Attribute, class, and style bindings

Attribute binding in Angular helps you set values for attributes directly. With attribute binding, you can improve accessibility, style your application dynamically, and manage multiple CSS classes or styles simultaneously.

See the for a working example containing the code snippets in this guide.

Binding to an attribute

It is recommended that you set an element property with a property binding whenever possible. However, sometimes you don't have an element property to bind. In those situations, use attribute binding.

For example, ARIA and SVG are purely attributes. Neither ARIA nor SVG correspond to element properties and don't set element properties. In these cases, you must use attribute binding because there are no corresponding property targets.

Syntax

Attribute binding syntax resembles property binding, but instead of an element property between brackets, you precede the name of the attribute with the prefix `attr`, followed by a dot. Then, you set the attribute value with an expression that resolves to a string.

<p [attr.attribute-you-are-targeting]="expression"></p>

When the expression resolves to `null` or `undefined`, Angular removes the attribute altogether.

Binding ARIA attributes

One of the primary use cases for attribute binding is to set ARIA attributes, as in this example:

{@a colspan}

Binding to `colspan`

Another common use case for attribute binding is with the `colspan` attribute in tables. Binding to the `colspan` attribute helps you keep your tables programmatically dynamic. Depending on the amount of data that your application populates a table with, the number of columns that a row spans could change.

To use attribute binding with the `<td>` attribute `colspan`:

1. Specify the `colspan` attribute by using the following syntax: `[attr.colspan]`.
2. Set `[attr.colspan]` equal to an expression.

In the following example, you bind the `colspan` attribute to the expression `1 + 1`.

This binding causes the `<td>` to span two columns.

Sometimes there are differences between the name of property and an attribute.

`colspan` is an attribute of `<td>`, while `colSpan` with a capital "S" is a property. When using attribute binding, use `colspan` with a lowercase "s". For more information on how to bind to the `colSpan` property, see the `colspan` and `colSpan` section of Property Binding.

{@a class-binding} ## Binding to the `class` attribute

Use class binding to add and remove CSS class names from an element's `class` attribute.

Binding to a single CSS `class`

To create a single class binding, use the prefix `class` followed by a dot and the name of the CSS class—for example, `[class.sale]="onSale"`. Angular adds the class when the bound expression, `onSale` is truthy, and it removes the class when the expression is falsy—with the exception of `undefined`. See styling delegation for more information.

Binding to multiple CSS classes

To bind to multiple classes, use `[class]` set to an expression—for example, `[class]="classExpression"`. The expression can be one of:

- A space-delimited string of class names.
- An object with class names as the keys and truthy or falsy expressions as the values.
- An array of class names.

With the object format, Angular adds a class only if its associated value is truthy.

With any object-like expression—such as `object`, `Array`, `Map`, or `Set`—the identity of the object must change for Angular to update the class list. Updating the property without changing object identity has no effect.

If there are multiple bindings to the same class name, Angular uses styling precedence to determine which binding to use.

The following table summarizes class binding syntax.

Binding Type

Syntax

Input Type

Example Input Values

Single class binding

[class.sale]="onSale"

boolean | undefined | null

true, false

Multi-class binding

[class]="classExpression"

string

"my-class-1 my-class-2 my-class-3"

Record<string, boolean | undefined | null>

{foo: true, bar: false}

Array<string>

['foo', 'bar']

{@a style-binding} ## Binding to the style attribute

Use style binding to set styles dynamically.

Binding to a single style

To create a single style binding, use the prefix `style` followed by a dot and the name of the CSS style property—for example, `[style.width]="width"`. Angular sets the property to the value of the bound expression, which is usually a string. Optionally, you can add a unit extension like `em` or `%`, which requires a number type.

You can write a style property name in either dash-case, or camelCase.

<nav [style.background-color]="expression"></nav>

<nav [style.backgroundColor]="expression"></nav>

Binding to multiple styles

To toggle multiple styles, bind to the `[style]` attribute—for example, `[style]="styleExpression"`. The `styleExpression` can be one of:

- A string list of styles such as `"width: 100px; height: 100px; background-color: cornflowerblue;"`.
- An object with style names as the keys and style values as the values, such as `{width: '100px', height: '100px', backgroundColor: 'cornflowerblue'}`.

Note that binding an array to `[style]` is not supported.

When binding `[style]` to an object expression, the identity of the object must change for Angular to update the class list. Updating the property without changing object identity has no effect.

Single and multiple-style binding example If there are multiple bindings to the same style attribute, Angular uses styling precedence to determine which binding to use.

