



Directivas estructurales

Esta guía analiza cómo Angular manipula el DOM con **directivas estructurales** y cómo puede escribir sus propias directivas estructurales para hacer lo mismo.

Pruebe el [ejemplo en vivo](#) / ejemplo de [descarga](#) .

¿Qué son las directivas estructurales?

Las directivas estructurales son responsables del diseño HTML. Ellos dan forma o remodelan la *estructura* del DOM, normalmente agregando, quitando o manipulando elementos.

Al igual que con otras directivas, se aplica una directiva estructural a un *elemento host*. La directiva entonces hace lo que se supone que tiene que hacer con ese elemento anfitrión y sus descendientes.

Las directivas estructurales son fáciles de reconocer. Un asterisco (*) precede al nombre del atributo de directiva como en este ejemplo.

```
src/app/app.component.html (ngif)
```

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

Sin corchetes. Sin paréntesis. Sólo tienes que poner una cuerda. `*ngIf`

Aprenderá en esta guía que el **asterisco (*)** es una **notación de conveniencia** y la cadena es un **microsyntax** en lugar de la **expresión** de plantilla habitual. Angular desugars esta notación en un marcado que rodea el elemento anfitrión y sus descendientes. Cada directiva estructural hace algo diferente con esa plantilla. `<ng-template>`

Tres de las directivas estructurales comunes incorporadas: **NgIf**, **NgFor** **NgSwitch...** : se describen en la guía [De sintaxis de plantilla](#) y se ven en ejemplos a lo largo de la documentación angular. Este es un ejemplo de ellos en una plantilla:

src/app/app.component.html (incorporado)

```
<div *ngIf="hero" class="name">{{hero.name}}</div>

<ul>
  <li *ngFor="let hero of heroes">{{hero.name}}</li>
</ul>

<div [ngSwitch]="hero?.emotion">
  <app-happy-hero    *ngSwitchCase="'happy'"    [hero]="hero"></app-happy-hero>
  <app-sad-hero      *ngSwitchCase="'sad'"      [hero]="hero"></app-sad-hero>
  <app-confused-hero *ngSwitchCase="'confused'" [hero]="hero"></app-confused-
hero>
  <app-unknown-hero *ngSwitchDefault           [hero]="hero"></app-unknown-
hero>
</div>
```

Esta guía no repetirá cómo *usarlas*. Pero explica *cómo funcionan* y cómo [escribir su propia](#) directiva estructural.

ORTOGRAFÍA DE LA DIRECTIVA

A lo largo de esta guía, verá una directiva escrita tanto en *UpperCamelCase* como en *lowerCamelCase*. Ya has visto y . Hay una razón. se refiere a la *clase* de directiva; hace referencia al nombre de *atributo* de la directiva .`NgIf``ngIf``NgIf``ngIf`

Una *clase* de directiva se escribe en *UpperCamelCase* (). El nombre de *atributo* de una directiva se escribe en *lowerCamelCase* (). La guía hace referencia a la *clase* de directiva cuando se habla de sus propiedades y lo que hace la directiva. La guía hace referencia al nombre del *atributo* al describir cómo se aplica la directiva a un elemento de la plantilla HTML.`NgIf``ngIf`

Hay otros dos tipos de directivas angulares, que se describen ampliamente en otros lugares: (1) componentes y (2) directivas de atributo.

Un *componente* administra una región de HTML a la manera de un elemento HTML nativo. Técnicamente es una directiva con una plantilla.

Una directiva de *atributo* cambia la apariencia o el comportamiento de un elemento, componente u otra directiva. Por ejemplo, la directiva `NgStyle` integrada cambia varios estilos de elemento al mismo tiempo.

Puede aplicar muchas directivas de *atributo* a un elemento host. Solo puede *aplicar una* directiva *estructural* a un elemento host.

Estudio de caso NgIf

`NgIf` es la directiva estructural más simple y la más fácil de entender. Toma una expresión booleana y hace que aparezca o desaparezca un fragmento completo del DOM.

src/app/app.component.html (ngif=true)

```
<p *ngIf="true">
  Expression is true and ngIf is true.
  This paragraph is in the DOM.
</p>
<p *ngIf="false">
  Expression is false and ngIf is false.
  This paragraph is not in the DOM.
</p>
```

La directiva no oculta elementos con CSS. Los agrega y quita físicamente del DOM. `NgIf`

```
<p _ngcontent-c0>
  Expression is true and ngIf is true.
  This paragraph is in the DOM.
</p>
<!--bindings=
  "ng-reflect-ng-if": "false"
-->
```

El párrafo superior está en el DOM. El párrafo inferior en desuso no lo es; en su lugar hay un comentario sobre "vinculaciones" (más sobre eso [más adelante](#)).

