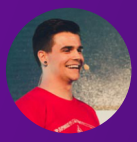
Ultimate Courses™

Angular Directives In-Depth



by Todd Motto

Google Developer Expert

Reactive patterns with Observables & Async Pipe

Master the built-in Directives:
NgIf, NgFor, NgSwitch, NgStyle and NgClass

Contents

Introduction **2**
What Directives will we cover? 2
Structural and Attribute Directives 3
Directive API 3

NgIf **4**
NgIf 4
NgIf, Else 4
NgIf, Else, Then 5
NgIf and <ng-container> 6
NgIf and Async Pipe 6
NgIf and <ng-template> Syntax 8

NgFor **9**
NgFor 9
NgFor and TrackBy Expression 9
NgFor Variables - Index, Odd, Even 11
NgFor and Async Pipe 12
NgFor and <ng-template> Syntax 13

NgSwitch **14**
NgSwitch 14
NgSwitch and Default Case 15
NgSwitch and <ng-container> 15
NgSwitch and <ng-template> Syntax 16

NgStyle and Style Bindings **18**
Style Bindings 18
Multiple Style Bindings 19
Style Bindings and Units 19
Style Bindings via Objects 19
Style Bindings from Class Methods 20
Style Bindings from Class Getters 21
Style Bindings from Strings 21
NgStyle 22

NgClass and Class Bindings **23**
Class Bindings 23
Multiple Class Bindings 24
Class Bindings via Objects 24
Class Bindings from Class Methods 24
Class Bindings from Class Getters 25
Class Bindings from Strings 26
Class Bindings from Arrays 26
NgClass 27

Hi there! I'm Todd Motto[1], a Google Developer Expert for Angular and founder of Ultimate Courses.

This is my straightforward, comprehensive guide to mastering Angular's Directives!

Whether you're just getting started or have been using Angular for years, I'm sure this book will give you valuable insights and practices to fully master the Directives that Angular has to offer.

I'd love to hear how you've enjoyed this book as well, please let me know on Twitter via @toddmotto[2].

Excited yet? Let's get started!

[1] https://ultimatecourses.com/about
[2] https://twitter.com/toddmotto

Introduction

If you're new to Angular, the word "Directive" might also be a fairly new addition to your vocabulary.

Directives are a core concept in Angular that allow us to add functionality, and behaviour, to either an element or component. They're an integral part of what makes Angular a great choice to build web applications as they allow us to encapsulate and reuse logic easily.

Angular provides various Directives for us out of the box, and this book is a deep-dive on the most common ones you'll use on a daily basis. Prepare to understand how each Angular Directive works, why it was created and what problems it solves.

Not only will you understand the Directives, but how they fit into the bigger picture when dealing with things such as State Management, Component Architecture, Observables and much more.

What Directives will we cover?

In order, we're going to fully explore the use cases, best practices and advanced syntax for the following Directives:

- NgIf
- NgFor
- NgSwitch
- NgStyle
- NgClass

> For all other built-in Angular Directives, feel free to check out the documentation[3] at any time.

All Directives that we're going to cover here are exported from Angular's CommonModule[4], which is a required import to use them. By default the `BrowserModule` will include this `CommonModule` for us.

If you're new to Angular, great! I'm going to assume that you're comfortable with JavaScript and the DOM (Document Object Model), so we can dive straight in and get to the good stuff. There's lots to cover!

[3] https://angular.io/api?type=directive
[4] https://angular.io/api/common/CommonModule

Structural and Attribute Directives

I'd like to take a moment to set the scene and give you some insight into the two types of Angular Directives; structural and attribute. To use a Directive, we declare its name on an element or component. There are a few variations in how to declare, based on the type of Directive we are using.

Structural Directives

Structural Directives are responsible for managing how templates are rendered inside our components. Structural Directives begin with an asterisk * syntax and tell Angular to treat the element or component as a template.

We can then instruct Angular what to do with that template. We can show or hide it, iterate over it alongside a data structure and much more.

This thinking is similar to how Web Components[5] work with the `<template>` element. In Angular, we have `<ng-template>` that offers similar functionality, and I'll uncover the more advanced use cases and reasons we should adopt this style.

Attribute Directives

Attribute Directives are simpler and change existing elements or components appearance or behaviour. The key difference is that they do not create new templates or render them, they alter existing elements and components. This could be by adding a `class` or setting an inline `style`.

Directive API

Angular exposes it's Directive API[6] to us, which enables us to create our own custom Directives.

