

Stay safe, friends. Learn to code from home. [Use our free 2,000 hour curriculum.](#)

29 OCTOBER 2018 / [#TECH](#)

Everything you need to know about ng-template, ng-content, ng-container, and *ngTemplateOutlet in Angular



Prateek Mishra

Read [more posts](#) by this author.

`<ng-template>`



`<ng-container>`

`*ngTemplateOutlet`

`<ng-content>`

It was one of those days when I was busy working on new features for my office project. All a sudden, something caught my attention:

```
<!--bindings={  
  "ng-reflect-ng-if": "[object Object]"  
}-->  
<div _ngcontent-c0>Mr. Nice</div>
```

Final rendered DOM in Angular

While inspecting the DOM I saw the `ngcontent` being applied on elements by Angular. Hmm... if

they contain the elements in the final DOM, then what is the use of `<ng-container>`? At that time I got confused between `<ng-container>` and `<ng-content>`.

In the quest to know the answers to my questions I discovered the concept of `<ng-template>`. To my surprise, there was also `*ngTemplateOutlet`. I started my journey seeking clarity about two concepts but now I had four of them, sounding nearly the same!

Have you ever been to this situation? If yes, then you are in the right place. So without further ado let's take them one by one.

1. `<ng-template>`

As the name suggests the `<ng-template>` is a template element that Angular uses with structural directives (`*ngIf`, `*ngFor`, `[ngSwitch]` and custom directives).

These template elements only work in the presence of structural directives. Angular wraps the host element (to which the directive is applied) inside `<ng-template>` and consumes the `<ng-template>` in the finished DOM by replacing it with diagnostic comments.

Consider a simple example of `*ngIf`:



```
<div *ngIf="shouldSayHello" class="hello-world">Hello World</div>

<!-- Converted element -->
<ng-template [ngIf]="shouldSayHello">
  <div class="hello-world">Hello World</div>
</ng-template>
```

Example 1- Angular process of interpreting structural directives

Shown above is the Angular interpretation of `*ngIf`. Angular puts the host element to which the directive is applied within `<ng-template>` and keeps the host as it is. The final DOM is similar to what we have seen at the beginning of this article:

```
▼ <body>
  ▼ <app-root _ngghost-c0 ng-version="6.1.10">
    <!--bindings={
      "ng-reflect-ng-if": "true"
    }-->
    <div _ngcontent-c0 class="hello-world">Hello World</div>
  </app-root>
```

Example 1- Final rendered DOM

Usage:

We have seen how Angular uses `<ng-template>` but what if we want to use it? As these elements work only with a structural directive, we can write as:

```
<ng-template *ngIf="home">Go Home!</ng-template>
```

Example 2- Using <ng-template>

Here `home` is a `boolean` property of the component set to `true` value. The output of the above code in DOM:

```
▶ <head>...</head>
▼ <body>
  ▼ <app-root _ngghost-c0 ng-version="6.1.10">
    <!--bindings={
      "ng-reflect-ng-if": "true"
    }-->
    <!------>
  </app-root>
```

Example 2- Final rendered DOM

Nothing got rendered! :(

But why can't we see our message even after using `<ng-template>` correctly with a structural

directive?

This was the expected result. As we have already discussed, Angular replaces the `<ng-template>` with diagnostic comments. No doubt the above code would not generate any error, as Angular is perfectly fine with your use case. You would never get to know what exactly happened behind the scenes.

Let's compare the above two DOMs that were rendered by Angular:

```
▼ <body>
  ▼ <app-root _ngghost-c0 ng-version="6.1.10">
    <!--bindings={
      "ng-reflect-ng-if": "true"
    }-->
    <div _ngcontent-c0 class="hello-world">Hello World</div>
  </app-root>
```

```
▶ <head>...</head>
▼ <body>
  ▼ <app-root _ngghost-c0 ng-version="6.1.10">
    <!--bindings={
      "ng-reflect-ng-if": "true"
    }-->
    <!-->
  </app-root>
```