Cuando la condición es false, quita su elemento host del DOM, lo separa de los eventos DOM (los datos adjuntos que realizó), separa el componente de la detección de cambios angulares y lo destruye. Los nodos de componente y DOM se pueden recopilar elementos no utilizados y liberar memoria. [NgIf](#)

¿Por qué *eliminar* en lugar de *ocultar*?

Una directiva podría ocultar el párrafo no deseado en su lugar estableciendo su estilo en [.displaynone](#)

src/app/app.component.html (pantalla-ninguno)

```
<p [style.display]='block'>
  Expression sets display to "block".
  This paragraph is visible.
</p>
<p [style.display]='none'>
  Expression sets display to "none".
  This paragraph is hidden but still in the DOM.
</p>
```

Mientras es invisible, el elemento permanece en el DOM.

```
<p _ngcontent-fwv-0 style="display: block;">
  Expression sets display to "block" .
  This paragraph is visible.
</p>
<p _ngcontent-fwv-0 style="display: none;">
  "
  Expression sets display to "none" .
  This paragraph is hidden but still in the DOM.
  "
</p>
```

La diferencia entre ocultar y eliminar no importa para un párrafo simple. Importa cuando el elemento host está asociado a un componente de uso intensivo de recursos. El comportamiento de un componente de este tipo continúa incluso cuando está oculto. El componente permanece asociado a su elemento DOM. Sigue escuchando eventos. Angular sigue buscando cambios que podrían afectar a los enlaces de datos. Sea lo que sea lo que el componente estaba haciendo, sigue haciendo.

Aunque invisible, el componente (y todos sus componentes descendientes) ata los recursos. La carga de rendimiento y memoria puede ser sustancial, la capacidad de respuesta puede degradarse y el usuario no ve nada.

En el lado positivo, mostrar el elemento de nuevo es rápido. El estado anterior del componente se conserva y está listo para mostrarse. El componente no se vuelve a inicializar, una operación que podría ser costosa. Así que esconderse y mostrar es a veces lo correcto.

Pero en ausencia de una razón convincente para mantenerlos alrededor, su preferencia debe ser eliminar elementos DOM que el usuario no puede ver y recuperar los recursos no utilizados con una directiva estructural como `.NgIf`

Estas mismas consideraciones se aplican a todas las directivas estructurales, ya sean integradas o personalizadas. Antes de aplicar una directiva estructural, es posible que desee hacer una pausa por un momento para tener en cuenta las consecuencias de agregar y quitar elementos y de crear y destruir componentes.

El prefijo de asterisco (*)

Seguramente se dio cuenta del prefijo de asterisco (*) al nombre de la directiva y se preguntó por qué es necesario y qué hace.

Aquí se muestra el nombre del héroe si existe: `*ngIfhero`

src/app/app.component.html (asterisco)

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

El asterisco es "azúcar sintáctico" para algo un poco más complicado. Internamente, Angular traduce el *atributo* en un *elemento*, envuelto alrededor del elemento host, así: `*ngIf<ng-template>`

src/app/app.component.html (plantilla de ngif)

```
<ng-template [ngIf]="hero">
  <div class="name">{{hero.name}}</div>
</ng-template>
```

- La directiva se movió al elemento donde se convirtió en un enlace de propiedad, `*ngIf<ng-template>` `[ngIf]`
- El resto del atributo `class`, incluido su atributo `class`, se movió dentro del elemento. `<div>` `<ng-template>`

El primer formulario no se representa realmente, solo el producto terminado termina en el DOM.

```
<!--bindings={
  "ng-reflect-ng-if": "[object Object]"
-->
```

```

}-->
<div _ngcontent-c0>Mr. Nice</div>

```

Angular consumió el contenido durante su representación real y reemplazó el con un comentario de diagnóstico. `<ng-template>` `<ng-template>`

Las directivas `NgFor` y `NgSwitch...` siguen el mismo patrón.

