Structural directives

Contents ›

This guide looks at how Angular manipulates the DOM with **structural directives** and how you can write your own structural directives to do the same thing.

Try the live example / download example.

What are structural directives?

Structural directives are responsible for HTML layout. They shape or reshape the DOM's *structure*, typically by adding, removing, or manipulating elements.

As with other directives, you apply a structural directive to a *host element*. The directive then does whatever it's supposed to do with that host element and its descendants.

Structural directives are easy to recognize. An asterisk (*) precedes the directive attribute name as in this example.

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

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

No brackets. No parentheses. Just `*ngIf` set to a string.

You'll learn in this guide that the asterisk (*) is a convenience notation and the string is a *microsyntax* rather than the usual template expression. Angular desugars this notation into a marked-up `<ng-template>` that surrounds the host element and its descendants. Each structural directive does something different with that template.

Three of the common, built-in structural directives—NgIf, NgFor, and NgSwitch...—are described in the Built-in directives guide and seen in samples throughout the Angular documentation. Here's an example of them in a template:

src/app/app.component.html (built-in)

```
<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>
```

This guide won't repeat how to *use* them. But it does explain *how they work* and how to write your own structural directive.

DIRECTIVE SPELLING

Throughout this guide, you'll see a directive spelled in both *UpperCamelCase* and *lowerCamelCase*. Already you've seen `NgIf` and `ngIf`. There's a reason. `NgIf` refers to the directive *class*; `ngIf` refers to the directive's *attribute name*.

A directive *class* is spelled in *UpperCamelCase* (`NgIf`). A directive's *attribute name* is spelled in *lowerCamelCase* (`ngIf`). The guide refers to the directive *class* when talking about its properties and what the directive does. The guide refers to the *attribute name* when describing how you apply the directive to an element in the HTML template.

There are two other kinds of Angular directives, described extensively elsewhere: (1) components and (2) attribute directives.

A *component* manages a region of HTML in the manner of a native HTML element. Technically it's a directive with a template.

An *attribute* directive changes the appearance or behavior of an element, component, or another directive. For example, the built-in `NgStyle` directive changes several element styles at the same time.

You can apply many *attribute* directives to one host element. You can only apply one *structural* directive to a host element.

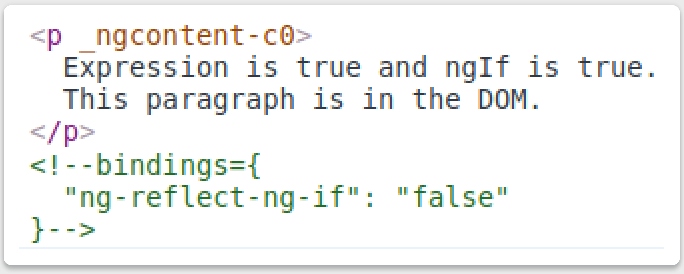
NgIf case study

`NgIf` is the simplest structural directive and the easiest to understand. It takes a boolean expression and makes an entire chunk of the DOM appear or disappear.

