JOHN BRYCE
Leading in IT Education
a matrix company

Directives

Advanced Directives in Angular 5

Our Goals

JOHN BRYCE
Leading in IT Education
a matrix company

- Understand what we can create with directives
- Create attribute directives
- Create structural directives

Attribute Directives

JOHN BRYCE
Leading in IT Education
a matrix company

- Attribute directives change the appearance or behaviour of an element, component, or another directive
- **ngClass** is an example of an attribute directive
- **ngClass** adds or removes css classes on an HTML element
- Example usage:

```
<some-element [ngClass]="{'first': true, 'second': true, 'third': false}">...</some-element>
```

- We will try to learn about attribute directive by creating our own simple **ngClass** directive

Bootstrapping Our App

JOHN BRYCE
Leading in IT Education
a matrix company

- We will use **@angular/cli** to create a new playground project to experiment with our directive endeavour.

  > ng new directives-tutorial

- After the project is installed we will also use **@angular/cli** to create our directive file

  > ng generate directive MyNgClass

MyNgClass @Input

JOHN BRYCE
Leading in IT Education
a matrix company

- The original **ngClass** directive gets an input

  <some-element [ngClass]=”someInput”></some-element>
- **someInput** can be a string, Array, Object.
- We will simplify our case and except only an Object
- As for deciding what is the selector of our directive, since the directive can be assigned to element or component we need to keep it either as class or attribute selector
- We will keep our selector similar to **ngClass** and make it an attribute selector
- Let’s add the input and selector logic to our component

ElementRef

- In order to add our classes we need a reference to the host DOM element
- We can ask the DI for **ElementRef** which wraps the host element
- In **ElementRef** there is a property **nativeElement** which you can use to directly access the **DOM**
- We can use the **@Input** decorator on a setter function
- The **nativeElement** can also be null, in what cases does **nativeElement** will be null?
- Let's manipulate the DOM and add the proper classes

Directly Accessing the DOM

JOHN BRYCE
Leading in IT Education
a matrix company

- In the new age of web development there is a problem coupling our application code with the DOM
  - Server side rendering won't work
  - Web workers won't work
- One solution is to run our code only in browser - **isPlatformBrowser**
  - problem is that there is no reason why our directive won't work on the server or on web workers
- A better solution will be to manipulate the DOM with **Renderer2**

What is Renderer2?

JOHN BRYCE
Leading in IT Education
a matrix company

- **Renderer2** is an injectable service
- Wrapper around DOM manipulation
- Support manipulating the DOM through web workers (by **postMessage/onmessage**)
- Support Server side

Using Renderer2

JOHN BRYCE
Leading in IT Education
a matrix company

- Let’s change our directive and make the dom manipulations through the **Renderer2** service
- We can use the renderer **addClass/removeClass** methods which get’s the **nativeElement** and the class name to add or remove

Adding Behavior to Our Directive

JOHN BRYCE
Leading in IT Education
a matrix company

- Our directive can change the appearance of a DOM element
- What if we want our directive to respond to user events
- We can decorate our class methods with the **@HostListener** decorator which accepts as argument the name of the event
- This will cause the decorated method to run when the events happen
- Angular will also know to clear the event when directive is destroyed
- As the second argument of the decorator you can pass array of string as arguments for the method for example

  @HostListener('mouseover', ['$event']) // will pass the original event object

- In our directive let's add a **mouseover** event that will log a message to the console

Bug in Our ngClass

- Let’s try and examine a use case for our directive
- We will move the classes object to the **app.component.ts** into a public property and our directive will read the class object from that property
- Let’s create a button that when clicked will toggle the boolean values of the keys of the object
- What do you think, will the classes in the element change? if so why not?

By Reference by Value

- Remember that primitives are passed by value and non primitives like Object and arrays are passed by reference
- This means that out **@Input** decorator won't detect a change and run our setter function
- One way to solve it is to use Immutable library
- Another way to solve it is by customizing the directive change detection
- What lifecycle hook is running every change detection?