If you'd like to learn how to create a custom Directive I'd encourage you to read my technical article on Creating a Custom Directive in Angular[7]. This book is dedicated to covering Angular's built-in Directives only.

[5] https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_templates_and_slots
[6] https://angular.io/api/core/Directive
[7] https://ultimatecourses.com/blog/

NgIf

NgIf[8] is a Structural Directive that allows us to conditionally render an element or component based on an expression.

NgIf

The standard `NgIf` syntax is `*ngIf="expression"` and provides us with the ability to render a template if the expression evaluates to `true`, and optionally supply a fallback if `false`.

We declare `NgIf` and supply a condition that must pass if the element is to be rendered:

```
@Component({
  template: `
    <div *ngIf="donuts.length">
      We have donuts!
    </div>
  `
})
export class AppComponent {
  donuts: Donut[] = [
    { name: 'Glazed Fudge', price: 119 }
  ];
}
```

As `donuts.length` will be `1` this will render the element, because `1` evaluates to `true` in JavaScript (and Angular expressions behave the same way).

Angular is a reactive framework and will also manage these updates if and when new data becomes available, therefore if `donuts.length` changes to `0`, evaluating to `false`, the `NgIf` Directive will respond accordingly and remove the element entirely.

NgIf, Else

`NgIf` also supports an `else` clause, much like `if` and `else` in JavaScript statements. This allows us to render a template should the expression evaluate false.

[8] https://angular.io/api/common/NgIf

To create an `else` block we first declare an `<ng-template>` and a Template Reference Variable[9] to create a local reference to the template.

This reference variable called `#nothing` is then passed into `NgIf` as part of the expression:

```
<div *ngIf="donuts.length; else nothing">
  We have donuts!
</div>

<ng-template #nothing>
  No donuts here.
</ng-template>
```

Angular will never render an `<ng-template>` unless you instruct it to via a Directive or Template Reference Variable.It is there to register a template partial and will remain in virtual memory until needed. If you create an `<ng-template>` and never specify a Directive, you will never see it rendered (much like a Web Component `<template>` element).

NgIf, Else, Then

As component templates grow it may be useful to consider the `then` syntax with `NgIf`. This syntax allows us to declare where the templates will render, but abstract where we create them.

This lets us break things out into two places, with the `then` and `else` clauses both expecting a Template Reference Variable from each `<ng-template>`:

```
<div *ngIf="donuts.length; then showDonuts; else nothing">
  This is always ignored.
</div>

<ng-template #showDonuts>
  We have donuts!
</ng-template>

<ng-template #nothing>
  No donuts here.
</ng-template>
```

This approach allows more separation, however our `<div *ngIf>` is always rendered which creates an additional element every single time.

If only there was a way to declare a Directive without rendering an additional element...

That's where `<ng-container>` comes in.

[9] https://ultimatecourses.com/blog

NgIf and <ng-container>

To avoid rendering an extra element, such as a `<div>`, we can turn to `<ng-container>` to wrap elements and provide a spot to bind Directives:

```
<ng-container *ngIf="donuts.length; then showDonuts; else nothing"></ng-container>

<ng-template #showDonuts>
  We have donuts!
</ng-template>

<ng-template #nothing>
  No donuts here.
</ng-template>
```

My personal preference is to use this approach, which uses an `<ng-container>` alongside just an `else` clause:

```
<ng-container *ngIf="donuts.length; else nothing">
  We have donuts!
</ng-container>

<ng-template #nothing>
  No donuts here.
</ng-template>
```

I prefer this approach because it allows me to see directly what is rendered at the declaration site.

Although the `then` syntax is powerful and clever, I don't feel it justifies the additional overhead to work out what's happening, nor does it make the template any cleaner.

The combination of `<ng-container>` and `<ng-template>` is a great solution to many real-world problems and scenarios.

It also handles Observables very well and allows us to expand a result as a variable.

NgIf and Async Pipe

Angular is a reactive framework, which means it supports and implements Observables.

We can subscribe to Observables directly inside our templates using the `async` pipe. We can also combine the `async` pipe with `NgIf`.

Using the Async Pipe with NgIf[10] at the `<ng-container>` level subscribes our `Observable` and stores the result on a variable that we can reference:

[10] https://ultimatecourses.com/blog/angular-ngif-async-pipe

```
@Component({
  template: `
    <ng-container *ngIf="(donuts$ | async) as donuts">
      {{ donuts }}
    </ng-container>
  `,
})
export class AppComponent {
  donuts$: Observable<Donut[]> = of([
    { name: 'Glazed Fudge', price: 119 }
  ]);
}
```

Notice here how we are using `as donuts` alongside the `async` expression. Once the `donuts$` Observable has resolved, the result is stored on the `donuts` local variable, which can then be passed down to child components or consumed inside the template.

