PAGE CONTENTS

AUTHOR: MACIEJ WÓJCIK

InDepth guide to repeat HTML elements multiple times — a short introduction to structural directives

What you will learn

- How to implement the custom structural directive in Angular
- How to use template variables
- How to implement a custom NgFor for your specific needs
- How to refactor component's code using directives

Intro

Almost everyday we encounter the task to render a list of times in a component view. This includes repeating HTML and

But do you know that we can create our own structural directives? In fact, there are cases when it's a good idea to build a custom structural directive to render a template that requires repeating HTML. Particularly if you want to repeat the same elements - like in some rating widget, which consists of 5 or more “stars” UI elements. You may imagine the syntax for this case could be similar to presented below:

```
<div *repeat> ⭐ </div>
```

In this guide we will learn how to create a directive, step by step, by implementing a structural directive to repeat HTML elements. I will also touch briefly on the transformation that the compiler applies to asterisk syntax, template variables, and compare our approach with Angular `NgFor`.

Dynamic HTML

Rendering the `HTML` elements in Angular is something that we do daily and it is a very easy concept - whenever we write `HTML` code in our component’s template, Angular will process this template and render it during the runtime automatically using the same elements that we implemented.

Rendering `HTML` elements that will be added to the DOM in a dynamic way — is a slightly more complicated process. The most popular techniques use `NgIf` and `NgFor` directives to render elements dynamically.

We may find an example of that below:

```
<div *ngIf="myVariable"> Dynamic element </div>
```

This code uses a `NgIf` directive to render the `div` element based on the `myVariable` value. If it will be `true`, the element will be rendered, if it will be `false`, it will not be added to the DOM at all.

Another example of that approach, but using the `NgFor` directive is as follows:

```
<ul>
    <li *ngFor="let item of items"> {{ item }} </li>
</ul>
```

The code above will render `li` elements for each element in the array called `items`. So it is another example of dynamically rendered elements.

There are use cases when the standard directives are not enough. Our projects often require specific mechanisms, and custom behaviors and we have to implement our own concepts for that.

Render multiple elements based on a number

We can think about a case where we would like to display a number of items. To make the example more specific, let us say that we want to display stars in some view, and the number of stars is dynamic, provided in some variable.

The view could look as follows:

The NgFor approach

How we could implement such a view? Well, we can use the `NgFor` to generate a dynamic number of elements in the `HTML`. The HTML code implements simply a `NgFor` to render the stars, which looks as follows:

```
<p *ngFor="let item of array"> ⭐ </p>
```

The component class has to provide the variable `array`, which will be used to populate the `NgFor`. The array should have a size equal to the number of stars we want to display.

```
@Component({
  selector: 'app-stars',
  templateUrl: './stars.component.html',
  styleUrls: ['./stars.component.css'],
})
export class AppComponent {
    stars: number = 5;

  array = Array(this.stars);
}
```

Notice that the array has empty items because it was only initialized with a size of 5 and that's all we need. We don't actually use the particular items of the array — we only care about the array for `NgFor` to work.

This approach, using the `NgFor`, works as expected however it requires creating an array of a specific size, each time we want to repeat elements in the `HTML` — that is not ideal. NgFor has limited customization opportunities, so in this guide, we will propose a better implementation, that will be easier to reuse, and won't produce code duplications.

Custom directive

We are going to use structural directives to solve our main problem. So to repeatedly generate elements, as many times as we want.

We will start by creating a directive and implementing it in such a way that it displays a single element. In other words, it should add to the DOM a single view, created based on the provided template.

Let us start by creating a directive (we may use Angular CLI to generate it for us). We will call it a repeat directive

```
ng generate directive repeat
```

The empty directive class looks as follows:

```
@Directive({
  selector: '[repeat]'
})
export class RepeatDirective {
  constructor() { }
}
```

We can use it now in our HTML component's code to see what happens. So instead of the previous use of `NgFor`, we are replacing it with our directive:

```
<p *repeat> ⭐ </p>
```

What happens when we run it and check it live? Well, we won't see any stars, not even one. Why is that?

Angular wraps our `<p>` element as template, like this:

```
<ng-template [repeat]="">
    <p> ⭐ </p>
</ng-template>
```

This is a template now, so it won't be added to the DOM automatically. We need to implement that by ourselves. Let's do that!

First we need to inject the reference to the template, and the view container in the constructor:

```
@Directive({
  selector: '[repeat]',
})
export class RepeatDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainerRef: ViewContainerRef
  ) {}
}
```

With that in place, we can work on rendering the template. For now, let's don't think about rendering the template multiple times — we will do it later. Right now, focus on simple render.

Syntax for creating a view and inserting it into the DOM is actually very easy - it's a single function in `ViewContainerRef` class:

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

We are going to add this line in the `OnInit` lifecycle hook:

```
@Directive({
  selector: '[repeat]',
})
export class RepeatDirective implements OnInit {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainerRef: ViewContainerRef
  ) {}

  ngOnInit() {
    this.viewContainerRef.createEmbeddedView(this.templateRef);
  }
}
```

Such directive should add to the DOM, a single element. Let's check the live app:

The directive works as expected.

Now we can focus on adding into DOM multiple elements, but first, we need to know how many elements we should render.

For that we can use the input property, so the number will be provided when applying the directive. Input properties work the same as in the components world. So we can declare them as follows:

```
@Input() amount: number;
```

and then, provide data in HTML like this:

```
<p *repeat [amount]="3"> ⭐ </p>
```

There is an interesting trick though. If the name of the input will be the same as the directive's selector the syntax could be simplified. Take a look at such input implementation:

```
@Input('repeat') amount: number;
```

We don't want to change the actual name of the variable, because it's more explanatory this way, but we can change the input's name following the convention described above. With such code, we could modify the HTML as follows:

```
<p *repeat="3"> ⭐ </p>
```

That's better, right? It also looks more closely at the popular directives we know, such as `NgIf` or `NgFor`.