The following table summarizes style binding syntax.

Binding Type

Syntax

Input Type

Example Input Values

Single style binding

[style.width]="width"

string | undefined | null

"100px"

Single style binding with units

[style.width.px]="width"

number | undefined | null

100

Multi-style binding

[style]="styleExpression"

string

"width: 100px; height: 100px"

Record<string, string | undefined | null>

{width: '100px', height: '100px'}

The NgStyle directive can be used as an alternative to direct `[style]` bindings. However, using the preceding style binding syntax without `NgStyle` is preferred because due to improvements in style binding in Angular, `NgStyle` no longer provides significant value, and might eventually be removed in the future.

{@a styling-precedence} ## Styling Precedence

A single HTML element can have its CSS class list and style values bound to multiple sources (for example, host bindings from multiple directives).

When there are multiple bindings to the same class name or style property, Angular uses a set of precedence rules to resolve conflicts and determine which classes or styles are ultimately applied to the element.

Styling precedence (highest to lowest)

1. Template bindings
   1. Property binding (for example, `<div [class.foo]="hasFoo">` or `<div [style.color]="color">`)
   2. Map binding (for example, `<div [class]="classExpr">` or `<div [style]="styleExpr">`)
   3. Static value (for example, `<div class="foo">` or `<div style="color: blue">`)
2. Directive host bindings
   1. Property binding (for example, `host: {'[class.foo]': 'hasFoo'}` or `host: {'[style.color]': 'color'}`)
   2. Map binding (for example, `host: {'[class]': 'classExpr'}` or `host: {'[style]': 'styleExpr'}`)
   3. Static value (for example, `host: {'class': 'foo'}` or `host: {'style': 'color: blue'}`)
3. Component host bindings
   1. Property binding (for example, `host: {'[class.foo]': 'hasFoo'}` or `host: {'[style.color]': 'color'}`)
   2. Map binding (for example, `host: {'[class]': 'classExpr'}` or `host: {'[style]': 'styleExpr'}`)
   3. Static value (for example, `host: {'class': 'foo'}` or `host: {'style': 'color: blue'}`)

The more specific a class or style binding is, the higher its precedence.

A binding to a specific class (for example, `[class.foo]`) takes precedence over a generic `[class]` binding, and a binding to a specific style (for example, `[style.bar]`) takes precedence over a generic `[style]` binding.

Specificity rules also apply when it comes to bindings that originate from different sources. It's possible for an element to have bindings in the template where it's declared, from host bindings on matched directives, and from host bindings on matched components.

Template bindings are the most specific because they apply to the element directly and exclusively, so they have the highest precedence.

Directive host bindings are considered less specific because directives can be used in multiple locations, so they have a lower precedence than template bindings.

Directives often augment component behavior, so host bindings from components have the lowest precedence.

In addition, bindings take precedence over static attributes.

In the following case, `class` and `[class]` have similar specificity, but the `[class]` binding takes precedence because it is dynamic.

{@a styling-delegation} ### Delegating to styles with lower precedence

It is possible for higher precedence styles to "delegate" to lower precedence styles using `undefined` values. Whereas setting a style property to `null` ensures the style is removed, setting it to `undefined` causes Angular to fall back to the next-highest precedence binding to that style.

For example, consider the following template:

Imagine that the `dirWithHostBinding` directive and the `comp-with-host-binding` component both have a `[style.width]` host binding. In that case, if `dirWithHostBinding` sets its binding to `undefined`, the `width` property falls back to the value of the `comp-with-host-binding` host binding. However, if `dirWithHostBinding` sets its binding to `null`, the `width` property will be removed entirely.

Injecting attribute values

There are cases where you need to differentiate the behavior of a Component or Directive based on a static value set on the host element as an HTML attribute. For example, you might have a directive that needs to know the `type` of a `<button>` or `<input>` element.

The Attribute parameter decorator is great for passing the value of an HTML attribute to a component/directive constructor using dependency injection.

The injected value captures the value of the specified HTML attribute at that moment. Future updates to the attribute value are not reflected in the injected value.

In the preceding example, the result of `app.component.html` is **The type of the input is: number**.

Another example is the RouterOutlet directive, which makes use of the Attribute decorator to retrieve the unique name on each outlet.

@Attribute() vs @Input()

Remember, use @Input() when you want to keep track of the attribute value and update the associated property. Use @Attribute() when you want to inject the value of an HTML attribute to a component or directive constructor.