Our `<ng-container>` is only being used to subscribe to our Observable, we have removed the `else` clause. This would now need to live inside the template to check if `donuts.length` was `true` or `false` to render our components.

Another approach we could take is to pass an `async` pipe subscription directly to a component `@Input`:

```
@Component({
  template: `
    <donut-list [donuts]="donuts$ | async"></donut-list>
  `,
})
export class AppComponent {
  donuts$: Observable<Donut[]> = of([
    { name: 'Glazed Fudge', price: 119 }
  ]);
}
```

This means inside the `<donut-list>` component, an `NgIf` check can be made on the `donuts.length` property and can conditionally render the list or `<ng-template>` fallback.

If the same Observable needs to be passed into multiple components you will still need to adopt the initial approach and unwrap the Observable via `<ng-container>`. This is to avoid duplicate subscriptions that would cause memory leaks, state management problems as well as debugging issues.

NgIf and <ng-template> Syntax

The * syntax alongside `NgIf` is there to provide shorthand sugar syntax for a more complex template definition.

To complete our picture, it's imperative for us to learn what Angular translates our `*ngIf` declaration into.

When we declare `*ngIf` two things are happening; a property binding and template declaration.

For more advanced use cases, we can instead use `<ng-template>` directly, which is what Angular translates our code into behind-the-scenes when compiling our components.

This means that the two examples below are essentially equivalent to eachother:

```
<!-- ng-container with sugar syntax -->
<ng-container *ngIf="donuts.length; else nothing">
  We have donuts!
</ng-container>

<!-- ng-template with expanded directive bindings -->
<ng-template [ngIf]="donuts.length" [ngIfElse]="nothing">
  We have donuts!
</ng-template>

<ng-template #nothing>
  No donuts here.
</ng-template>
```

My preference is to use `*ngIf` consistently and only introduce the `<ng-template>` approach when you have a more complex requirement or strictly need `<ng-template>`.

NgFor

NgFor[11] is a Structural Directive that allows us to iterate over a collection of data and access each value, index and more.

NgFor

The standard `NgFor` syntax is `*ngFor="let x of iterable"` and provides us with the ability to iterate over a data structure, such as an Array or Iterable[12].

We declare `NgFor` on an element or component that we wish to repeat for each iteration of our data structure.

```
@Component({
  template: `
    <div *ngFor="let donut of donuts">
      {{ donut.name }}
    </div>
  `
})
export class AppComponent {
  donuts: Donut[] = [
    { id: 'x4f0ae', name: 'Glazed Fudge', price: 119 },
    { id: 'o2k95v', name: 'Sour Supreme', price: 129 }
  ];
}
```

As `donuts` contains `2` items, the `<div>` with our `{{ donut.name }}` interpolated values will be printed twice, once for each iteration of the data structure. Each value of the loop is stored on the variable `donut` for each iteration.

Angular's `NgFor` is essentially a Directive implementation of JavaScript's ES2015 `for...of` construct, with some additional features.

NgFor and TrackBy Expression

Time to talk about immutable data structures and performance.

In JavaScript two object identities are never the same, despite the fact the may look the same:

[11] https://angular.io/api/common/NgForOf
[12] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols

```
{ name: 'Glazed Fudge' } === { name: 'Glazed Fudge' } // false
['Sour Supreme'] === ['Sour Supreme'] // false
```

This is a good thing, and you've now understood the concept of "object identity".

> Angular favours State Management principles that follow immutable data structures and one-way data flow.

The fact that two objects are never the same means we can easily check if our state has changed, because the identities are different. Combined with `ChangeDetection.OnPush` inside our components, it is a predictable way for us to manage state.

Not only this, but it is a crucial pattern to adopt alongside one-way data flow.

So what does this have to do with `NgFor`? A lot, actually.

Let's assume we have a large list, and our data changes. Angular uses an internal diffing algorithm that now has to check each identity before we can re-render our UI updates.

Once it's done that, Angular will re-render the entire DOM collection as it thinks all of our data has changed (due to the identities changing).

So what if we could tell Angular what's changed so it could re-render more intelligently? That would save vital DOM node destruction and creation time.

