

# Directives

BUILDING WEB APPLICATIONS USING ANGULAR



# Introduction

- Directives come in three forms in Angular
  - Attribute directives – changing the appearance or behaviour of an element, component or other directive
  - Structural directives – changing the DOM through the additional or removal of elements
  - Components – directives with templates

# Attribute Directives - Creation

- Attribute directives make changes to an element and are simply classes with the Directive decorator added to them

```
@Directive({  
    selector: '[appSpecial]'  
})  
export class SpecialDirective { }
```

- The constructor of such a class is passed an ElementRef by Angular, which provides access to the underlying DOM element

```
export class SpecialDirective {  
    constructor(private el: ElementRef) {  
        el.nativeElement.style.fontSize = "5em"  
    }  
}
```

```
<p appSpecial>Some very large text indeed</p>
```

## Attribute Directives – Reacting to Events

- We can also listen for events on the element through the use of the `HostListener` decorator which we attach to methods to handle such events.

```
export class SpecialDirective {  
    ...  
  
    @HostListener('mouseenter') onMouseEnter() {  
        this.show();  
    }  
}
```

- The method can have any name, it is the string passed to the decorator that describes the event we're listening for

# Attribute Directives – Taking Input

- Attribute Directives can take values just as we saw with Components

```
export class SpecialDirective {  
    @Input() fontSize: number  
}
```

- Perhaps to then tailor the directive according to the value passed (remember our life cycle events?)

```
export class SpecialDirective {  
    @Input() fontSize: number  
  
    ngOnInit(private el: ElementRef) {  
        el.nativeElement.style.fontSize = this.fontSize  
    }  
}
```

```
<p appSpecial fontSize="5em">
```

# Structural Directives

- Structural directives are used to modify the DOM, they can be fairly simple like the built in \*ngIf directive or quite complex like the built in \*ngFor directive
- They are created through the use of the @Directive decorate applied to a class, it is then injected with a TemplateRef and ViewContainerRef

```
@Directive({
  selector: '[appLoggedIn]'
})
export class LoggedInDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef,
  ) { }
```

## Structural Directives - \* Syntax

- The \*syntax we're used to seeing on Angular's structural directives is there to simplify our lives, but it is important to know what the expanded syntax looks like so that we understand why we build Structural Directives the way we do.

```
//easy on the eyes syntax
<div *ngIf="bool">...</div>

//the same as writing
<ng-template [ngIf]="bool">
  <div>...</div>
</ng-template>
```

- Angular parses the \*syntax into the ng-template version you see below. Take the directive up to an ng-template which then wraps the target element and its children.

# Structural Directives - \* Syntax

- We can go further and look at \*ngFor

```
//easy(ish) on the eyes syntax
<div *ngFor="let user of users; let i=index" [class.odd]="odd">
  ({{i}}) {{hero.name}}
</div>

//the same as writing
<ng-template ngFor let-user [ngForOf]="users" let-i="index">
  <div [class.odd]="odd">({{i}}) {{user.name}}</div>
</ng-template>
```

- The let keyword creates a template input variables (in this case, **user** and **i**)
- The microsyntax parser take the “of” keyword, title-cases them and prefixes them with the directive name (ngForOf)
- The directive sets and resets context object properties, for \*ngFor these include \$implicit and index. Our template input variable “i” gets assigned to the context object property “index” and as “user” did not specify a property, it receives \$implicit.



