How to use ng-template & TemplateRef in Angular

4 Comments / 7 minutes of reading / March 9, 2023

← ngContainer | Angular Tutorial | NgTemplateoutlet →

In this guide, we will learn what is ng-template and TemplateRef. We also learn how it works and how Angular makes use of them in various directives like ngIf, ngFor & ngSwitch etc. We can use ng-template with ngTemplateOutlet to display the dynamic templates, which is a separate tutorial.

Table of Contents

What is ng-Template?

The `<ng-template>` is an Angular element, which contains the template. A template is an HTML snippet. The template does not render itself on DOM.

To understand let us create a new Angular Application and copy the following code to app.component.html

```
<h2>Defining a Template using ng-Template</h2>

<ng-template>
  <p> Say Hello</p>
</ng-template>
```

The above code generates the following output. The Angular does not render `Say Hello` . You won't even find it as a hidden element in the DOM.

```
//output

Defining a Template using ng-Template
```

i.e because `ng-template` only defines a template. It is our job to tell angular where & when to display it.

There are few ways you can display the template.

1. Using the `ngTemplateOutlet` directive.
2. Using the `TemplateRef` & `ViewContainerRef`

Displaying the Template

ngTemplateOutlet

The ngTemplateOutlet, is a structural directive, which renders the template.

To use this directive, first, we need to create the template and assign it to a template reference variable (sayHelloTemplate in the following template).

```
<h1>ng-template & TemplateRef</h1>

<h2>Using the ngTemplateOutlet</h2>


<ng-template #sayHelloTemplate>
  <p> Say Hello</p>
</ng-template>

```

We use the ngTemplateOutlet in the DOM, where we want to render the template.

The following code assigns the Template variable sayHelloTemplate to the ngTemplateOutlet directive using the Property Binding.

```
<ng-container *ngTemplateOutlet="sayHelloTemplate">
  This text is not displayed
</ng-container>

```

```
6
7
8  //Output
9  ng-template & TemplateRef
10 Using the ngTemplateOutlet
11 Say Hello
12
```

The content inside the `ngTemplateOutlet` directive is not displayed. It replaces it with content it gets from the `sayHelloTemplate` .

> The ngTemplateOutlet is a very powerful directive. You can use it render templates, pass data to the template, pass the template to child components, etc. You can learn all these from our ngTemplateOutlet tutorial

TemplateRef & ViewContainerRef

What is TemplateRef?

`TemplateRef` is a class and the way to reference the `ng-template` in the component or directive class. Using the TemplateRef we can manipulate the template from component code.

Remember `ng-template` is a bunch of HTML tags enclosed in a HTML element `<ng-template>`

```
1
2  <ng-template>
3    <p> Say Hello</p>
4  </ng-template>
5
```

To access the above `ng-template` in the component or directive, first, we need to assign a template reference variable. `#sayHelloTemplate` is that variable in the code below.

```
<ng-template #sayHelloTemplate>
  <p> Say Hello</p>
</ng-template>
```

Now, we can use the `ViewChild` query to inject the `sayHelloTemplate` into our component as an instance of the class `TemplateRef`.

```
@ViewChild('sayHelloTemplate', { read: TemplateRef }) sayHelloTemplate:TemplateRef<any>
```

Now, we need to tell Angular where to render it. The way to do is to use the `ViewContainerRef`.

The ViewContainerRef is also similar to TemplateRef. Both hold the reference to part of the view.

- The TemplateRef holds the reference template defined by ng-template.
- ViewContainerRef, when injected to via DI holds the reference to the host element, that hosts the component (or directive).

Once, we have ViewContainerRef, we can use the `createEmbeddedView` method to add the template to the component.

```
1
2   constructor(private vref:ViewContainerRef) {
3   }
4
5   ngAfterViewInit() {
6     this.vref.createEmbeddedView(this.sayHelloTemplate);
7   }
8
9
```

The template is appended at the bottom.

Angular makes use of `ngTemplate` extensively in its structural directives. But it hides its complexities from us.

ng-template with ngIf

You might have used ngIf a lot of times. Here is how we use it. We use an `*` before ngIf

```
1
2 <label>
3   <input [(ngModel)]="selected" type="checkbox">Select Me
4 </label>
5
6 <div *ngIf="selected">
7   <p>You are selected</p>
8 </div>
9
```

There is another way to write the above code. That is using the ng-template syntax. To do that follow these steps

1. Create a new element ng-template and bind ngIf to it
2. Use the property binding syntax [ngIf]="selected" instead of *ngIf="selected"
3. Move the div element on which ngIf is attached inside the ng-templete

```
<label>
  <input [(ngModel)]="selected" type="checkbox">Select Me
</label>

<ng-template [ngIf]="selected">
  <div>
    <p>You are selected</p>
  </div>
</ng-template>

```

The code works just like a normal *ngIf would do.

Behind the scenes, Angular converts every *ngIf to ng-template Syntax. In fact, it does so every structural directive like ngFor , ngSwitch , ngTemplateOutlet , etc

How it works

To understand how structural directives using ng-template works let us look at ttIf directive which we built in the tutorial custom structural directive. The ttIf directive is a simplified clone of *ngIf .