With the `trackBy` clause, we can do exactly this.

This would then only re-render the changes necessary, instead of triggering a cascade of DOM manipulation. It's all about performance, and DOM manipulation is costly and kept to a minimum where possible.

It's likely each item we iterate over will contain a unique `id`, so we can pass this to Angular instead of it's default differ, which Angular can then associate each `NgFor` item with:

```
@Component({
  template: `
    <div *ngFor="let donut of donuts; trackBy: trackByFn;">
      {{ donut.name }}
    </div>
  `
})
export class AppComponent {
  donuts: Donut[] = [
    { id: 'x4f0ae', name: 'Glazed Fudge', price: 119 },
    { id: 'o2k95v', name: 'Sour Supreme', price: 129 }
  ];

```

```
14    trackByFn(index: number, value: Donut) {
15      return value.id;
16    }
17  }
```

Now Angular will know if a piece of data has changed, and update the UI accordingly. This is a far better approach to heavy and blind DOM manipulation, especially with how easy the `trackBy` clause is to implement.

Although the `index` is available as the first parameter, I would avoid using this as it will instruct Angular to never re-render that element, which means any lifecycle hooks on components are never going to fire again.

NgFor Variables - Index, Odd, Even

Our "for loop" functionality wouldn't be complete without the ability to access the `index` of each iteration.

Angular also makes some extra utility variables available to us as well to determine if we're dealing with an odd and even iteration.

To start let's access our `index`. This is done via an `as` syntax similar to that seen in `NgIf`:

```
1  <div *ngFor="let donut of donuts; index as i">
2    {{ i }} {{ donut.name }}
3  </div>
```

To access `odd` and `even` variables, which are just `boolean` values, perhaps to set an alternating `class` or `style` property the approach is very similar:

```
1  <div *ngFor="let donut of donuts; odd as o; even as e">
2    {{ o }} {{ e }} {{ donut.name }}
3  </div>
```

To set an alternating `class` called `highlighted` for only our `even` numbers, we could implement as follows:

```
<div
  *ngFor="let donut of donuts; even as e"
  [class.highlighted]="e">
  {{ donut.name }}
</div>
```

NgFor and Async Pipe

It's common to pass an Observable to a template, we've learned this with `NgIf`. We can also pass an Observable directly to an `NgFor` Directive via the `async` pipe.

Whilst possible, it's likely that there's a slightly better pattern to adopt. We'll explore that after.

An Observable can be passed into the `async` pipe with `NgFor` as follows, which creates a subscription:

```
@Component({
  template: `
    <div *ngFor="let donut of (donuts$ | async)">
      {{ donut.name }}
    </div>
  `,
})
export class AppComponent {
  donuts$: Observable<Donut[]> = of([
    { id: 'x4f0ae', name: 'Glazed Fudge', price: 119 },
    { id: 'o2k95v', name: 'Sour Supreme', price: 129 }
  ]);
}
```

If we need to access the `donuts$` data elsewhere in the template, we'll either create another subscription (which is generally an anti-pattern, so we'll avoid that) or have to refactor our code.

Which leads us to a better solution that uses the best of both `NgFor` and `NgIf`:

```
<ng-container *ngIf="(donuts$ | async) as donuts">
  <div *ngFor="let donut of donuts">
    {{ donut.name }}
  </div>
</ng-container>
```

This allows our `NgFor` to deal with just the raw data structure instead of wiring up an Observable subscription.

We then use an `<ng-container>` to unwrap our Observable data source and expand the result as a variable of `donuts`.

NgFor and <ng-template> Syntax

The * syntax alongside `NgFor` is there to provide shorthand sugar syntax for a more complex template definition.

To complete our picture, it's imperative for us to learn what Angular translates our `*NgFor` declaration into.

When we declare `*NgFor` two things are happening; a property binding and template declaration.

For more advanced use cases, we can instead use `<ng-template>` directly, which is what Angular translates our code into behind-the-scenes when compiling our components.

This means that the two examples below are essentially equivalent to eachother:

```
<!-- sugar syntax -->
<div *ngFor="let donut of donuts; index as i; trackBy: trackByFn;">
  {{ donut.name }}
</div>

<!-- expanded directive bindings -->
<ng-template ngFor [ngForOf]="donuts" let-donut let-i="index" [ngForTrackBy]="trackB\
yFn">
  {{ donut.name }}
</ng-template>
```

The above examples are equivalent, however the first example will render a `<div>` around the contents and the `<ng-template>` will not as it contains no element.