# Structural Directives

- Now that we know the syntax and the way it is parsed under the covers, we can write our own directives to take advantage of these features. Reminding ourselves of the basic construct:

```
@Directive({
  selector: '[appLoggedIn]'
})
export class LoggedInDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef,
  ) { }
```

- The `templateRef` gives us access to the `<ng-template>` that angular creates for us when using the `*syntax`
- The `viewContainer` gives us the ability to render the view based on our logic and any data being passed back and forth

# Structural Directives

- Angular creates a view-container adjacent to the host element (the one we placed the directive on)
- This view-container is where we place an embedded view, created using the angular generated ng-template

```
this.viewContainer.createEmbeddedView(this.templateRef);
```

- So we can use some logic to intercept the creation of this view, such as how `*ngIf` creates/destroys the view and `*ngFor` creates many copies of the view.

```
if (bool) {  
    this.viewContainer.createEmbeddedView(this.templateRef);  
}
```

```
for (...) {  
    this.viewContainer.createEmbeddedView(this.templateRef);  
}
```

## Structural Directives - @Input variables

- To pass data into the directive we use our input variables as we have done with Components and Attribute directives

```
<div *qaSpecial="inputData">...</div>
```

```
@Input() set qaSpecial(inputData) {  
    //do something with inputData  
}
```

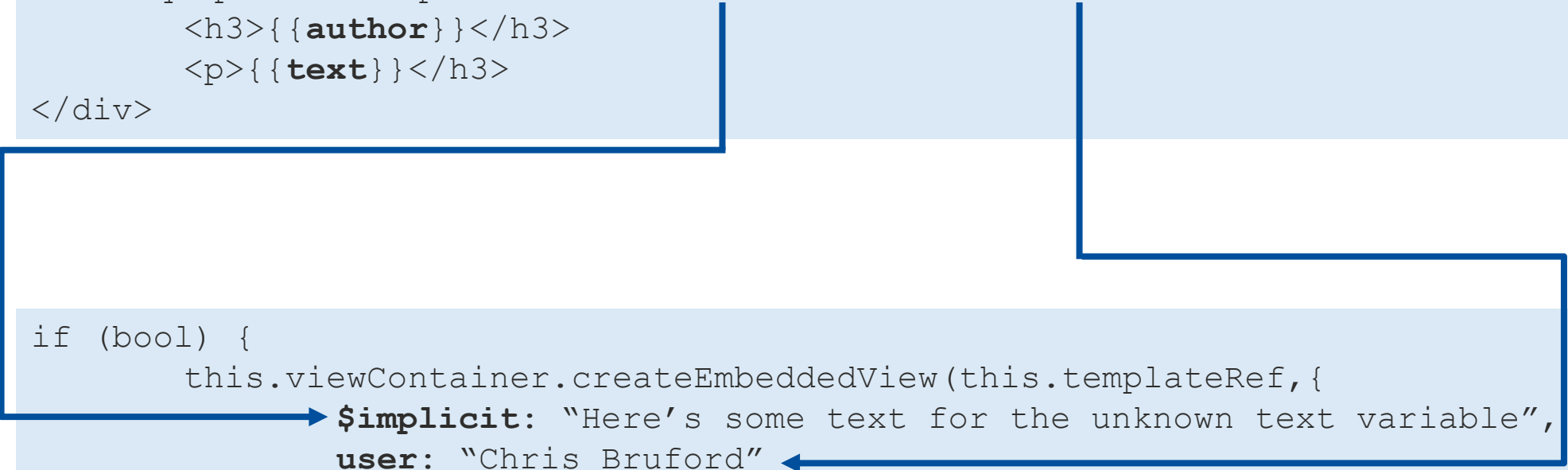
- We use a setter here so that if inputData changes our logic is re-evaluated. No getter is required as no one is fetching this data.

# Structural Directives – Template Input Variables

- If we wish to use data returned from the directive in the rendering of our view, much like we saw with `*ngFor` then we need to use the context object

```
<div *qaSpecial="inputData; let text; let author=user">
  <h3>{{author}}</h3>
  <p>{{text}}</p>
</div>
```

```
if (bool) {
  this.viewContainer.createEmbeddedView(this.templateRef, {
    $implicit: "Here's some text for the unknown text variable",
    user: "Chris Bruford"
  });
}
```



# Exercise

BUILD A CUSTOM DIRECTIVE