Dentro de **ngFor*

Angular transforma la sintaxis de asterisco (*) de forma similar a *element*. `*ngFor` `<ng-template>`

Aquí hay una aplicación completa de , escrito en ambos sentidos: `NgFor`

src/app/app.component.html (dentro de ngfor)

```

<div *ngFor="let hero of heroes; let i=index; let odd=odd; trackBy: trackById"
[class.odd]="odd">
  ({{i}}) {{hero.name}}
</div>

<ng-template ngFor let-hero [ngForOf]="heroes" let-i="index" let-odd="odd"
[ngForTrackBy]="trackById">
  <div [class.odd]="odd">({{i}}) {{hero.name}}</div>
</ng-template>

```

Esto es manifiestamente más complicado que y con razón. La directiva tiene más características, tanto obligatorias como opcionales, que las que se muestran en esta guía. Como mínimo necesita una variable de bucle () y una lista (). `ngIf` `NgFor` `NgIf` `NgFor` `let hero` `heroes`

Estas características se habilitan en la cadena asignada a , que se escriben en el [microsyntax](#) de Angular. `ngFor`

Todo *lo que está fuera de* la cadena permanece con el elemento host (el) a medida que se mueve dentro del archivo . En este ejemplo, las estancias en el archivo . `ngFor` `<div>` `<ng-template>` `[class.odd]="odd"` `<div>`

Microsyntax

El microsyntax angular le permite configurar una directiva en una cadena compacta y amigable. El analizador microsyntax traduce esa cadena en atributos en: `<ng-template>`

- La palabra clave declara una *variable de entrada de plantilla* a la que se hace referencia dentro de la plantilla. Las variables de entrada de este ejemplo son `, , y`. El analizador traduce `, , y` en variables denominadas `, , y`. `let hero i odd let hero let i let odd let-hero let-i let-odd`
- El analizador microsyntax title-cases todas las directivas y las prefija con el nombre de atributo de la directiva, como `.`. Por ejemplo, las propiedades de entrada `y ,` se convierten en `y ,` respectivamente. Así es como la directiva aprende que la lista es `y` y la función track-by es `.ngFor ngFor of trackBy ngForOf ngForTrackBy heroes trackById`
- A medida que la directiva recorre la lista, establece y restablece las propiedades de su propio objeto de *contexto*. Estas propiedades pueden incluir, pero no se limitan a `, , y` una propiedad especial denominada `.NgFor index odd $implicit`
- Las variables `y` y las variables se definieron como `y`. Angular los establece en el valor actual del contexto y las propiedades. `let-i let-odd let i=index let odd=odd index odd`
- No se especificó la propiedad context para `.`. Su origen previsto es implícito. Angular se establece en el valor de la propiedad del contexto, que se ha inicializado con el héroe para la iteración actual. `let-hero let-hero $implicit NgFor`
- La guía de la API de `NgFor` describe propiedades de directiva y propiedades de contexto adicionales. `NgFor`
- La directiva implementa `.`. Obtenga más información sobre las propiedades de directiva y las propiedades de contexto adicionales en la referencia de la [API de NgForOf](#). `NgForOf NgForOf NgForOf`

Writing your own structural directives

These microsyntax mechanisms are also available to you when you write your own structural directives. For example, microsyntax in Angular allows you to write instead of `.`. The following sections provide detailed information on constraints, grammar, and translation of microsyntax. `<div`

```
*ngFor="let item of items">{{item}}</div><ng-template ngFor let-item
[ngForOf]="items"><div>{{item}}</div></ng-template>
```

Constraints

Microsyntax must meet the following requirements:

- It must be known ahead of time so that IDEs can parse it without knowing the underlying semantics of the directive or what directives are present.
- It must translate to key-value attributes in the DOM.

Grammar

When you write your own structural directives, use the following grammar:

```
*:prefix="( :let | :expression ) (';' | ',')? ( :let | :as | :keyExp )"
```

The following tables describe each portion of the microsyntax grammar.