This `<ng-template>` example works by declaring `ngFor` as the Directive. Then we pass in the "of" by binding our `donuts` array directly to the `ngForOf` property.

The magical piece here is `let-donut` which holds no value, therefore Angular sets this value as "implicit", which means we use this variable for each iteration value.

We also have `let-i="index"` which allows us to ask for the `index`, and by declaring `let-i` we create a local variable called `i` and assign it the value.

Finally, we bind a `trackByFn` to an `ngForTrackBy` property. You can see why the short hand syntax here is nice and effective.

My preference is to use `*ngFor` consistently and only introduce the `<ng-template>` approach when you have a more complex requirement or strictly need `<ng-template>`.

NgSwitch

NgSwitch[13] is a Structural Directive that allows us to render a specific element or component based on a value.

NgSwitch

The standard `NgSwitch` syntax is `[ngSwitch]="expression"` and provides us with the ability to switch template partials by declaring a case via `*ngSwitchCase`.

We declare `NgSwitch` on an element or component that we wish to provide conditional rendering and pass in an expression to be evaluated by a child `*ngSwitchCase`:

```
@Component({
  template: `
    <div [ngSwitch]="donut.promo">
      <span *ngSwitchCase="'new'">New Arrival</span>
      <span *ngSwitchCase="'limited'">Limited Edition</span>
    </div>
  `
})
export class AppComponent {
  donut: Donut = {
    id: 'x4f0ae', name: 'Glazed Fudge', price: 119, promo: 'new'
  };
}
```

The value of `donut.promo` will is evaluated by each `*ngSwitchCase`, which then matches and renders the appropriate template. Also note here that the `*ngSwitchCase` is a template and not the `[ngSwitch]` binding.

> As we are using the `*` sugar syntax, we are creating a template and binding at the same time and therefore need to specify the strings as literals, i.e. `'new'` instead of `new`.

Angular's `NgSwitch` is essentially a Directive implementation of JavaScript's `switch` and `case`.

[13] https://angular.io/api/common/NgSwitch

NgSwitch and Default Case

If no match is found, we have the opportunity to supply a default case with the `*ngSwitchDefault` Directive.

A default case will be applied any time a match doesn't exist, so is the perfect place to add fallback content, if you require.

```
<div [ngSwitch]="donut.promo">
  <span *ngSwitchCase="'new'">New Arrival</span>
  <span *ngSwitchCase="'limited'">Limited Edition</span>
  <span *ngSwitchDefault>Shown if no matches!</span>
</div>
```

At this point, we have a fully declarative switch case implementation.

NgSwitch and <ng-container>

The opportunity presents itself very nicely with `NgSwitch` to once again turn to the `<ng-container>` element if we do not require a parent element to be rendered.

In our example, the `<div>` we bind `[ngSwitch]` with is also rendered.

Using `<ng-container>` we can keep the generated DOM lighter and simply render the contents only:

```
<ng-container [ngSwitch]="donut.promo">
  <span *ngSwitchCase="'new'">New Arrival</span>
  <span *ngSwitchCase="'limited'">Limited Edition</span>
  <span *ngSwitchDefault>Shown if no matches!</span>
</ng-container>
```

I always opt for this approach, and turn to an element for the initial binding if needed.

We could also introduce an `<ng-container>` on the child element level, though. Why? In the example above we are creating the same `<span>` tag each time.

If we use an `<ng-container>` for the child elements, we can simply swap the text out instead and use the single `<span>` tag as the parent:

```
<span [ngSwitch]="donut.promo">
  <ng-container *ngSwitchCase="'new'">New Arrival</ng-container>
  <ng-container *ngSwitchCase="'limited'">Limited Edition</ng-container>
  <ng-container *ngSwitchDefault>Shown if no matches!</ng-container>
</span>
```

This approach would work well if you always required a `<span>` tag to be rendered and the markup remained the same each time.

NgSwitch and <ng-template> Syntax

The `*` syntax alongside `NgSwitch` (`*ngSwitchCase` and `*ngSwitchDefault`) is there to provide shorthand sugar syntax for a more complex template definition.

To complete our picture, it's imperative for us to learn what Angular translates our `*NgSwitch` declaration into.

When we declare `*NgSwitch` two things are happening; a property binding and template declaration.