Example 1 vs Example 2

If you watch closely, there is one **extra comment tag** in the final DOM of **Example 2**. The code that Angular interpreted was:



```
<ng-template *ngIf="home">Go Home!</ng-template>
```

```
<!-- Converted element -->
```

```
<ng-template [ngIf]="home">
```

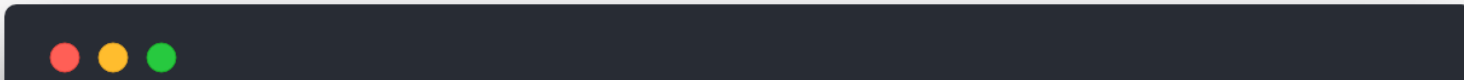
```
  <ng-template>Go Home!</ng-template>
```

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

Angular interpretation process for **Example 2**

Angular wrapped up your host `<ng-template>` within another `<ng-template>` and converted not only the outer `<ng-template>` to diagnostic comments but also the inner one! This is why you could not see any of your message.

To get rid of this there are two ways to get your desired result:



```
<!-- Method 1 -->
<ng-template [ngIf]="home">Go Home!</ng-template>

<!-- Method 2 -->
<ng-template *ngIf="home then goHome"></ng-template>
<ng-template #goHome>Go Home!</ng-template>
```

Correct usage of <ng-template>

Method 1:

In this method, you are providing Angular with the de-sugared format that needs no further processing. This time Angular would only convert `<ng-template>` to comments but leaves the content inside it untouched (they are no longer inside any `<ng-template>` as they were in the previous case). Thus, it will render the content correctly.

To know more about how to use this format with other structural directives refer to this [article](#).

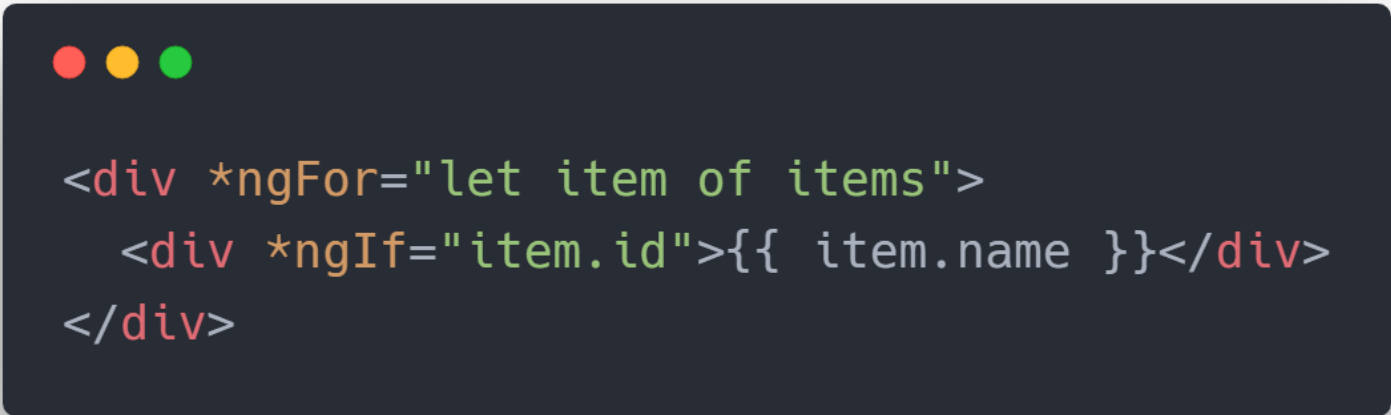
Method 2:

This is a quite unseen format and is seldom used (using two sibling `<ng-template>`). Here we are giving a template reference to the `*ngIf` in its `then` to tell it which template should be used if the condition is true.

Using multiple `<ng-template>` like this is not advised (you could use `<ng-container>` instead) as this is not what they are meant for. They are used as a container to templates that can be reused at multiple places. We will cover more on this in a later section of this article.