<code>prefix</code>	HTML attribute key
<code>key</code>	HTML attribute key
<code>local</code>	local variable name used in the template
<code>export</code>	value exported by the directive under a given name
<code>expression</code>	standard Angular expression
<code>keyExp = :key ":"? :expression ("as" :local)? ";"?</code>	
<code>let = "let" :local "=" :export ";"?</code>	
<code>as = :export "as" :local ";"?</code>	

Translation

A microsyntax is translated to the normal binding syntax as follows:

Microsyntax	Translation
<code>prefix</code> and naked <code>expression</code>	<code>[prefix]="expression"</code>
<code>keyExp</code>	<code>[prefixKey] "expression" (let-prefixKey="export")</code> Notice that the is added to the <code>prefix</code> <code>key</code>
<code>let</code>	<code>let-local="export"</code>

Microsyntax examples

The following table demonstrates how Angular desugars microsyntax.

Microsyntax	Desugared
<code>*ngFor="let item of [1,2,3]"</code>	<code><ng-template ngFor let-item [ngForOf]="[1,2,3]"></code>
<code>*ngFor="let item of [1,2,3] as items; trackBy: myTrack; index as i"</code>	<code><ng-template ngFor let-item [ngForOf]="[1,2,3]" let-items="ngForOf" [ngForTrackBy]="myTrack" let-i="index"></code>
<code>*ngIf="exp"</code>	<code><ng-template [ngIf]="exp"></code>
<code>*ngIf="exp as value"</code>	<code><ng-template [ngIf]="exp" let-value="ngIf"></code>

Studying the [source code for NgIf](#) and [NgForOf](#) is a great way to learn more.

Template input variable

A *template input variable* is a variable whose value you can reference *within* a single instance of the template. There are several such variables in this example: `hero`, `i`, `odd` and `let`. All are preceded by the keyword `let`.

A *template input variable* is *not* the same as a [template reference variable](#), neither *semantically* nor *syntactically*.

You declare a template *input* variable using the keyword `()`. The variable's scope is limited to a *single instance* of the repeated template. You can use the same variable name again in the definition of other structural directives. `let let hero`

You declare a template *reference* variable by prefixing the variable name with `()`. A *reference* variable refers to its attached element, component or directive. It can be accessed *anywhere* in the *entire template*. `##var`

Template *input* and *reference* variable names have their own namespaces. The `in` is never the same variable as the declared as `hero let hero hero#hero`

One structural directive per host element

Someday you'll want to repeat a block of HTML but only when a particular condition is true. You'll *try* to put both an `ngIf` and an `ngFor` on the same host element. Angular won't let you. You may apply only one *structural* directive to an element. `*ngFor *ngIf`

The reason is simplicity. Structural directives can do complex things with the host element and its descendents. When two directives lay claim to the same host element, which one takes precedence? Which should go first, the `ngIf` or the `ngFor`? Can the `ngIf` cancel the effect of the `ngFor`? If so (and it seems like it should be so), how should Angular generalize the ability to cancel for other structural directives?

`NgIf NgFor NgIf NgFor`

There are no easy answers to these questions. Prohibiting multiple structural directives makes them moot. There's an easy solution for this use case: put the `ngIf` on a container element that wraps the `ngFor` element. One or both elements can be an `ng-container` so you don't have to introduce extra levels of HTML. `*ngIf *ngFor`

Inside *NgSwitch* directives

The Angular *NgSwitch* is actually a set of cooperating directives: `ngSwitch`, `ngSwitchCase`, and `ngSwitchDefault`

`NgSwitch NgSwitchCase NgSwitchDefault`

Here's an example.

src/app/app.component.html (ngswitch)

```
<div [ngSwitch]="hero?.emotion">
  <app-happy-hero    *ngSwitchCase="'happy'"    [hero]="hero"></app-happy-hero>
  <app-sad-hero      *ngSwitchCase="'sad'"      [hero]="hero"></app-sad-hero>
  <app-confused-hero *ngSwitchCase="'confused'" [hero]="hero"></app-confused-
hero>
  <app-unknown-hero  *ngSwitchDefault           [hero]="hero"></app-unknown-
hero>
</div>
```

The switch value assigned to () determines which (if any) of the switch cases are displayed. `NgSwitch` `hero.emotion`

`NgSwitch` itself is not a structural directive. It's an *attribute* directive that controls the behavior of the other two switch directives. That's why you write , never . `[ngSwitch]` `*ngSwitch`