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>
```

The `ngIf` directive doesn't hide elements with CSS. It adds and removes them physically from the DOM. Confirm that fact using browser developer tools to inspect the DOM.



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

The top paragraph is in the DOM. The bottom, disused paragraph is not; in its place is a comment about "bindings" (more about that later).

When the condition is false, `NgIf` removes its host element from the DOM, detaches it from DOM events (the attachments that it made), detaches the component from Angular change detection, and destroys it. The component and DOM nodes can be garbage-collected and free up memory.

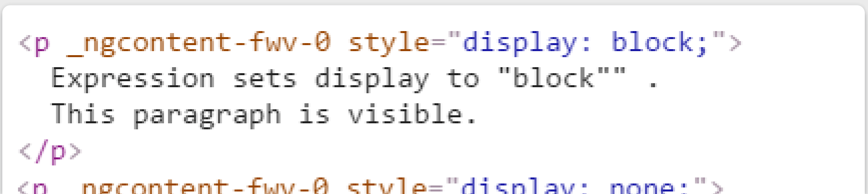
Why *remove* rather than *hide*?

A directive could hide the unwanted paragraph instead by setting its `display` style to `none`.

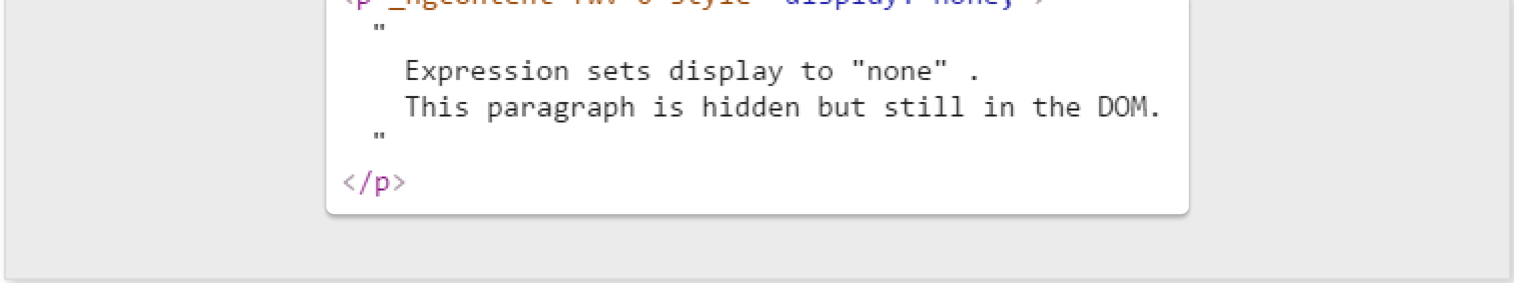
src/app/app.component.html (display-none)

```
<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>
```

While invisible, the element remains in the 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>
```

The difference between hiding and removing doesn't matter for a simple paragraph. It does matter when the host element is attached to a resource intensive component. Such a component's behavior continues even when hidden. The component stays attached to its DOM element. It keeps listening to events. Angular keeps checking for changes that could affect data bindings. Whatever the component was doing, it keeps doing.

Although invisible, the component—and all of its descendant components—tie up resources. The performance and memory burden can be substantial, responsiveness can degrade, and the user sees nothing.

On the positive side, showing the element again is quick. The component's previous state is preserved and ready to display. The component doesn't re-initialize—an operation that could be expensive. So hiding and showing is sometimes the right thing to do.

But in the absence of a compelling reason to keep them around, your preference should be to remove DOM elements that the user can't see and recover the unused resources with a structural directive like `NgIf` .

These same considerations apply to every structural directive, whether built-in or custom. Before applying a structural directive, you might want to pause for a moment to consider the consequences of adding and removing elements and of creating and destroying components.

The asterisk (*) prefix

Surely you noticed the asterisk (*) prefix to the directive name and wondered why it is necessary and what it does.

Here is `*ngIf` displaying the hero's name if `hero` exists.

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

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

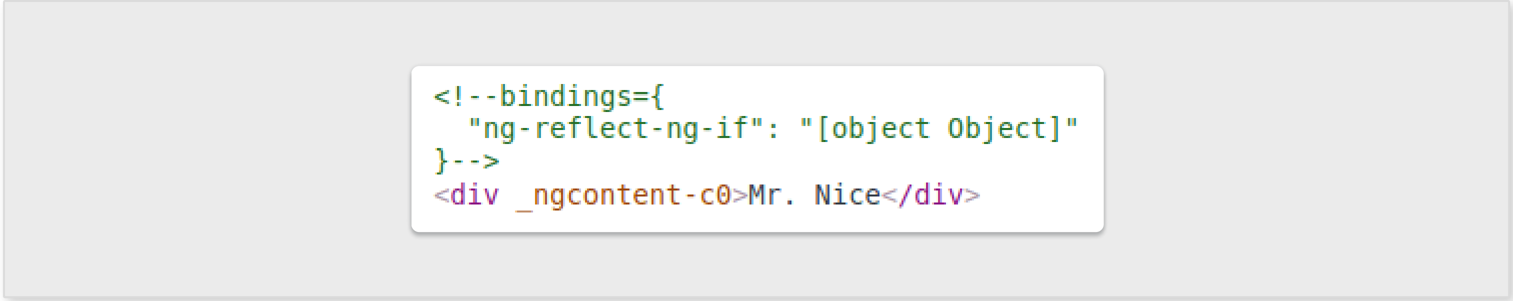
The asterisk is "syntactic sugar" for something a bit more complicated. Internally, Angular translates the `*ngIf` *attribute* into a `<ng-template>` *element*, wrapped around the host element, like this.