Now, when we have the number of elements we would like to render, we can actually implement that. We are going to

use a simple `for` loop to add elements to the DOM as many times as we want. The snippet below presents the complete concept:

```
@Directive({
  selector: '[repeat]',
})
export class RepeatDirective implements OnInit {
  @Input('repeat') amount: number = 0;

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

  ngOnInit() {
        // loop from 0 to the requsted number of elements
    for (let i = 0; i < this.amount; i++) {
            // add element to the DOM
      this.viewContainerRef.createEmbeddedView(this.templateRef);
    }
  }
}
```

We can check the live app, and it works exactly as we wanted:

Advantages

There are plenty of advantages of this approach. The main one are listed below:

Readability — when reading the HTML code, it's far more obvious how it will behave when using our directive:

```
<p *repeat="3"> ⭐ </p>

<!-- vs -->

<p *ngFor="let item of array"> ⭐ </p>
```

Easy to reuse — it's extremely easy to use this feature in other places. It's only a matter of applying the directive in `HTML`, and there is no need for declaring an array in the component's class as in the `NgFor` approach.

```
// no need for that:
array = Array(this.amount);
```

Simplicity — the logic is moved from component to directive, so the implementation details are hidden. The only thing

the end-user has to care about is to provide how many items need to be rendered using a simple number value! No need for creating an array of a specific size.

```
<!-- simple, easy to understand interface -->

<p *repeat="3"> ⭐ </p>

<!-- implementation hidden in the directive's code -->
```

In-depth bits

This section explains the concepts used in this guide by going in-depth into the Angular features.

Structural directives

Directives in Angular can serve many purposes. They are divided into two groups, structural directives and attribute directives. In this guide, we will focus on the first group — which groups directives that changes the DOM structure.

Structural directives define the DOM structure and that is exactly what we need. We want to add elements to the DOM.

How does a directive interact with the DOM? It uses two key elements:

- **Template** - stores information about the element to render
- **View** - the container which is our reference for interacting with the DOM

Both elements can be injected in directive's constructor, as follows:

```
@Directive({ selector: '[myDirective]'})
export class MyDirective {

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

}
```

`TemplateRef` is a reference to the template of our element. The template is generated by Angular automatically when we use the asterisk notation. For instance, when we use `NgIf`, or `MyDirective` from the snippet above as follows:

```
<p *ngIf="value"> Hello from NgIf </p>

<p *myDirective> Hello from MyDirective</p>
```

Angular will compile this in such a way:

```
<ng-template [ngIf]="value">
    <p> Hello from NgIf </p>
</ng-template>

<ng-template [myDirective]="">
    <p> Hello from MyDirective </p>
</ng-template>
```

The `<ng-template>` element is the Angular template, and the reference to that exact structure is passed via `TemplateRef`. Please note, that because Angular wraps the content in `<ng-template>` tags — it means, that it will not be added to the DOM automatically. It will be stored as a template for further use by some directives or any other code that interacts with the DOM and manages dynamic views.

The NgFor

Our `Repeat` directive has a lot of similarities with the popular `NgFor` directive so let's quickly check for differences. It's always good to know what are the advantages and disadvantages of the approach you choose to implement. You may also use this information to apply some changes to our directive based on the `NgFor` implementation.

The main difference is that `NgFor` renders elements in the `ngDoCheck` hook. It is very important for performance reasons. `NgFor` has a mechanism for tracking rendered elements by using the special function `trackBy`. So in order to efficiently render elements, Angular checks object in the Array and decide whether to re-render them or not. If identity is positive, then Angular will not clear the element and render it once again, it will simply do nothing for this particular element.

Here is an InDepth guide about `NgFor`, which explains this concept in detail.

Template variables - bonus feature

Our directive works fine, but there is one more thing that could be improved. When using the `NgFor`, we have access to some helpful variables — for instance, the index of the element. This could be useful when we would like to apply some dynamic features for specific indexes. For example, we may want to set some styling for every star, after the third one, or change the content slightly after the fifth star. So the information about the index could be quite useful. Let's implement this feature into our directive.

The function that creates a view, supports passing an optional object, which will be used to populate template variables. These variables can be used when needed. To reference the variable we will modify slightly our HTML code:

```
<p *repeat="3; let index"> ⭐ {{ index }}</p>
```

To implement the logic for keeping the index of the particular element, we add the object as a second argument, when creating a view:

```
for (let i = 0; i < this.amount; i++) {
    const parameters = {
    $implicit: i,
  };

    this.viewContainerRef.createEmbeddedView(this.templateRef, parameters);
}
```

We are using the parameter called `$implicit` to allow anyone who uses our directive to choose its own name for the template variable. In our example from above, we assigned it to the `let index`.

Summary

Structural directives are very powerful and can be extremely useful when it comes to modifying the DOM structure. We may naturally think about solving problems in our components, but often it would be easier if we put directives in the equation.

Implementing a custom directive means, that components will have less code, the HTML will be easier to read, and finally, there would be fewer code duplications. The logic in the directive will be easier to test and understand. I encourage you to use directives in your projects and have fun exploring their possibilities.

Here is a link to StackBlitz with the complete code for this guide.

If you don't remember everything, remember this: We can use structural directives to render elements and have a full control over that in just a few lines of code:

```
constructor(
  private templateRef: TemplateRef,
  private viewContainerRef: ViewContainerRef
) {}

ngOnInit() {
    this.viewContainerRef.createEmbeddedView(this.templateRef);
}
```

Enjoy!

Comments (0)

Be the first to leave a comment

JOIN THE DISCUSSION

Improve article