2. `<ng-container>`

Have you ever written or seen code resembling this:



```
<div *ngFor="let item of items">
  <div *ngIf="item.id">{{ item.name }}</div>
</div>
```

Example 1

The reason why many of us write this code is the inability to use multiple structural directives on a single host element in Angular. Now this code works fine but it introduces several extra empty `<div>` in the DOM if `item.id` is a falsy value which might not be required.

```
▶ <div _ngcontent-c0>...</div>
▼ <div _ngcontent-c0>
  <!--bindings={
    "ng-reflect-ng-if": "2"
  }-->
  <div _ngcontent-c0>Item 2</div>
</div>
▼ <div _ngcontent-c0>
  <!--bindings={
    "ng-reflect-ng-if": null
  }-->
</div>
▼ <div _ngcontent-c0>
  <!--bindings={}-->
</div>
▼ <div _ngcontent-c0>
  <!--bindings={}-->
</div>
▼ <div _ngcontent-c0>
  <!--bindings={}-->
</div>
```

```
</div>  
▼<div _ngcontent-c0>  
  <!--bindings={  
    "ng-reflect-ng-if": "7"  
  }-->  
  <div _ngcontent-c0>Item 7</div>  
</div>
```

Example 1- Final rendered DOM

One may not be concerned for a simple example like this but for a huge application that has a complex DOM (to display tens of thousands of data) this might become troublesome as the elements might have listeners attached to them which will still be there in the DOM listening to events.

What's even worse is the level of nesting that you have to do to apply your styling (CSS)!





Image from: [Inside Unbounce](#)

No worries, we have `<ng-container>` to the rescue!

The Angular `<ng-container>` is a grouping element that doesn't interfere with styles or layout because Angular *doesn't put it in the DOM*.

So if we write our **Example 1** with `<ng-container>`:



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

Example 1 with <ng-container>

We get the final DOM as:


```

<!--bindings={
  "ng-reflect-ng-if": "2"
}-->
<div _ngcontent-c0>Item 2</div>
<!-->
<!-->
<!--bindings={
  "ng-reflect-ng-if": null
}-->
<!-->
<!-->
<!--bindings={}-->
<!-->
<!-->
<!--bindings={}-->
<!-->
<!-->

```

```
<!--bindings={}-->
<!-->
<!-->
<!--bindings={
  "ng-reflect-ng-if": "7"
}-->
<div _ngcontent-c0>Item 7</div>
<!-->
<!-->
<!--bindings={
  "ng-reflect-ng-if": "8"
```

Final rendered DOM with `<ng-container>`


See we got rid of those empty `<div>` s. We should use `<ng-container>` when we just want to apply multiple structural directives without introducing any extra element in our DOM.

For more information refer to the [docs](#). There's another use case where it is used to inject a template dynamically into a page. I'll cover this use case in the last section of this article.

3. `<ng-content>`


They are used to create configurable components. This means the components can be configured depending on the needs of its user. This is well known as **Content Projection**. Components that are used in published libraries make use of `<ng-content>` to make themselves configurable.

Consider a simple `<project-content>` component:



```
<!-- project-content.html -->
<div class="heading">
  <h1>Welcome to Content Projection</h1>
</div>
<div class="body">
  <div>Some content...</div>
</div>
<div class="footer">
  <ng-content></ng-content>
</div>
```

Example 1- <project-content> definition



```
<project-content>  
  <div>This is custom footer...</div>  
</project-content>
```

Content Projection with <project-content> component

The HTML content passed within the opening and closing tags of `<project-content>` component is the content to be projected. This is what we call **Content Projection**. The content will be rendered inside the `<project-content>` within the component. This allows the consumer of `<project-content>`

inside the `<ng-content>` within the component. This allows the consumer of `<project-content>` component to pass any custom footer within the component and control *exactly* how they want it to be rendered.

Multiple Projections:

What if you could decide which content should be placed where? Instead of every content projected inside a single `<ng-content>`, you can also control how the contents will get projected with the `select` attribute of `<ng-content>`. It takes an element selector to decide which content to project inside a particular `<ng-content>`.