`NgSwitchCase` and *are* structural directives. You attach them to elements using the asterisk (*) prefix notation. An displays its host element when its value matches the switch value. The displays its host element when no sibling matches the switch

value. `NgSwitchDefault` `NgSwitchCase` `NgSwitchDefault` `NgSwitchCase`

The element to which you apply a directive is its *host* element. The is the host element for the happy . The is the host element for the . `<happy-hero>` `*ngSwitchCase` `<unknown-hero>` `*ngSwitchDefault`

As with other structural directives, the and can be desugared into the element form. `NgSwitchCase` `NgSwitchDefault` `<ng-template>`

src/app/app.component.html (ngswitch-template)

```
<div [ngSwitch]="hero?.emotion">
  <ng-template [ngSwitchCase]="'happy'">
    <app-happy-hero [hero]="hero"></app-happy-hero>
  </ng-template>
  <ng-template [ngSwitchCase]="'sad'">
    <app-sad-hero [hero]="hero"></app-sad-hero>
  </ng-template>
  <ng-template [ngSwitchCase]="'confused'">
    <app-confused-hero [hero]="hero"></app-confused-hero>
  </ng-template >
  <ng-template ngSwitchDefault>
    <app-unknown-hero [hero]="hero"></app-unknown-hero>
  </ng-template>
</div>
```

Prefer the asterisk (*) syntax.

The asterisk (*) syntax is more clear than the desugared form. Use `<ng-container>` when there's no single element to host the directive.

While there's rarely a good reason to apply a structural directive in template *attribute* or *element* form, it's still important to know that Angular creates a and to understand how it works. You'll refer to the when you [write your own structural directive](#). `<ng-template>` `<ng-template>`

The `<ng-template>`

The `<ng-template>` is an Angular element for rendering HTML. It is never displayed directly. In fact, before rendering the view, Angular *replaces* the and its contents with a comment. `<ng-template>`

If there is no structural directive and you merely wrap some elements in a , those elements disappear. That's the fate of the middle "Hip!" in the phrase "Hip! Hip! Hooray!". `<ng-template>`

src/app/app.component.html (template-tag)

```
<p>Hip!</p>
<ng-template>
  <p>Hip!</p>
</ng-template>
<p>Hooray!</p>
```

Angular erases the middle "Hip!", leaving the cheer a bit less enthusiastic.

<pre><p _ngcontent-c0>Hip!</p> <!-- --> <p _ngcontent-c0>Hooray!</p></pre>	<div>Hip! Hooray!</div>
--	-----------------------------

A structural directive puts a to work as you'll see when you [write your own structural directive](#). `<ng-template>`

Group sibling elements with `<ng-container>`

There's often a *root* element that can and should host the structural directive. The list element (``) is a typical host element of an repeater. `NgFor`

src/app/app.component.html (ngfor-li)

```
<li *ngFor="let hero of heroes">{{hero.name}}</li>
```

When there isn't a host element, you can usually wrap the content in a native HTML container element, such as a `<div>`, and attach the directive to that wrapper. `<div>`

src/app/app.component.html (ngif)

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

Introducing another container element—typically a `<div>` or ``—to group the elements under a single *root* is usually harmless. *Usually* ... but not *always*. `` `<div>`

The grouping element may break the template appearance because CSS styles neither expect nor accommodate the new layout. For example, suppose you have the following paragraph layout.

src/app/app.component.html (ngif-span)

```
<p>
  I turned the corner
  <span *ngIf="hero">
    and saw {{hero.name}}. I waved
  </span>
  and continued on my way.
</p>
```

You also have a CSS style rule that happens to apply to a within a aragraph. `` `<p>`

src/app/app.component.css (p-span)

```
p span { color: red; font-size: 70%; }
```

The constructed paragraph renders strangely.

I turned the corner and saw Mr. Nice. I waved and continued on my way.

The style, intended for use elsewhere, was inadvertently applied here. `p span`

Another problem: some HTML elements require all immediate children to be of a specific type. For example, the element requires children. You can't wrap the *options* in a conditional or a `<select>`

`<option>` `<div>` ``

When you try this,

src/app/app.component.html (select-span)

```
<div>
  Pick your favorite hero
  (<label><input type="checkbox" checked (change)="showSad = !showSad">show
sad</label>)
</div>
<select [(ngModel)]="hero">
  <span *ngFor="let h of heroes">
    <span *ngIf="showSad || h.emotion !== 'sad'">
      <option [ngValue]="h">{{h.name}} ({{h.emotion}})</option>
    </span>
  </span>
</select>
```

the drop down is empty.