The two examples below are essentially equivalent to eachother:

```
<!-- sugar syntax -->
<ng-container [ngSwitch]="donut.promo">
  <span *ngSwitchCase="'new'">New Arrival</span>
  <span *ngSwitchCase="'limited'">Limited Edition</span>
  <span *ngSwitchDefault>Shown if no matches!</span>
</ng-container>

<!-- expanded directive bindings -->
<ng-container [ngSwitch]="donut.promo">
  <ng-template [ngSwitchCase]="'new'">
    <span>New Arrival</span>
  </ng-template>
  <ng-template [ngSwitchCase]="'limited'">
    <span>Limited Edition</span>
  </ng-template>
  <ng-template [ngSwitchDefault]>
    <span>Shown if no matches!</span>
  </ng-template>
</ng-container>
```

The above examples are equivalent, however, the first example minimises our template footprint by creating fewer elements.

This `<ng-template>` example works by declaring `[ngSwitchCase]` as the Directive.

Given the simplicity of `NgSwitch`, there is little difference between the sugar syntax and fully expanded version besides the requirement to create a child `<span>` tag inside each `<ng-template>`.

One small thing to note is that we can opt to not bind to `[ngSwitchCase]` and simply declare it as an attribute instead. This approach allows us to specify string values more easily:

```
<ng-container [ngSwitch]="donut.promo">
  <ng-template ngSwitchCase="new">
    <span>New Arrival</span>
  </ng-template>
  <ng-template ngSwitchCase="limited">
    <span>Limited Edition</span>
  </ng-template>
  <ng-template ngSwitchDefault>
    <span>Shown if no matches!</span>
  </ng-template>
</ng-container>
```

My preference is to use `*ngSwitchCase` consistently and only introduce the `<ng-template>` approach when you have a more complex requirement or strictly need `<ng-template>`.

NgStyle and Style Bindings

NgStyle[14] is an Attribute Directive that allows us to apply inline styles to an element or component.

There are two ways to add inline styles with Angular; using `NgStyle` or via a `[style]` property binding.

Traditionally, `NgStyle` was the first choice for applying inline styles as it supported applying styles via an object. Using `[style]` had this limitation, until recently.

That meant `[style]` was mostly used to add a single style binding at a time. But now `[style]` supports objects too, do we even need `NgStyle`?

Despite this feature parity, `NgStyle` is not currently going anywhere. It may be removed in the future but there are currently no plans.

Another reason to adopt `[style]` over `NgStyle` is that it offers a special inline style property and unit binding. This gives us a lot of power and a consistent approach to work with.

That said, we're still going to cover `NgStyle` - just not like before. And we'll start with `[style]` bindings.

While inline styles are possible, there is also the moral aspect of doing so. It's not always a great approach, and a better solution may be to use a `class` and simply toggle it.

We'll come onto this next with `[class]` and `NgClass` though.

Style Bindings

Create a single inline style by binding to the `style` property along with the property name:

```
@Component({
  template: `
    <div [style.border]="donut.promo ? '2px solid #eee' : 'none'">
      {{ donut.name }}
    </div>
  `
})
export class AppComponent {
  donut: Donut = {
    id: 'x4f0ae', name: 'Glazed Fudge', price: 119, promo: 'new'
  };
}
```

[14] https://angular.io/api/common/NgStyle

We are applying a `border` if the `donut.promo` property evaluates `true`. As it is a `string` and has a `length` it will result in a `true` value when evaluated, therefore the style is added.

I'm using a ternary statement to demonstrate how you can apply different styles based on your condition.

Multiple Style Bindings

It's possible to also specify multiple single styles on the same element:

```
<div
  [style.border]="donut.promo ? '2px solid #eee' : 'none'"
  [style.color]="donut.id ? 'red' : 'blue'">
  {{ donut.name }}
</div>
```

This gives us a flexible approach which is clean, readable, and concise.

Style Bindings and Units

A lesser known feature of `[style]` bindings is to specify the unit of the property as well.

This can be achieved by extending the syntax to include the unit. An example using `font-size`:

```
<div
  [style.font-size.px]="12">
  {{ donut.name }}
</div>
```

There are two very great things happening here.

First, we are passing `12` as a `number`, not `"12"` as a `string`. Why is this good? Dynamic styles. It's highly unlikely you will send back `"12px"` from the server when a simple `12` is perfect. This would also let the user easily change units, if you implemented a custom `font-size` feature that stored the user preference.

Second, note `font-size` and not `fontSize`. Other libraries and frameworks follow different conventions such as `fontSize` in React. This means your code is less portable from `.css` file to component and would require changing to camelCase.