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

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

- The `*ngIf` directive moved to the `<ng-template>` element where it became a property binding,`[ngIf]`.
- The rest of the `<div>`, including its class attribute, moved inside the `<ng-template>` element.

The first form is not actually rendered, only the finished product ends up in the DOM.



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

Angular consumed the `<ng-template>` content during its actual rendering and replaced the `<ng-template>` with a diagnostic comment.

The `NgFor` and `NgSwitch...` directives follow the same pattern.

Inside **ngFor*

Angular transforms the `*ngFor` in similar fashion from asterisk (*) syntax to `<ng-template>` *element*.

Here's a full-featured application of `NgFor`, written both ways:

src/app/app.component.html (inside-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>
```

This is manifestly more complicated than `ngIf` and rightly so. The `NgFor` directive has more features, both required and optional, than the `NgIf` shown in this guide. At minimum `NgFor` needs a looping variable (`let hero`) and a list (`heroes`).

You enable these features in the string assigned to `ngFor`, which you write in Angular's microsyntax.

> Everything *outside* the `ngFor` string stays with the host element (the `<div>`) as it moves inside the `<ng-template>`. In this example, the `[class.odd]="odd"` stays on the `<div>`.

Microsyntax

The Angular microsyntax lets you configure a directive in a compact, friendly string. The microsyntax parser translates that string into attributes on the `<ng-template>`:

- The `let` keyword declares a *template input variable* that you reference within the template. The input variables in this example are `hero`, `i`, and `odd`. The parser translates `let hero`, `let i`, and `let odd` into variables named `let-hero`, `let-i`, and `let-odd`.

- The microsyntax parser title-cases all directives and prefixes them with the directive's attribute name, such as `ngFor`. For example, the `ngFor` input properties, `of` and `trackBy`, become `ngForOf` and `ngForTrackBy`, respectively. That's how the directive learns that the list is `heroes` and the track-by function is `trackById`.
- As the `NgFor` directive loops through the list, it sets and resets properties of its own *context* object. These properties can include, but aren't limited to, `index`, `odd`, and a special property named `$implicit`.
- The `let-i` and `let-odd` variables were defined as `let i=index` and `let odd=odd`. Angular sets them to the current value of the context's `index` and `odd` properties.
- The context property for `let-hero` wasn't specified. Its intended source is implicit. Angular sets `let-hero` to the value of the context's `$implicit` property, which `NgFor` has initialized with the hero for the current iteration.
- The `NgFor` API guide describes additional `NgFor` directive properties and context properties.
- The `NgForOf` directive implements `NgFor`. Read more about additional `NgForOf` directive properties and context properties in the NgForOf API reference.

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 `<div *ngFor="let item of items">{{item}}</div>` instead of `<ng-template ngFor let-item [ngForOf]="items"><div>{{item}}</div></ng-template>`. The following sections provide detailed information on constraints, grammar, and translation of microsyntax.

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.

`prefix`	HTML attribute key
`key`	HTML attribute key
`local`	local variable name used in the template

`export`	value exported by the directive under a given name
`expression`	standard Angular expression

`keyExp = :key ":"? :expression ("as" :local)? ";"?`
`let = "let" :local "=" :export ";"?`
`as = :export "as" :local ";"?`

Translation

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

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

Microsyntax examples

The following table demonstrates how Angular desugars microsyntax.

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