Pick your favorite hero, who is ☒ not sad
▼

The browser won't display an within a `<option>`

<ng-container> to the rescue

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

Here's the conditional paragraph again, this time using `<ng-container>`

src/app/app.component.html (ngif-ngcontainer)

```
<p>
  I turned the corner
  <ng-container *ngIf="hero">
    and saw {{hero.name}}. I waved
  </ng-container>
  and continued on my way.
</p>
```

It renders properly.

I turned the corner and saw Mr. Nice. I waved and continued on my way.

Now conditionally exclude a *select* with `<option>` `<ng-container>`

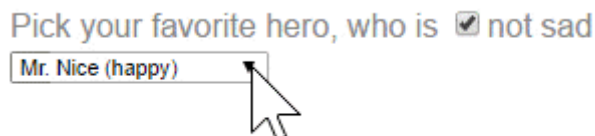
src/app/app.component.html (select-ngcontainer)

```
<div>
  Pick your favorite hero
  (<label><input type="checkbox" checked (change)="showSad = !showSad">show
sad</label>)
</div>
<select [(ngModel)]="hero">
  <ng-container *ngFor="let h of heroes">
    <ng-container *ngIf="showSad || h.emotion !== 'sad'">
      <option [ngValue]="h">{{h.name}} ({{h.emotion}})</option>
    </ng-container>
  </ng-container>
</select>
```

The drop down works properly.

Pick your favorite hero, who is ☒ not sad

Mr. Nice (happy)



Nota: Recuerde que la directiva `ngModel` se define como parte de `Angular FormsModule` y debe incluir `FormsModule` en la sección `imports: [...]` de los metadatos del módulo Angular, en la que desea usarla.

Es un elemento de sintaxis reconocido por el analizador angular. No es una directiva, componente, clase o interfaz. Es más como las llaves en un bloque JavaScript: `<ng-container>` `if`


```
if (someCondition) {  
  statement1;  
  statement2;  
  statement3;  
}
```

Sin esas llaves, JavaScript solo ejecutaría la primera instrucción cuando tenga la intención de ejecutar condicionalmente todas ellas como un solo bloque. El `<ng-container>` satisface una necesidad similar en las plantillas Angular.

Escribir una directiva estructural

En esta sección, se escribe una directiva estructural que hace lo contrario de `ngIf`. muestra el contenido de la plantilla cuando la condición es `true`. muestra el contenido cuando la condición es

`false`. `UnlessDirective` `NgIf` `NgIf true` `UnlessDirective`

src/app/app.component.html (appUnless-1)

```
<p *appUnless="condition">Show this sentence unless the condition is true.</p>
```

La creación de una directiva es similar a la creación de un componente.

- Importe el decorador (en lugar del decorador). `Directive` `Component`
- Importar los símbolos `Input`, `TemplateRef`, y `ViewContainerRef`; los necesitarás para *cualquier* directiva estructural.
- Aplique el decorador a la clase de directiva.
- Establezca el *selector de atributos* CSS que identifica la directiva cuando se aplica a un elemento de una plantilla.

Así es como puedes comenzar:

src/app/unless.directive.ts (esqueleto)

```
import { Directive, Input, TemplateRef, ViewContainerRef } from  
'@angular/core';  
  
@Directive({ selector: '[appUnless]' })  
export class UnlessDirective {  
}
```

El *selector* de la directiva suele ser el nombre de **atributo** de la directiva entre corchetes, . Los corchetes definen un [selector de atributos](#) [CSS](#). `[appUnless]`

El *nombre del atributo* de directiva debe escribirse en *lowerCamelCase* y comenzar con un prefijo. No use . Ese prefijo pertenece a Angular. Elige algo corto que se adapte a ti o a tu empresa. En este ejemplo, el prefijo es `.ngapp`

El nombre de *la clase* de directiva termina según la guía [de estilos](#). Las propias directivas de Angular no lo hacen. `Directive`

TemplateRef y ViewContainerRef

Una directiva estructural simple como esta crea una *vista incrustada* a partir de la angular generada e inserta esa vista en un *contenedor de vista* adyacente al elemento host original de la directiva. `<ng-template>` `<p>`