Here's how:




```
<div class="heading">
  <ng-content select="h1"></ng-content>
</div>
<div class="body">
  <ng-content select="div"></ng-content>
</div>
<div class="footer">
  <ng-content></ng-content>
</div>
```

Example 2- Multi-content projection with updated <project-content>

We have modified the `<project-content>` definition to perform Multi-content projection. The `select` attribute selects the type of content that will be rendered inside a particular `<ng-content>`. Here we have first `select` to render header `h1` element. If the projected content has no `h1` element it won't render anything. Similarly the second `select` looks for a `div`. The rest of the

element it won't render anything. Similarly, the second `<select>` looks for a `<div>`. The rest of the content gets rendered inside the last `<ng-content>` with no `select`.

Calling the component will look like:



```
<project-content>
  <h1>Header for first ng-content</h1>
  <div>Div for second ng-content</div>
  <span>Span for third ng-content</span>
</project-content>
```

Example 2- Calling of `<project-content>` component in parent component

4. *ngTemplateOutlet

...They are used as a container to templates that can be reused at multiple places. We will cover more on this in a later section of this article.

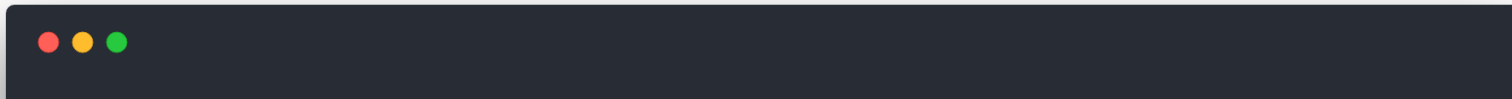
...There's another use case where it is used to inject a template dynamically into a page. I'll cover this use case in the last section of this article.

This is the section where we will discuss the above two points mentioned before. `*ngTemplateOutlet` is used for two scenarios – to insert a common template in various sections of a view irrespective of loops or condition and to make a highly configured component.

Template reuse:

Consider a view where you have to insert a template at multiple places. For example, a company logo to be placed within a website. We can achieve it by writing the template for the logo once and reusing it everywhere within the view.

Following is the code snippet:




```

<div>
  <ng-container *ngTemplateOutlet="companyLogoTemplate"></ng-container>
  <h1>Company History</h1>
  <div>{{companyHistory}}</div>
</div>
<form (ngSubmit)="onSubmit()">
  <ng-container *ngTemplateOutlet="companyLogoTemplate"></ng-container>
  <h1>User info</h1>
  <label>Name:</label><input type="text" [(ngModel)]="userName" />
  <label>Account ID:</label><input type="text" [(ngModel)]="accountId" />
  <button>Submit</button>
</form>
<div class="footer">
  <ng-container *ngTemplateOutlet="companyLogoTemplate"></ng-container>
</div>

<ng-template #companyLogoTemplate>
  <div class="companyLogo">
    <img [src]="logoSourceUrl">
    <label>The ACME company, {{employeeCount}} people working for you!</label>
  </div>
</ng-template>

```

Example 1- Template reuse

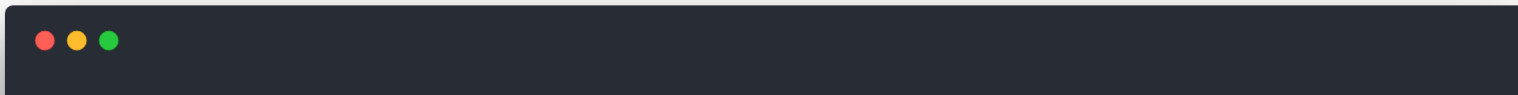
As you can see we just wrote the logo template once and used it three times on the same page with

single line of code!

`*ngTemplateOutlet` also accepts a context object which can be passed to customize the common template output. For more information about the context object refer to the official [docs](#).