Custom Change Detection

JOHN BRYCE
Leading in IT Education
a matrix company

- We can hook to **DoCheck** hook and check if our input is changed
- But checking if an object is changed is a complex task of its own
- Angular can help we that with **Differs** services

Differs

- **Differs** services are used to track changes on an object over time
- They can specify when a change is made and what was the previous value and what's the current value
- Two kind of **Differ Factories** are provided by Angular and can be injected
  - **KeyValueDiffers -** used on object
  - **IterableDiffers -** used for iterables… can u think of a directive that use this differ?
- We can create the differ for our directive in the **ngOnChanges**
  - `this._differ = this._differs.find(value).create();`
- We can use the differ we are creating to see the changes on the **ngDoCheck**

```
const changes: KeyValueChanges<string, boolean> =
this._differ.diff(this.classObject);
```

Structural Directives

JOHN BRYCE
Leading in IT Education
a matrix company

- Structural directives manipulate the DOM
- Can you give examples of structural directives you know?
- Structural directives have an asterix (*) before the attribute: ***ngIf**
- You can only apply one structural directive to one host element

The Asterix (*) Prefix

JOHN BRYCE
Leading in IT Education
a matrix company

- The asterix tells Angular to wrap the host element in an **ng-template**
- For example this usage of ***ngIf**

```
<div *ngIf="todo" class="name">{{todo.title}}</div>
```

will be translated to:

```
<ng-template [ngIf]="todo">
  <div class="name">{{todo.title}}</div>
</ng-template>
```

- Another example with ***ngFor**

```
<div *ngFor="let hero of heroes; let i=index; let odd=odd; trackBy:
trackById" [class.odd]="odd">
  ({{i}}) {{hero.name}}
</div>
```

The Asterix (*) Prefix

JOHN BRYCE
Leading in IT Education
a matrix company

- Another example with ***ngFor**

```
<div *ngFor="let task of tasks; let i=index; let odd=odd" [class.odd]="odd">
  ({{i}}) {{hero.name}}
</div>
```

will be translated to:

```
<ng-template ngFor let-task [ngForOf]="tasks" let-i="index" let-odd="odd" >
  <div [class.odd]="odd">({{i}}) {{hero.name}}</div>
</ng-template>
```

- The syntax we place in the structural directive is not a regular expression and it follows Angular **Microsyntax** to short the expression we place

Microsyntax

JOHN BRYCE
Leading in IT Education
a matrix company

- Microsyntax lets you configure a directive in a compact, friendly string
- The microsyntax parser translates that string into attributes on the **ng-template**
  - **let -** declare template input variable that you reference within the template
  - **of, trackBy -** the parser capitalize them and prefix them with the directive attribute name: **ngForOf, ngForTrackBy** and those become two input params for the directive
  - there are a few additional context variables that **NgFor** sets like **index, odd,** and we assigne them to template input variable
  - ***ngFor="let task of tasks"** we also define a template input variable called **task** and we didn't specify what to assign it from the context so by default it's set to **$implicit** context variable
- The microsyntax is available for us when we develop our own structural directives

Template Input Variable

JOHN BRYCE
Leading in IT Education
a matrix company

- Template input variable is a variable whose value you can reference within a single instance of the template
- In out **ngFor** example we are defining a few of them:
  - **task, i, odd**
- We declare a template input variable using the **let** keyword in our **microsyntax** expression

exportAs

JOHN BRYCE
Leading in IT Education
a matrix company

- In the metadata of the **@Directive/@Component** we can set the **exportAs** key
- That key accepts string
- The **exportAs** control how to refer to the instance of this **directive** or **component** when using **Template reference variable**

Our ngIf

- From what we learned let's try to create our own ngIf
- In our directive we can inject the **ng-template** with **TemplateRef**
- We can also inject the view container of the directive with **ViewContainerRef**
- With the **createEmbeddedView** in **ViewContainerRef** we can choose when to create our **TemplateRef**
- The **clear** method of **ViewContainerRef** can clear the **ViewContainerRef**
- Let's try to create a simple structural directive, our own implementation of ***ngIf**

Our ngFor

JOHN BRYCE
Leading in IT Education
a matrix company

- Let's create our own simple case of **ngFor**
- The syntax will be

  **<div *myFor="let task of tasks" > **

- Recall the **task** will be a template input variable set to **$implicit**
- Recall the **of tasks** will be translate to **[myForOf]="tasks"**
- When we call the **createEmbeddedView** we can pass template context and we can pass the **$implicit**
- Lets try and create ***myFor**

Summary

JOHN BRYCE
Leading in IT Education
a matrix company

- Directives are a powerful tool in Angular
- We can create attribute directive which adds behaviour or appearance
- Or we can create structural directives which change the DOM
- To create our directives we use the tools we learned in the previous lessons:
  - **ng-template**
  - **ViewContainerRef**
  - **Differs**
  - **Microsyntax**
  - **Template input variable**