Adquirirá el contenido con `TemplateRef` y tendrá acceso al *contenedor de vista* a través de `ViewContainerRef`. `<ng-template>`

Se inyectan ambos en el constructor de directivas como variables privadas de la clase.

src/app/unless.directive.ts (ctor)

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

La propiedad *appUnless*

La directiva consumer espera enlazar una condición true/false a . Eso significa que la directiva necesita una propiedad, decorada con `[appUnless]` `appUnless@Input`

Lea en la guía [Sintaxis de plantilla](#). `@Input`

src/app/unless.directive.ts (set)

```
@Input() set appUnless(condition: boolean) {  
  if (!condition && !this.hasView) {  
    this.viewContainer.createEmbeddedView(this.templateRef);  
    this.hasView = true;  
  } else if (condition && this.hasView) {  
    this.viewContainer.clear();  
    this.hasView = false;  
  }  
}
```

Angular establece la propiedad cada vez que cambia el valor de la condición. Debido a que la propiedad funciona, necesita un establecedor. `appUnless` `appUnless`

- Si la condición es falsa y la vista no se ha creado anteriormente, indique al *contenedor de vista* que cree la vista *incrustada* desde la plantilla.
- Si la condición es veraz y la vista se muestra actualmente, borre el contenedor que también destruye la vista.

Nadie lee la propiedad, así que no necesita un captador. `appUnless`

El código de directiva completado tiene este aspecto:

src/app/unless.directive.ts (extracto)

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

/**
 * Add the template content to the DOM unless the condition is true.
 */
@Directive({ selector: '[appUnless]' })
export class UnlessDirective {
  private hasView = false;

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

  @Input() set appUnless(condition: boolean) {
    if (!condition && !this.hasView) {
      this.viewContainer.createEmbeddedView(this.templateRef);
      this.hasView = true;
    } else if (condition && this.hasView) {
      this.viewContainer.clear();
      this.hasView = false;
    }
  }
}
```

Add this directive to the array of the AppModule. [declarations](#)

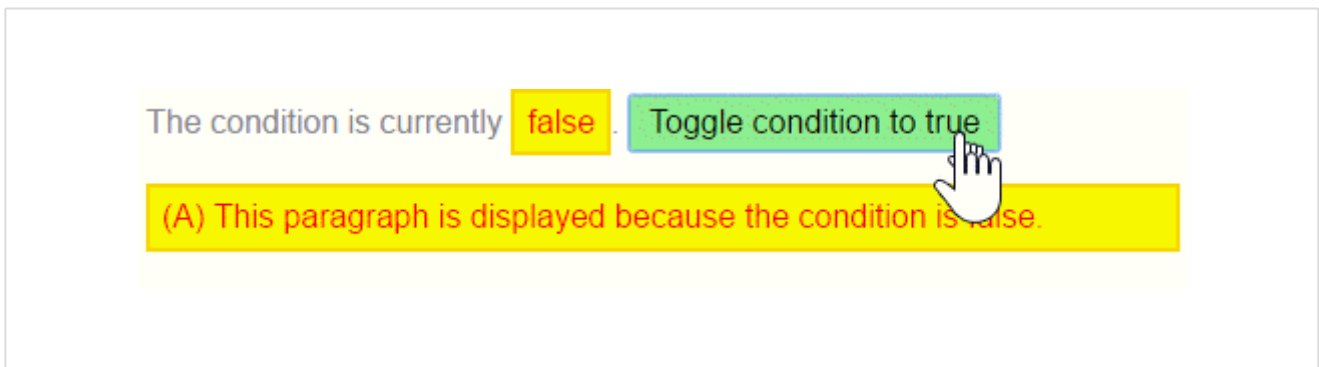
Then create some HTML to try it.

src/app/app.component.html (appUnless)

```
<p *appUnless="condition" class="unless a">
  (A) This paragraph is displayed because the condition is false.
</p>