Customizable components:

The second use case for `*ngTemplateOutlet` is highly customized components. Consider our previous example of `<project-content>` component with some modifications:



```

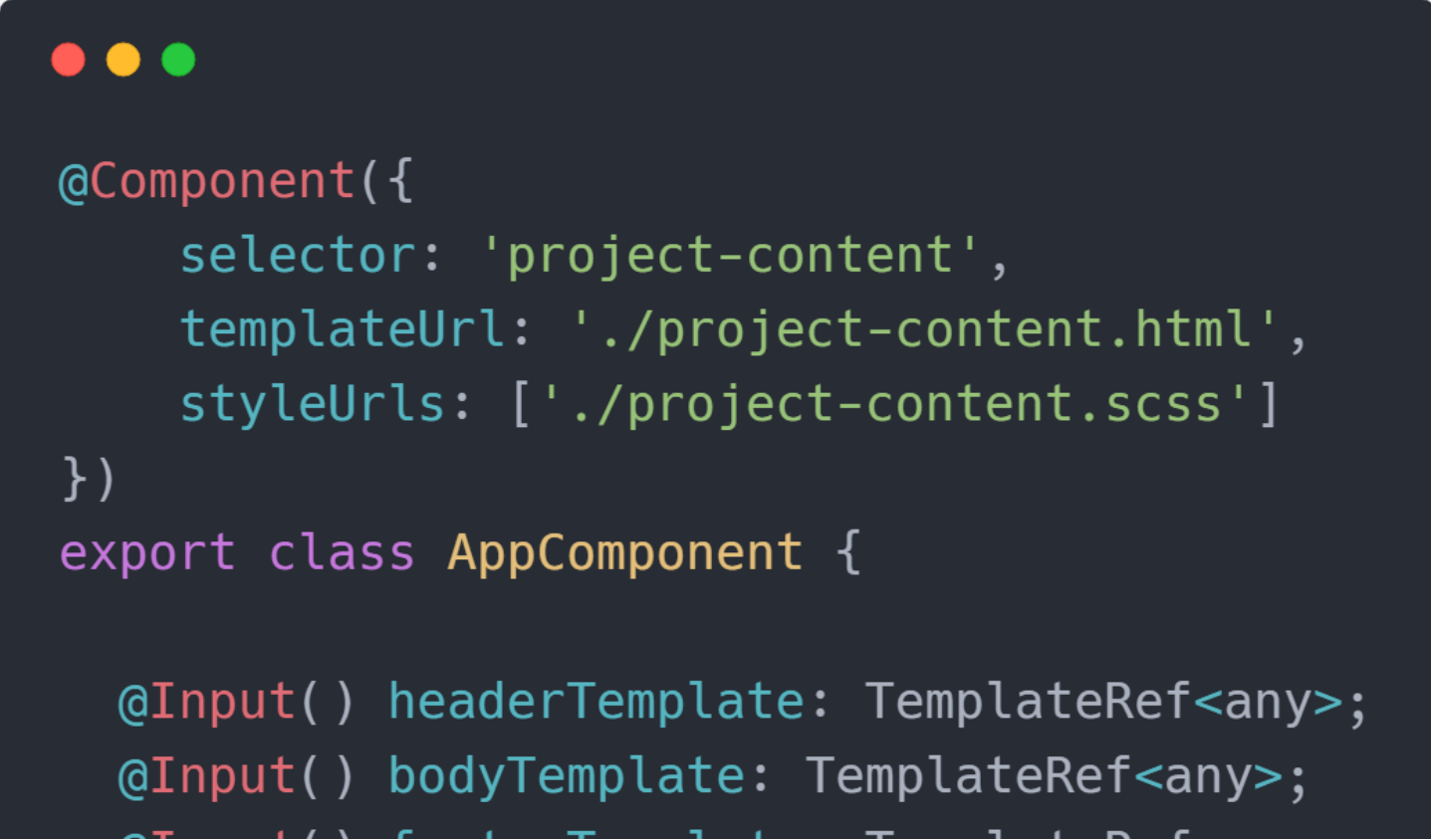
<div class="heading">
  <ng-container *ngTemplateOutlet="headerTemplate ? headerTemplate :
    defaultHeader"></ng-container>
</div>
<div class="body">
  <ng-container *ngTemplateOutlet="bodyTemplate ? bodyTemplate : defaultBody" >
    </ng-container>
</div>
<div class="footer">
  <ng-container *ngTemplateOutlet="footerTemplate ? footerTemplate :
    defaultFooter"></ng-container>
</div>

<ng-template #defaultHeader>Some header...</ng-template>
<ng-template #defaultBody>Some body...</ng-template>
<ng-template #defaultFooter>Some footer...</ng-template>