```
*ngIf="exp as value"                <ng-template [ngIf]="exp" let-value="ngIf">
```

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`, and `odd`. 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 `let` keyword (`let hero`). 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.

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

Template *input* and *reference* variable names have their own namespaces. The `hero` in `let hero` is never the same variable as the `hero` declared as `#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 `*ngFor` and an `*ngIf` on the same host element. Angular won't let you. You may apply only one *structural* directive to an element.

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?

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.

Inside *NgSwitch* directives

The Angular *NgSwitch* is actually a set of cooperating directives: `NgSwitch`, `NgSwitchCase`, and `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 `NgSwitch` (`hero.emotion`) determines which (if any) of the switch cases are displayed.

`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 `[ngSwitch]`, never `*ngSwitch`.

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

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

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

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 `<ng-template>` and to understand how it works. You'll refer to the `<ng-template>` when you write your own structural directive.

The *<ng-template>*

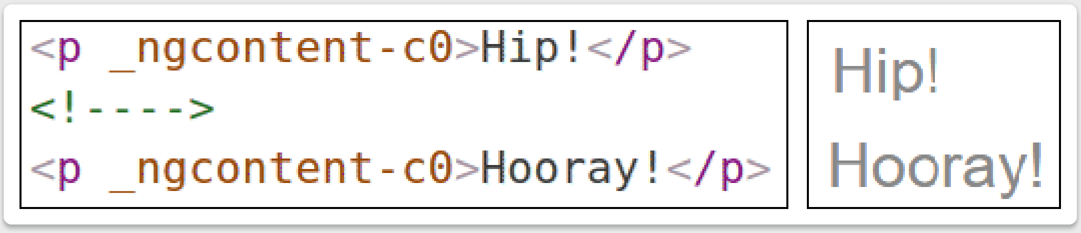
The <ng-template> is an Angular element for rendering HTML. It is never displayed directly. In fact, before rendering the view, Angular *replaces* the `<ng-template>` and its contents with a comment.

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

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.



