Angular ng-template, ng-container and ngTemplateOutlet - The Complete Guide To Angular Templates

In this post, we are going to dive into some of the more **advanced features** of Angular Core!

You have probably already come across with the ng-template Angular core directive, such as for example while using `ngIf`/else, or `ngSwitch`.

The ng-template directive and the related ngTemplateOutlet directive are very powerful Angular features that support a wide variety of advanced use cases.

These directives are frequently used with ng-container, and because these directives are designed to be used together, it will help if we learn them all in one go, as we will have more context around each directive.

Let's then see some of the more advanced use cases that these directives enable. Note: All the code for this post can be found in this Github repository.

Table Of Contents

In this post, we will be going over the following topics:

- The ng-template directive use with ngIf
- ngIf de-suggared syntax and ng-template
- ng-template template references and the **TemplateRef** injectable
- Configurable Components with Template Partial `@Inputs`
- The **ng-container** directive, when to use it?
- Dynamic Template with the **ngTemplateOutlet** custom directive
- Template outlet `@Input` Properties
- Final Combined Example
- Summary and Conclusions

Introduction to the ng-template directive

Like the name indicates, the ng-template directive represents an Angular template: this means that the content of this tag will contain part of a template, that can be then be composed together with other templates in order to form the final component template.

Angular is already using ng-template under the hood in many of the structural directives that we use all the time: `ngIf`, `ngFor` and `ngSwitch`.

Let's get started learning ng-template with an example. Here we are defining two tab buttons of a tab component (more on this later):

```
1
2  @Component({
3    selector: 'app-root',
4    template: `
```

```
      <ng-template>
          <button class="tab-button"
                  (click)="login()">{{loginText}}</button>
          <button class="tab-button"
                  (click)="signUp()">{{signUpText}}</button>
      </ng-template>
  `})
export class AppComponent {
    loginText = 'Login';
    signUpText = 'Sign Up';
    lessons = ['Lesson 1', 'Lessons 2'];

    login() {
        console.log('Login');
    }

    signUp() {
        console.log('Sign Up');
    }
}

```

The first thing that you will notice about ng-template

If you try the example above, you might be surprised to find out that this example **does not render anything** to the screen!

This is normal and it's the expected behavior. This is because with the ng-template tag we are simply defining a template, but we are not using it yet.

Let's then find an example where we can render an output, using some of the most commonly used Angular directives.

The `ng-template` directive and `ngIf`

You probably came across ng-template for the first time while implementing an if/else scenario such as for example this one:

```
<div class="lessons-list" *ngIf="lessons else loading">
  ...
</div>

<ng-template #loading>
    <div>Loading...</div>
</ng-template>
```

This is a very common use of the ngIf/else functionality: we display an alternative `loading` template while waiting for the data to arrive from the backend.

As we can see, the else clause is pointing to a template, which has the name `loading`. The name was assigned to it via a template reference, using the `#loading` syntax.

But besides that else template, the use of ngIf *also* creates a second implicit ng-template! Let's have a look at what is happening under the hood:

```
<ng-template [ngIf]="lessons" [ngIfElse]="loading">
   <div class="lessons-list">
     ...
   </div>
</ng-template>

<ng-template #loading>
    <div>Loading...</div>
</ng-template>
```

```
11
```

This is what happens internally as Angular desugars the more concise `*ngIf` structural directive syntax. Let's break down what happened during the desugaring:

- the element onto which the structural directive ngIf was applied has been moved into an ng-template
- The expression of *ngIf has been split up and applied to two separate directives, using the `[ngIf]` and `[ngIfElse]` template input variable syntax

And this is just one example, of a particular case with ngIf. But with ngFor and ngSwitch a similar process also occurs.

These directives are all very commonly used, so this means these templates are present everywhere in Angular, either implicitly or explicitly.

But based on this example, one question might come to mind:

> *How does this work if there are multiple structural directives applied to the same element?*

Multiple Structural Directives

Let's see what happens if for example we try to use `ngIf` and `ngFor` in the same element:

```
1
2  <div class="lesson" *ngIf="lessons"
3         *ngFor="let lesson of lessons">
```

```
    <div class="lesson-detail">
        {{lesson | json}}
    </div>
</div>

```

This would not work! Instead, we would get the following error message:

```
Uncaught Error: Template parse errors:
Can't have multiple template bindings on one element. Use
only one attribute
named 'template' or prefixed with *
```

This means that its not possible to apply two structural directives to the same element. In order to do so, we would have to do something similar to this:

```

<div *ngIf="lessons">
    <div class="lesson" *ngFor="let lesson of lessons">
        <div class="lesson-detail">
            {{lesson | json}}
        </div>
    </div>
</div>

```

In this example, we have moved the ngIf directive to an outer wrapping div, but in order for this to work we have to create that extra div element.

This solution would already work, but is there a way to apply a structural directive to a section of the page without having to create an extra element?

Yes and that is exactly what the `ng-container` structural directive allows us to do!

The `ng-container` directive

In order to avoid having to create that extra div, we can instead use ng-container directive:

```
<ng-container *ngIf="lessons">
    <div class="lesson" *ngFor="let lesson of lessons">
        <div class="lesson-detail">
            {{lesson | json}}
        </div>
    </div>
</ng-container>
```

As we can see, the ng-container directive provides us with an element that we can attach a structural directive to a section of the page, without having to create an extra element just for that.

There is another major use case for the ng-container directive: it can also provide a placeholder for injecting a template dynamically into the page.

Dynamic Template Creation with the `ngTemplateOutlet` directive

Being able to create template references and point them to other directives such as ngIf is just the beginning.