Create tt-if.directive.ts and add the following code. Also, remember to declare the ttIfDirective in app.module.ts

```

```

```
import { Directive, ViewContainerRef, TemplateRef, Input } from '@angular/core';

@Directive({
  selector: '[ttIf]'
})
export class ttIfDirective  {

  _ttif: boolean;

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


  @Input()
  set ttIf(condition) {
    this._ttif = condition
    this._updateView();
  }

  _updateView() {
    if (this._ttif) {
      this._viewContainer.createEmbeddedView(this.templateRef);
    }
    else {
      this._viewContainer.clear();
    }
  }

}

```

Open the app.component.html . You can use both <div *ttIf="selected"> and <ng-template [ttIf]="selected"> syntax.

```
Show/hide
<input type="checkbox" [(ngModel)]="selected">

<div *ttIf="selected">
  Using the ttIf directive via *ttIf
</div>


<ng-template [ttIf]="selected">
  <div>
    <p>Using the ttIf directive via ng-template</p>
  </div>
</ng-template>
```

app.component.ts

```
import { Component } from '@angular/core';

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

Now let us look at the directive code. We are injecting `ViewContainerRef` and `TemplateRef` instances in the constructor

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

We looked at `ViewContainerRef` in the previous section. It contains the reference to the host element that hosts our directive.

In the previous example, we used the `ViewChild` to get the reference to the template. But it is not possible here. Hence we use the Angular Dependency Injection to inject the template into our directive using the `TemplateRef` DI token.

> We use the `*` notation to tell Angular that we have a structural directive and we will be manipulating the DOM. It basically tells angular to inject the `TemplateRef`. When we attach our directive to an `ng-template`, and ask for the `TemplateRef` in the constructor, the Angular injects the reference to the template enclosed by the `ng-template`.

The Template is inserted into the DOM when the condition is true. We do that using the `createEmbeddedView` method of the `ViewContainerRef`. The `clear` removes the template from the DOM

```
1
2 this._viewContainer.createEmbeddedView(this.templateRef);
3
```

Multiple Structural Directives

You cannot assign multiple Structural Directives on a single `ng-template`.

For Example, the `ngIf` & `ngFor` on same `div`, will result in an an `Template parse errors`

```
1
2 <div *ngIf="selected"
3     *ngFor="let item of items">
```

```
    {{item.name}}
</div>

```

```

Uncaught Error: Template parse errors:

Can't have multiple template bindings on one element.
Use only one attribute named 'template' or prefixed with *

```

You can use ng-container to move one directive to enclose the other as shown below.

```

<ng-container *ngIf="selected">
  <div *ngFor="let item of items">
     {{item.name}}
  </div>
</ng-container>

```

ng-template with ngIf, then & else

The following code shows the ng-template using the ngIf, then & else example.

Here we use the ng-template specify the template for the then & else clause. We use the template reference variable to get the reference to those blocks.

In the *ngIf condition we specify the template to render by pointing to the template variable to the then & else condition.

```
<h2>Using ngTemplate with ngIf then & else</h2>

<div *ngIf="selected; then thenBlock1 else elseBlock1">
  <p>This content is not shown</p>
</div>

<ng-template #thenBlock1>
  <p>content to render when the selected is true.</p>
</ng-template>

<ng-template #elseBlock1>
  <p>content to render when selected is false.</p>
</ng-template>
```

The above ngif can be written using the ng-template syntax.

```
<ng-template [ngIf]="selected" [ngIfThen]="thenBlock2" [ngIfElse]="elseBlock2">
  <div>
    <p>This content is not shown</p>
  </div>
</ng-template>

<ng-template #thenBlock2>
  <p>content to render when the selected is true.</p>
</ng-template>

<ng-template #elseBlock2>
  <p>content to render when selected is false.</p>
</ng-template>
```

ng-template with ngFor

The following ngFor Directive.

```
<ul>
  <li *ngFor="let movie of movies; let i=index; let even=even;trackBy: trackById">
    {{ movie.title }} - {{movie.director}}
  </li>
</ul>
```

Is written as follows using the ng-template syntax

```
<ul>
<ng-template
  ngFor let-movie [ngForOf]="movies"
  let-i="index"
  let-even="even"
  [ngForTrackBy]="trackById">

  <li>
    {{ movie.title }} - {{movie.director}}
  </li>

</ng-template>
</ul>
```

ng-template with ngSwitch

The following is the example of ngSwitch.

```
<input type="text" [(ngModel)]="num">

<div [ngSwitch]="num">
  <div *ngSwitchCase="'1'">One</div>
  <div *ngSwitchCase="'2'">Two</div>
  <div *ngSwitchCase="'3'">Three</div>
  <div *ngSwitchCase="'4'">Four</div>
  <div *ngSwitchCase="'5'">Five</div>
  <div *ngSwitchDefault>This is Default</div>
</div>
```

The above example using the `ng-template` with ngSwitch. Note that ngSwitch is not a structural directive but `ngSwitchCase` & `ngSwitchDefault` are.

```
<div [ngSwitch]="num">
  <ng-template [ngSwitchCase]="'1'">
    <div>One</div>
  </ng-template>
  <ng-template [ngSwitchCase]="'2'">
    <div>Two</div>
  </ng-template>
  <ng-template [ngSwitchCase]="'3'">
    <div>Three</div>
  </ng-template>
  <ng-template [ngSwitchCase]="'4'">
    <div>Four</div>
  </ng-template>
  <ng-template [ngSwitchCase]="'5'">
    <div>Five</div>
  </ng-template>
  <ng-template ngSwitchDefault>
    <div>This is default</div>
  </ng-template>
</div>

```

Reference

1. NgTemplateOutlet
2. Structural directives

Source Code

1. ngtemplateoutlet
2. ViewContainerRef
3. ttIf Directive

Read More