```
<p _ngcontent-c0>Hip!</p>
<!---->
<p _ngcontent-c0>Hooray!</p>
```

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

Group sibling elements with <ng-container>

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

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.

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

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

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

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)

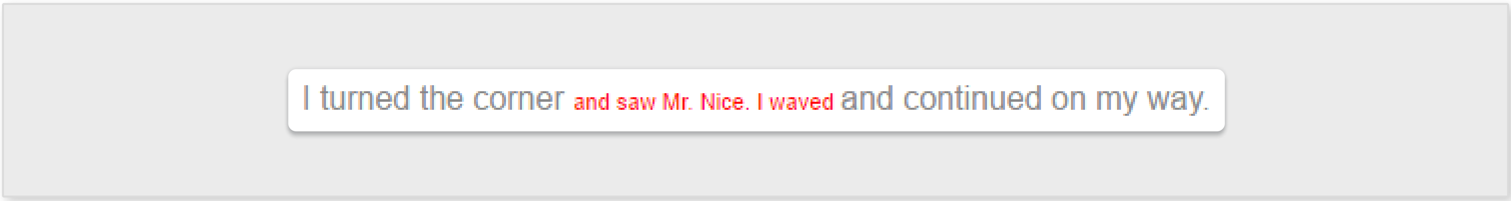
```
<p>
  I turned the corner
  <span *ngIf="hero">
    and saw {{hero.name}}. I waved
  </span>
  and continued on my way.
</p>
```

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

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

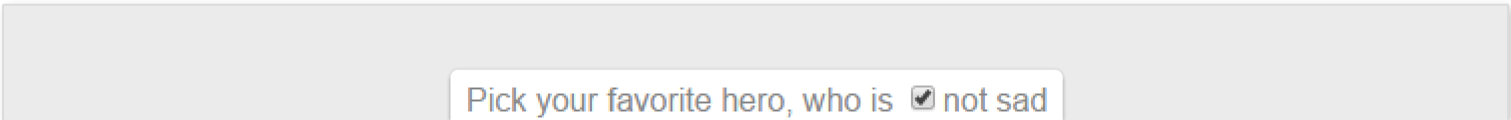
Another problem: some HTML elements require all immediate children to be of a specific type. For example, the `<select>` element requires `<option>` children. You can't wrap the *options* in a conditional `<div>` or a `<span>`.

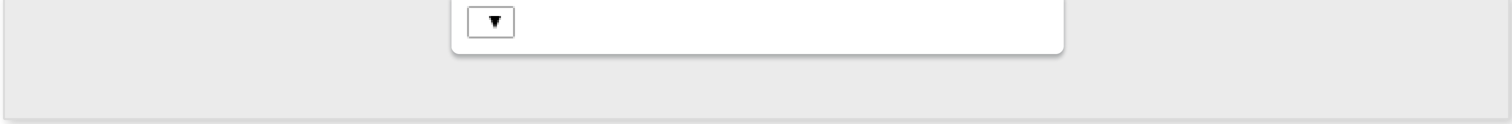
When you try this,

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 `<option>` within a `<span>`.

<ng-container> to the rescue

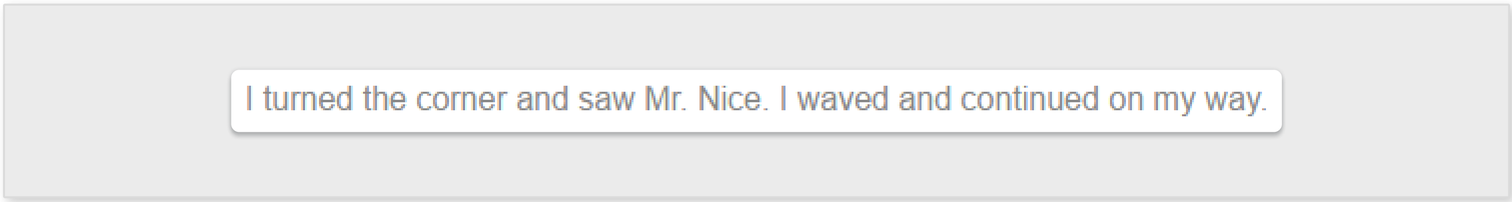
The Angular `<ng-container>` is a grouping element that doesn't interfere with styles or layout because Angular *doesn't put it in the DOM*.

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.



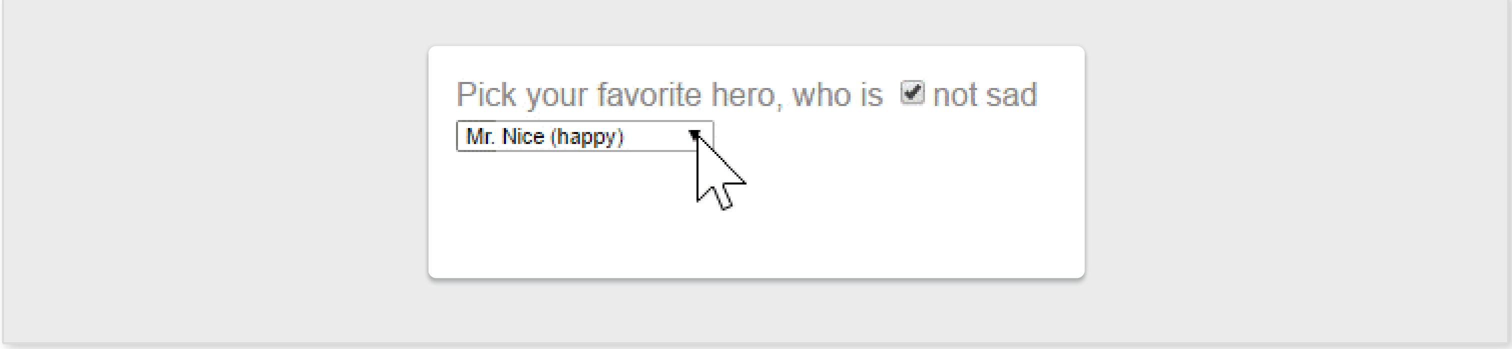
I turned the corner and saw Mr. Nice. I waved and continued on my way.

Now conditionally exclude a *select* `<option>` with `<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.



> **Note:** Remember that ngModel directive is defined as a part of Angular FormsModule and you need to include FormsModule in the imports: [...] section of the Angular module metadata, in which you want to use it.

The `<ng-container>` is a syntax element recognized by the Angular parser. It's not a directive, component, class, or interface. It's more like the curly braces in a JavaScript `if`-block:

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

Without those braces, JavaScript would only execute the first statement when you intend to conditionally execute all of them as a single block. The `<ng-container>` satisfies a similar need in Angular templates.

Write a structural directive

In this section, you write an `UnlessDirective` structural directive that does the opposite of `NgIf`. `NgIf` displays the template content when the condition is `true`. `UnlessDirective` displays the content when the condition is ***false***.

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

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

Creating a directive is similar to creating a component.

- Import the `Directive` decorator (instead of the `Component` decorator).
- Import the `Input`, `TemplateRef`, and `ViewContainerRef` symbols; you'll need them for *any* structural directive.
- Apply the decorator to the directive class.
- Set the CSS *attribute selector* that identifies the directive when applied to an element in a template.

Here's how you might begin:

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

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

@Directive({ selector: '[appUnless]'})
export class UnlessDirective {
}
```

The directive's *selector* is typically the directive's **attribute name** in square brackets, `[appUnless]`. The brackets define a CSS attribute selector.