We can also take the template itself and instantiate it anywhere on the page, using the `ngTemplateOutlet` directive:

```
<ng-container *ngTemplateOutlet="loading"></ng-container>
```

We can see here how ng-container helps with this use case: we are using it to instantiate on the page the `loading` template that we defined above.

We are refering to the loading template via its template reference `#loading`, and we are using the `ngTemplateOutlet` structural directive to instantiate the template.

We could add as many `ngTemplateOutlet` tags to the page as we would like, and instantiate a number of different templates. The value passed to this directive can be any expression that evaluates into a template reference, more on this later.

Now that we know how to instantiate templates, let's talk about what is accessible or not by the template.

Template Context

One key question about templates is, what is visible inside them?

> *Does the template have its own separate variable scope, what variables can the template see?*

Inside the ng-template tag body, we have access to the same context variables that are visible in the outer template, such as for example the variable `lessons`.

And this is because all ng-template instances have access also to the same context on which they are embedded.

But each template can also define its own set of input variables! Actually, each template has associated a context object containing all the template-specific input variables.

Let's have a look at an example:

```
1
2  @Component({
3    selector: 'app-root',
4    template: `
5
6  <ng-template #estimateTemplate let-lessonsCounter="estimate">
7      <div> Approximately {{lessonsCounter}} lessons ...</div>
8  </ng-template>
9
10 <ng-container
11    *ngTemplateOutlet="estimateTemplate;context:ctx">
12 </ng-container>
13 `})
14 export class AppComponent {
15
16     totalEstimate = 10;
17     ctx = {estimate: this.totalEstimate};
18
19 }
20
```

Here is the breakdown of this example:

- this template, unlike the previous templates also has one input variable (it could also have several)

- the input variable is called `lessonsCounter`, and it's defined via a ng-template property using the prefix `let-`
- The variable lessonsCounter is visible inside the ng-template body, but not outside
- the content of this variable is determined by the expression that its assigned to the property `let-lessonsCounter`
- That expression is evaluated against a context object, passed to `ngTemplateOutlet` together with the template to instantiate
- This context object must then have a property named `estimate`, for any value to be displayed inside the template
- the context object is passed to `ngTemplateOutlet` via the context property, that can receive any expression that evaluates to an object

Given the example above, this is what would get rendered to the screen:

```
Approximately 10 lessons ...
```

This gives us a good overview of how to define and instantiate our own templates.

Another thing that we can also do is interact with a template programmatically at the level of the component itself: let's see how we can do that.

Template References

The same way that we can refer to the loading template using a template reference, we can also have a template injected directly into

our component using the `ViewChild` decorator:

```
@Component({
  selector: 'app-root',
  template: `
      <ng-template #defaultTabButtons>

          <button class="tab-button" (click)="login()">
            {{loginText}}
          </button>

          <button class="tab-button" (click)="signUp()">
            {{signUpText}}
          </button>

      </ng-template>
`})
export class AppComponent implements OnInit {

    @ViewChild('defaultTabButtons')
    private defaultTabButtonsTpl: TemplateRef<any>;

    ngOnInit() {
        console.log(this.defaultTabButtonsTpl);
    }

}
```

As we can see, the template can be injected just like any other DOM element or component, by providing the template reference name `defaultTabButtons` to the `ViewChild` decorator.

This means that templates are accessible also at the level of the component class, and we can do things such as for example pass them to child components!

An example of why we would want to do that is to create a more customizable component, where can pass to it not only a configuration parameter or configuration object: we can also pass *a template as an input parameter*.

Configurable Components with Template Partial `@Inputs`

Let's take for example a tab container, where we would like to give the user of the component the possibility of configuring the look and feel of the tab buttons.

Here is how that would look like, we would start by defining the custom template for the buttons in the parent component:

```
1
2  @Component({
3    selector: 'app-root',
4    template: `
5  <ng-template #customTabButtons>
6      <div class="custom-class">
7          <button class="tab-button" (click)="login()">
8            {{loginText}}
9          </button>
10         <button class="tab-button" (click)="signUp()">
11           {{signUpText}}
12         </button>
13     </div>
14 </ng-template>
15 <tab-container [headerTemplate]="defaultTabButtons"></tab-container>
16 `})
17 export class AppComponent implements OnInit {
18
19 }
20
```

And then on the tab container component, we could define an input property which is also a template named `headerTemplate`:

```
@Component({
    selector: 'tab-container',
    template: `

<ng-template #defaultTabButtons>

    <div class="default-tab-buttons">
        ...
    </div>

</ng-template>

<ng-container
  *ngTemplateOutlet="headerTemplate ? headerTemplate: defaultTabButtons">

</ng-container>

... rest of tab container component ...

`})
export class TabContainerComponent {
    @Input()
    headerTemplate: TemplateRef<any>;
}

```

A couple of things are going on here, in this final combined example. Let's break this down:

- there is a default template defined for the tab buttons, called `defaultTabButtons`
- This template will be used only if the input property `headerTemplate` remains undefined

- If the property is defined, then the custom input template passed via `headerTemplate` will be used to display the buttons instead
- the headers template is instantiated inside a ng-container placeholder, using the `ngTemplateOutlet` property
- the decision of which template to use (default or custom) is taken using a ternary expression, but if that logic was complex we could also delegate this to a controller method

The end result of this design is that the tab container will display a default look and feel for the tab buttons if no custom template is provided, but it will use the custom template if its available.

Summary and Conclusions

The core directives ng-container, ng-template and ngComponentOutlet all combine together to allow us to create highly dynamical and customizable components.

We can even change completely the look and feel of a component based on *input templates*, and we can define a template and instantiate on multiple places of the application.

And this is just one possible way that these features can be combined!