontSize.%]
```

```
<div [ngStyle]="{color: 'white', 'background-color': 'blue'}">
        White text on blue background
</div>
```

Use it to set multiple css properties.

Value is a JavaScript Object literal (quote invalid keys)

ngClass

We pass an **object literal**. The object is expected to have the **keys** as the **class names** and the **values** should be a **truthy/falsy** value to indicate whether the class should be applied or not.

```
.special { background-color: yellow } .big { font-size: 36px }
```

```
<div [class.special]="false">class special will not be applied</div>
<div [class.big]="true">class big will be applied</div>
<div [ngClass]="{special: false, big: true}">text</div>
```

We can also use a list of class names to specify multiple classes should be added to the element.

```
<div class="base" [ngClass]="['special', 'big']">Text</div>
```

ngFor

To repeat a given DOM element. The syntax is:

```
*ngFor="let item of items"
```

The `let item` syntax specifies a (template) variable that's receiving each element of the `items` array

The `items` is the collection of items from your controller.

```
@Component({
        template: `<ul>
                        <li *ngFor="let name of names">Hello {{ name }}</li>
                    </ul>`
})
export class MyComponent {
        public names: string[]
        constructor() {
                this.names = ['Asaad', 'Mike', 'Mada'];
        }
}
```

Getting an Index

```
@Component({
        template: `<ul>
                        <li *ngFor="let name of names; let num = index">
                                {{ num+1 }} - Hello {{ name }}
                        </li>
                    </ul>`
})

export class MyComponent {
        constructor( public names: string[]) {
                names = ['Asaad', 'Mike', 'Mada'];
        }
}
```

ngNonBindable

We use ngNonBindable when we don't want to compile or bind a particular section of our page.

```
<div>
   <span ngNonBindable>This {{ Hello }} will not be evaluated</span>
</div>
```

Creating our Custom Directive

To create a new custom Directive class from Angular CLI we use:

```
ng g d directiveName
```

Notice

- Angular creates a Directive using the **@Directive()** decorator
- Directives don't have template/styles
- The selector is a **CSS selector [attribute]**
- CLI will add our directive class to the **`declarations[]`** array in **`module.ts`** along with all other components to be instantiated.

Services we usually use within Directives

All these Services can be imported from `@angular/core`

Reference to the Element we are applying the directive on

Reference to a Service (helper Object)

```
constructor(   private element: ElementRef,
               private renderer2: Renderer2  ){

        element.nativeElement.style.fontSize = '22px';
        renderer2.setStyle(element.nativeElement, 'font-size', '22px');
}
```

When you set an **input** property in your directive, you cannot read its value from the constructor, cause it's not been created yet! You will have to wait for the constructor to create the directive object and set this property first, then you may read it within another stage of the directive lifecycle (ngOnInit).

Host Element

To turn an Angular component into something rendered in the DOM you have to associate an Angular component with a DOM element. We call such elements host elements.

A directive can interact with its host DOM element in the following ways:

- It can listen to its events.
- It can update its properties.
- It can invoke methods on it.

Angular automatically checks host property bindings during change detection. If a binding changes, it will update the host element of the directive.

Example

```
@HostBinding('style.backgroundColor') background:string;
@HostBinding('value') value:string;
@HostBinding('attr.role') role:string;
@HostBinding('style.width.px') width:number;
@HostBinding('disabled') disabled:boolean;
```

set the properties of the host element

```
@HostListener('mouseenter') foo(){
        this.renderer.setStyle(this.elem.nativeElement, 'color', 'red')
}
@HostListener("input", "$event.target.value") onChange(updatedValue: string) {
        this.value = updatedValue.trim();
}
```

Listening and Writing to Host

DOM Interaction

We don’t interact with the DOM directly. Angular aims to provide a higher-level API, so the native platform, the DOM, will just reflect the state of the Angular application. This is useful for a couple of reasons:

- It makes components easier to refactor.
- It allows unit testing most of the behavior of an application without touching the DOM.
- It allows running Angular applications in a web worker, server, or other platforms where a native DOM isn’t present.

Main Point

There are three kinds of directives in Angular: Components, Structural directives and Attribute directives.

Components are the most common of the three directives.

Structural Directives change the structure of the view. Two examples are NgFor and NgIf in the Template Syntax page.

Attribute directives are used as attributes of elements. The built-in NgStyle directive in the Template Syntax page, for example, can change several element styles at the same time.

Pipes

Pipes transform displayed values within a template.

Example: *When data arrives in our component, we could just push their raw values directly to the view. That might make a bad user experience. Everyone prefers a simple birthday date like April 15, 1988 to the original raw string format — Fri Apr 15 1988 00:00:00 GMT-0700 (Pacific Daylight Time).*

```
{{ dateObj | date }} // output is 'Apr 15, 1988'
```

Built-in Pipes

@angular/common

- async
- currency
- date
- decimal
- json
- lowercase
- percent
- slice
- uppercase

Examples:

```
<p>{{ myValue | uppercase }}</p>
<p>{{ myDate | date:"MM/dd/yy" }}</p>
<p>{{ myValue | slice:3:7 | uppercase }}</p>
```

Custom Pipes

To create a custom pipe from CLI: **`ng g p myPipe`** or by adding the **`@Pipe`** decorator to a class.

Custom pipes should be declared in the **`declarations[]`** array at **`app.module.ts`**

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({ name: 'double' })
export class DoublePipe implements PipeTransform {
        transform(value: any, args?: any): any { return value * 2; }
}

<input type="text" #input (keyup)="0">
<p>{{ input.value | double }}</p>
```

Pure and Impure Pipes

When Pipe is pure, it means that Angular will **NOT re-run** them on the value they are applied to upon each change detection cycle. This behavior makes sense, as it saves performance.

If you need to **re-run the pipe on each change detection cycle**, you may mark your pipe as impure by setting **'pure' to false**.

```
@Pipe({ name: 'myPipe', pure: false })
export class myPipe implements PipeTransform { ... }
```

Async Pipe

The **async pipe** (a built-in pipe) is an impure pipe. Its job is to fetch asynchronously returned data from Promises or Observables.

Therefore, the async pipe is a great helper if you want to print some data to the screen which isn't available upon component initialization.

```
@Component({
        template: `<p>{{asyncValue | async}}</p> `
})
export class AppComponent {
        asyncValue = new Promise((resolve, reject) => {
                        setTimeout(() => resolve('Data is here!'), 2000); });
}
```

What's going to happen if we don't add the async pipe?

keyvalue Pipe

You may pipe an Object through the **keyvalue** pipe, which will give you an array suitable for use within an ***ngFor**.

```
@Component({
   template: `<div *ngFor="let item of data | keyvalue">
                    {{item.key}} - {{item.value}}
              </div>`
})
export class MyComponent {
   data = { "key": "value", "key2": "value2" };
}
```