The directive *attribute name* should be spelled in *lowerCamelCase* and begin with a prefix. Don't use `ng`. That prefix belongs to Angular. Pick something short that fits you or your company. In this example, the prefix is `app`.

The directive *class* name ends in `Directive` per the style guide. Angular's own directives do not.

TemplateRef and *ViewContainerRef*

A simple structural directive like this one creates an *embedded view* from the Angular-generated `<ng-template>` and inserts that view in a *view container* adjacent to the directive's original `<p>` host element.

You'll acquire the `<ng-template>` contents with a `TemplateRef` and access the *view container* through a `ViewContainerRef`.

You inject both in the directive constructor as private variables of the class.

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

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

The *appUnless* property

The directive consumer expects to bind a true/false condition to `[appUnless]`. That means the directive needs an `appUnless` property, decorated with `@Input`

Read about `@Input` in the `@Input()` and `@Output()` properties guide.

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 sets the `appUnless` property whenever the value of the condition changes. Because the `appUnless` property does work, it needs a setter.

- If the condition is falsy and the view hasn't been created previously, tell the *view container* to create the *embedded view* from the template.
- If the condition is truthy and the view is currently displayed, clear the container which also destroys the view.

Nobody reads the `appUnless` property so it doesn't need a getter.

The completed directive code looks like this:

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

```
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 `declarations` array of the AppModule.

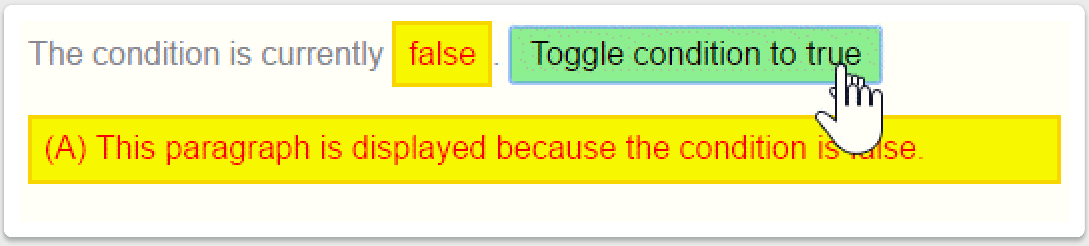
Then create some HTML to try it.

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



(A) This paragraph is displayed because the condition is false.

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

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 NgIf implementation uses type-narrowing to ensure that the template is only instantiated if the input expression to *ngIf is truthy. To provide the specific type requirement, the NgIf directive defines a static property ngTemplateGuard_ngIf: 'binding'. The binding 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.

To provide a more specific type for an input expression to a directive within the template, add a ngTemplateGuard_xx property to the directive, where the suffix to the static property name is the @Input field name. The value of the property can be either a general type-narrowing function based on its return type, or the string "binding" as in the case of 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: `<div *ifLoaded="state">{{ state.data }}</div>`,
})
export class AppComponent {
  state: LoadingState;
}
```

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

The IfLoadedDirective definition declares the static field ngTemplateGuard_state, which expresses the narrowing behavior. Within the AppComponent template, the *ifLoaded structural directive should render this template only when state is actually Loaded<Person>. The type guard allows the type checker to infer that the acceptable type of state within the template is a Loaded<T>, and further infer that T must be an instance of Person.

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 ngTemplateContextGuard function. The following snippet shows an example of such a function.

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 `src/app/` folder.

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 `NgIf`, `NgFor` and `NgSwitch` built-in directives.
- about the *microsyntax* that expands into a `<ng-template>`.
- to write a custom structural directive, `UnlessDirective`.