```

Example 2- Making customizable component, project-content.html

Above is the modified version of `<project-content>` component which accepts three input properties— `headerTemplate`, `bodyTemplate`, `footerTemplate`. Following is the snippet for `project-content.ts`:



```
@Component({
  selector: 'project-content',
  templateUrl: './project-content.html',
  styleUrls: ['./project-content.scss']
})
export class AppComponent {

  @Input() headerTemplate: TemplateRef<any>;
  @Input() bodyTemplate: TemplateRef<any>;
  @Input() footerTemplate: TemplateRef<any>;
```

```
@Input() footerTemplate: TemplateRef<any>;  
  
constructor() {}  
}
```

Example 2- Making customizable component, project-content.ts

What we are trying to achieve here is to show header, body and footer as received from the parent component of `<project-content>`. If any one of them is not provided, our component will show the default template in its place. Thus, creating a highly customized component.

To use our recently modified component:

```
<!-- Header -->
<ng-template #customHeader>
  <div class="custom-class">Some content...</div>
</ng-template>

<!-- Body -->
<ng-template #customBody>
  <div class="custom-class">Some content...</div>
</ng-template>

<!-- Footer -->
<ng-template #customFooter>
  <div class="custom-class">Some content...</div>
</ng-template>

<project-content [headerTemplate]="customHeader" [bodyTemplate]="customBody"
[footerTemplate]="customFooter"></project-content>
```

Example 2- Using newly modified component <project-content>

This is how we are going to pass the template refs to our component. If any one of them is not

passed then the component will render the default template.

ng-content vs. *ngTemplateOutlet

They both help us to achieve highly customized components but which to choose and when?

It can clearly be seen that `*ngTemplateOutlet` gives us some more power of showing the default template if none is provided.

This is not the case with `ng-content`. It renders the content as is. At the maximum you can split the content and render them at different locations of your view with the help of `select` attribute. You cannot conditionally render the content within `ng-content`. You have to show the content that is received from the parent with no means to make decisions based on the content.

However, the choice of selecting among the two completely depends on your use case. At least now we have a new weapon `*ngTemplateOutlet` in our arsenal which provides more control on the content in addition to the features of `ng-content` !

If this article was helpful, [tweet it.](#)

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers. [Get started](#)

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) nonprofit organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public. We also have thousands of freeCodeCamp study groups around the world.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can [make a tax-deductible donation here](#).

Trending Guides

[SQL Interview Questions](#)

[Statistical Significance](#)

[SQL Queries](#)

[What is Blockchain?](#)

[Full Stack Developer](#)

[CSS Flexbox](#)

[Linux Commands](#)

[JavaScript Map](#)

[What is HTTPS?](#)

[Python List Append](#)

[JavaScript Substring](#)

[What Does a VPN Do?](#)

[Docker Remove Image](#)

[Tar GZ](#)

[What is a CSV File?](#)

[Correlation VS Causation](#)

[Permutation VS Combination](#)

[Computer Programming](#)

[JWT](#)

[How to Find a Square Root](#)

[What is Chromium?](#)

[Smoke Testing](#)

[Clear History](#)

[Incognito Mode](#)

[Linux Add User](#)

[MD5 Hash](#)

[What is Cached Data?](#)

[Completing the Square](#)

[Error 403 Forbidden](#)

[CSS Inline Style](#)

Our Nonprofit

[About](#)

[Alumni Network](#)

[Open Source](#)

[Shop](#)

[Support](#)

[Sponsors](#)

[Academic Honesty](#)

[Code of Conduct](#)

[Privacy Policy](#)

[Terms of Service](#)

[Copyright Policy](#)