<p *appUnless="!condition" class="unless b">
  (B) Although the condition is true,
  this paragraph is displayed because appUnless is set to false.
</p>
```

When the is falsy, the top (A) paragraph appears and the bottom (B) paragraph disappears. When the is truthy, the top (A) paragraph is removed and the bottom (B) paragraph appears. `condition` `condition`



Improving template type checking for custom directives

You can improve template type checking for custom directives by adding template guard properties to your directive definition. These properties help the Angular template type checker find mistakes in the template at compile time, which can avoid runtime errors those mistakes can cause.

Use the type-guard properties to inform the template type checker of an expected type, thus improving compile-time type-checking for that template.

- A property lets you specify a more accurate type for an input expression within the template. `ngTemplateGuard_(someInputProperty)`
- The static property declares the type of the template context. `ngTemplateContextGuard`

This section provides example of both kinds of type-guard property.

For more information, see [Template type checking guide](#).

Make in-template type requirements more specific with template guards

A structural directive in a template controls whether that template is rendered at run time, based on its input expression. To help the compiler catch template type errors, you should specify as closely as possible the required type of a directive's input expression when it occurs inside the template.

A type guard function *narrows* the expected type of an input expression to a subset of types that might be passed to the directive within the template at run time. You can provide such a function to help the type-checker infer the proper type for the expression at compile time.

For example, the implementation uses type-narrowing to ensure that the template is only instantiated if the input expression is truthy. To provide the specific type requirement, the directive defines a

static property `ngTemplateGuard_ngIf: 'binding'`. The value is a special case for a common kind of type-narrowing where the input expression is evaluated in order to satisfy the type requirement. `NgIf*ngIfNgIfbinding`

To provide a more specific type for an input expression to a directive within the template, add a property to the directive, where the suffix to the static property name is the field name. The value of the property can be either a general type-narrowing function based on its return type, or the string as in the case of `ngTemplateGuard_xx@Input"binding"NgIf`

For example, consider the following structural directive that takes the result of a template expression as an input.

IfLoadedDirective

```
export type Loaded = { type: 'loaded', data: T };
export type Loading = { type: 'loading' };
export type LoadingState = Loaded | Loading;
export class IfLoadedDirective {
  @Input('ifLoaded') set state(state: LoadingState) {}
  static ngTemplateGuard_state(dir: IfLoadedDirective, expr: LoadingState):
    expr is Loaded { return true; };
  export interface Person {
    name: string;
  }

  @Component({
    template: `
    {{ state.data }}
    `,
  })
  export class AppComponent {
    state: LoadingState;
  }
}
```

In this example, the type permits either of two states, or . The expression used as the directive's input is of the umbrella type , as it's unknown what the loading state is at that point. `LoadingState<T>Loaded<T>LoadingstateLoadingState`

The definition declares the static field , which expresses the narrowing behavior. Within the template, the structural directive should render this template only when is actually . The type guard allows the type checker to infer that the acceptable type of within the template is a , and further infer that must be an instance of

`IfLoadedDirectivengTemplateGuard_stateAppComponent*ifLoadedstateLoaded<Person>stateLo`

Typing the directive's context

If your structural directive provides a context to the instantiated template, you can properly type it inside the template by providing a static function. The following snippet shows an example of such a function. [ngTemplateContextGuard](#)

myDirective.ts

```
@Directive({...})
export class ExampleDirective {
    // Make sure the template checker knows the type of the context with which
    the
    // template of this directive will be rendered
    static ngTemplateContextGuard(dir: ExampleDirective, ctx: unknown): ctx is
    ExampleContext { return true; };

    // ...
}
```

Summary

You can both try and download the source code for this guide in the [live example](#) / [download example](#).

Here is the source from the folder. [src/app/](#)

app.component.ts

app.component.html

app.component.css

app.module.ts

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

import { Hero, heroes } from './hero';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: [ './app.component.css' ]
})
export class AppComponent {
    heroes = heroes;
    hero = this.heroes[0];

    condition = false;
```

```
logs: string[] = [];  
showSad = true;  
status = 'ready';  
  
trackById(index: number, hero: Hero): number { return hero.id; }  
}
```

You learned:

- that structural directives manipulate HTML layout.
- to use `<ng-container>` as a grouping element when there is no suitable host element.
- that the Angular desugars **asterisk (*) syntax** into a `<ng-template>`
- how that works for the `,` and built-in directives `NgIf` `NgFor` `NgSwitch`
- about the *microsyntax* that expands into a `<ng-template>`.
- to write a **custom structural directive**, `UnlessDirective`