Style Bindings via Objects

The most powerful feature of `[style]` is the ability to pass an object. This adds multiple styles at once:

```
<div
  [style]="{
    border: donut.promo ? '2px solid #eee' : 'none',
    color: donut.id ? 'red' : 'blue'
  }">
  {{ donut.name }}
</div>
```

This object definition is what `NgStyle` was famous for. Now that `style` supports this too, `NgStyle` no longer provides any significant benefit.

Passing an object is clean and concise and can also be abstracted away into the component `class` as well.

Style Bindings from Class Methods

When styles grow larger, they become more difficult to manage. To avoid littering a template with more complexity, we can break out styles into component methods and pass a function as the expression:

```
@Component({
  template: `
    <div [style]="getStyles()">
      {{ donut.name }}
    </div>
  `
})
export class AppComponent {
  donut: Donut = {
    id: 'x4f0ae', name: 'Glazed Fudge', price: 119, promo: 'new'
  };

  getStyles() {
    return {
      border: this.donut.promo ? '2px solid #eee' : 'none',
      color: this.donut.id ? 'red' : 'blue'
    };
  }
}
```

This approach cleans things up nicely and can offer more complex and dynamic logic through means of further methods to manage changes to styles.

Style Bindings from Class Getters

By using a TypeScript getter[15] we can also specify styles to be passed from the component to the template:

```
@Component({
  template: `
    <div [style]="styles">
      {{ donut.name }}
    </div>
  `
})
export class AppComponent {
  donut: Donut = {
    id: 'x4f0ae', name: 'Glazed Fudge', price: 119, promo: 'new'
  };

  get styles() {
    return {
      border: this.donut.promo ? '2px solid #eee' : 'none',
      color: this.donut.id ? 'red' : 'blue'
    };
  }
}
```

I personally like to use the `get` syntax and pass in a property to the template.

Style Bindings from Strings

Inline styles may also be set via a `string` that is formatted like valid CSS property and value:

[15] https://ultimatecourses.com/blog/typescript-setters-getter

```
@Component({
  template: `
    <div [style]="styles">
      {{ donut.name }}
    </div>
  `
})
export class AppComponent {
  donut: Donut = {
    id: 'x4f0ae', name: 'Glazed Fudge', price: 119, promo: 'new'
  };

  get styles() {
    return `
      border: ${this.donut.promo ? '2px solid #eee' : 'none'};
      color: ${this.donut.id ? 'red' : 'blue'};
    `;
  }
}
```

This approach may feel nicer to you if you prefer to not use an object definition.

Just keep in mind that you won't get any TypeScript benefit here like you would with objects.

NgStyle

So what about `NgStyle`? Everything you've learned above with the object syntax can also be applied to `NgStyle`, however there is no real reason to adopt it over `[style]` bindings now.

That said, the following will work just the same as everything above:

```
<div
  [ngStyle]="{
    border: donut.promo ? '2px solid #eee' : 'none',
    color: donut.id ? 'red' : 'blue'
  }">
  {{ donut.name }}
</div>
```

`NgStyle` does not support the same features or provide any reasonable benefit over `[style]`, it actually lacks some of the quick, short and powerful functionality that `[style]` offers.

My preference is to use `[style]` consistently for both single, multiple and object configuration styles.

NgClass and Class Bindings

NgClass[16] is an Attribute Directive that allows us to apply classes to an element or component.

There are two ways to add classes with Angular; using `NgClass` or via a `[class]` property binding.

Traditionally, `NgClass` was the first choice for applying classes as it supported applying classes via an object. Using `[class]` had this limitation, until recently.

That meant `[class]` was mostly used to add a single class at a time. But now `[class]` supports objects too, do we even need `NgClass`?

Despite this feature parity, `NgClass` is not currently going anywhere. It may be removed in the future but there are currently no plans.

Another reason to adopt `[class]` over `NgClass` is that it offers a special class binding. This gives us a lot of power and a consistent approach to work with.

That said, we're still going to cover `NgClass` - just not like before. And we'll start with `[class]` bindings.

Class Bindings

Add a single class by binding to the `class` property along with the property name:

```
@Component({
  template: `
    <div [class.promo]="donut.promo">
      {{ donut.name }}
    </div>
  `,
  styles: [`.promo { border: 2px solid #eee; }`]
})
export class AppComponent {
  donut: Donut = {
    id: 'x4f0ae', name: 'Glazed Fudge', price: 119, promo: 'new'
  };
}
```

We are applying a `promo` class if the `donut.promo` property evaluates `true`. As it is a `string` and has a `length` it will result in a `true` value when evaluated, therefore the class is added.

Should `donut.promo` evaluate `false` no class is added.

[16] https://angular.io/api/common/NgClass

Multiple Class Bindings

It's possible to also specify multiple single class bindings on the same element:

```
<div
  [class.new]="donut.promo === 'new'"
  [class.limited]="donut.promo === 'limited'">
  {{ donut.name }}
</div>
```

This gives us a flexible approach which is clean, readable, and concise.

Class Bindings via Objects

The most powerful feature of `[class]` is the ability to pass an object. This adds multiple classes at once:

```
<div
  [class]="{
    new: donut.promo === 'new',
    limited: donut.promo === 'limited'
  }">
  {{ donut.name }}
</div>
```

This object definition is what `NgClass` was famous for. Now that `class` supports this too, `NgClass` no longer provides any significant benefit.

Passing an object is clean and concise and can also be abstracted away into the component `class` as well.

Class Bindings from Class Methods

When classes grow larger, they become more difficult to manage. To avoid littering a template with more complexity, we can break out classes into component methods and pass a function as the expression:

```
@Component({
  template: `
    <div [class]="getClasses()">
      {{ donut.name }}
    </div>
  `
})
export class AppComponent {
  donut: Donut = {
    id: 'x4f0ae', name: 'Glazed Fudge', price: 119, promo: 'new'
  };

  getClasses() {
    return {
      new: this.donut.promo === 'new',
      limited: this.donut.promo === 'limited'
    };
  }
}
```

This approach cleans things up nicely and can offer more complex and dynamic logic through means of further methods to manage changes to classes.

Class Bindings from Class Getters

By using a TypeScript getter[17] we can also specify classes to be passed from the component to the template:

```
@Component({
  template: `
    <div [class]="classes">
      {{ donut.name }}
    </div>
  `
})
export class AppComponent {
  donut: Donut = {
    id: 'x4f0ae', name: 'Glazed Fudge', price: 119, promo: 'new'
  };

```

[17] https://ultimatecourses.com/blog/typescript-setters-getter

```
  get classes() {
    return {
      new: this.donut.promo === 'new',
      limited: this.donut.promo === 'limited'
    };
  }
}
```

I personally like to use the `get` syntax and pass in a property to the template.

Class Bindings from Strings

Classes may also be set via a `string` that is whitespace separated:

```
@Component({
  template: `
    <div [class]="classes">
      {{ donut.name }}
    </div>
  `
})
export class AppComponent {
  donut: Donut = {
    id: 'x4f0ae', name: 'Glazed Fudge', price: 119, promo: 'new'
  };

  get classes() {
    return `
      ${this.donut.promo === 'new' ? 'new' : ''}
      ${this.donut.promo === 'limited' ? 'limited' : ''}
    `;
  }
}
```

This approach may feel nicer to you if you prefer to not use an object definition.

Just keep in mind that you won't get any TypeScript benefit here like you would with objects.

Class Bindings from Arrays

Classes may also be set via a string `array`:

```
@Component({
  template: `
    <div [class]="classes">
      {{ donut.name }}
    </div>
  `
})
export class AppComponent {
  donut: Donut = {
    id: 'x4f0ae', name: 'Glazed Fudge', price: 119, promo: 'new'
  };

  get classes() {
    return [
      `${this.donut.promo === 'new' ? 'new' : ''}`,
      `${this.donut.promo === 'limited' ? 'limited' : ''}`
    ].filter(Boolean);
  }
}
```

This method uses a ternary statement to declare the class or not. Note the addition of `.filter(Boolean)` to remove any empty strings from the array, this is because the `classList` API that Angular uses under-the-hood will error on empty strings so we must remove them prior to setting.

The `filter(Boolean)` trick works because it returns any truthy values from the array. An empty string is a falsy value and therefore is removed.

NgClass

So what about `NgClass`? Everything you've learned above with the object syntax can also be applied to `NgClass`, however there is no real reason to adopt it over `[class]` bindings now.

That said, the following will work just the same as everything above:

```
<div
  [ngClass]="{
    new: donut.promo === 'new',
    limited: donut.promo === 'limited'
  }">
  {{ donut.name }}
</div>
```

`NgClass` does not support the same features or provide any reasonable benefit over `[class]`, it actually lacks some of the quick, short and powerful functionality that `[class]` offers.

My preference is to use `[class]` consistently for both single, multiple and object configuration classes.