Built-in directives

Directives are classes that add additional behavior to elements in your Angular applications. Use Angular's built-in directives to manage forms, lists, styles, and what users see.

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

The different types of Angular directives are as follows:

1. Components—directives with a template. This type of directive is the most common directive type.
2. Attribute directives—directives that change the appearance or behavior of an element, component, or another directive.
3. Structural directives—directives that change the DOM layout by adding and removing DOM elements.

This guide covers built-in attribute directives and structural directives.

{@a attribute-directives} ## Built-in attribute directives

Attribute directives listen to and modify the behavior of other HTML elements, attributes, properties, and components.

Many NgModules such as the `RouterModule` and the `FormsModule` define their own attribute directives. The most common attribute directives are as follows:

- `NgClass`—adds and removes a set of CSS classes.
- `NgStyle`—adds and removes a set of HTML styles.
- `NgModel`—adds two-way data binding to an HTML form element.

Built-in directives use only public APIs. They do not have special access to any private APIs that other directives can't access.

{@a ngClass} ## Adding and removing classes with `NgClass`

Add or remove multiple CSS classes simultaneously with `ngClass`.

To add or remove a *single* class, use class binding rather than `NgClass`.

Using `NgClass` with an expression

On the element you'd like to style, add `[ngClass]` and set it equal to an expression. In this case, `isSpecial` is a boolean set to `true` in `app.component.ts`. Because `isSpecial` is true, `ngClass` applies the class of `special` to the `<div>`.

Using `NgClass` with a method

1. To use `NgClass` with a method, add the method to the component class. In the following example, `setCurrentClasses()` sets the property `currentClasses` with an object that adds or removes three classes based on the `true` or `false` state of three other component properties.

Each key of the object is a CSS class name. If a key is `true`, `ngClass` adds the class. If a key is `false`, `ngClass` removes the class.

1. In the template, add the `ngClass` property binding to `currentClasses` to set the element's classes:

For this use case, Angular applies the classes on initialization and in case of changes. The full example calls `setCurrentClasses()` initially with `ngOnInit()` and when the dependent properties change through a button click. These steps are not necessary to implement `ngClass`. For more information, see the `app.component.ts` and `app.component.html`.

{@a ngstyle} ## Setting inline styles with `NgStyle`

Use `NgStyle` to set multiple inline styles simultaneously, based on the state of the component.

1. To use `NgStyle`, add a method to the component class.

In the following example, `setCurrentStyles()` sets the property `currentStyles` with an object that defines three styles, based on the state of three other component properties.

1. To set the element's styles, add an `ngStyle` property binding to `currentStyles`.

For this use case, Angular applies the styles upon initialization and in case of changes. To do this, the full example calls `setCurrentStyles()` initially with `ngOnInit()` and when the dependent properties change through a button click. However, these steps are not necessary to implement `ngStyle` on its own. See the `app.component.ts` and `app.component.html` for this optional implementation.

{@a ngModel} ## Displaying and updating properties with `ngModel`

Use the `NgModel` directive to display a data property and update that property when the user makes changes.

1. Import `FormsModule` and add it to the NgModule's `imports` list.

1. Add an `[(ngModel)]` binding on an HTML `<form>` element and set it equal to the property, here `name`.

This `[(ngModel)]` syntax can only set a data-bound property.

To customize your configuration, write the expanded form, which separates the property and event binding. Use property binding to set the property and event binding to respond to changes. The following example changes the `<input>` value to uppercase:

Here are all variations in action, including the uppercase version:

`NgModel` and value accessors

The `NgModel` directive works for an element supported by a ControlValueAccessor. Angular provides *value accessors* for all of the basic HTML form elements. For more information, see Forms.

To apply `[(ngModel)]` to a non-form built-in element or a third-party custom component, you have to write a value accessor. For more information, see the API documentation on DefaultValueAccessor.

When you write an Angular component, you don't need a value accessor or `NgModel` if you name the value and event properties according to Angular's two-way binding syntax.

{@a structural-directives}

Built-in structural directives

Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, and manipulating the host elements to which they are attached.

This section introduces the most common built-in structural directives:

- `NgIf`—conditionally creates or disposes of subviews from the template.
- `NgFor`—repeat a node for each item in a list.
- `NgSwitch`—a set of directives that switch among alternative views.

For more information, see Structural Directives.

{@a ngIf} ## Adding or removing an element with `NgIf`

Add or remove an element by applying an `NgIf` directive to a host element.

When `NgIf` is `false`, Angular removes an element and its descendants from the DOM. Angular then disposes of their components, which frees up memory and resources.

To add or remove an element, bind `*ngIf` to a condition expression such as `isActive` in the following example.

When the `isActive` expression returns a truthy value, `NgIf` adds the `ItemDetailComponent` to the DOM. When the expression is falsy, `NgIf` removes the `ItemDetailComponent` from the DOM and disposes of the component and all of its sub-components.

For more information on `NgIf` and `NgIfElse`, see the NgIf API documentation.

Guarding against `null`

By default, `NgIf` prevents display of an element bound to a null value.

To use NgIf to guard a , add *ngIf="yourProperty" to the . In the following example, the currentCustomer name appears because there is a currentCustomer.

However, if the property is null, Angular does not display the . In this example, Angular does not display the nullCustomer because it is null.

{@a ngFor} ## Listing items with NgFor

Use the NgFor directive to present a list of items.

1. Define a block of HTML that determines how Angular renders a single item.
2. To list your items, assign the short hand let item of items to *ngFor.

The string "let item of items" instructs Angular to do the following:

- Store each item in the items array in the local item looping variable
- Make each item available to the templated HTML for each iteration
- Translate "let item of items" into an <ng-template> around the host element
- Repeat the <ng-template> for each item in the list

For more information see the Structural directive shorthand section of Structural directives. ### Repeating a component view

To repeat a component element, apply *ngFor to the selector. In the following example, the selector is <app-item-detail>.

Reference a template input variable, such as item, in the following locations:

- within the ngFor host element
- within the host element descendants to access the item's properties

The following example references item first in an interpolation and then passes in a binding to the item property of the <app-item-detail> component.

For more information about template input variables, see Structural directive shorthand.

Getting the index of *ngFor

Get the index of *ngFor in a template input variable and use it in the template.

In the *ngFor, add a semicolon and let i=index to the short hand. The following example gets the index in a variable named i and displays it with the item name.

The index property of the NgFor directive context returns the zero-based index of the item in each iteration.

Angular translates this instruction into an <ng-template> around the host element, then uses this template repeatedly to create a new set of elements and

bindings for each `item` in the list. For more information about shorthand, see the Structural Directives guide.

{@a one-per-element} ## Repeating elements when a condition is true

To repeat a block of HTML when a particular condition is true, put the `*ngIf` on a container element that wraps an `*ngFor` element. One or both elements can be an `<ng-container>` so you don't have to introduce extra levels of HTML.

Because structural directives add and remove nodes from the DOM, apply only one structural directive per element.

For more information about `NgFor` see the NgForOf API reference.

{@a ngfor-with-trackby} ### Tracking items with `*ngFor trackBy`

Reduce the number of calls your application makes to the server by tracking changes to an item list. With the `*ngFor trackBy` property, Angular can change and re-render only those items that have changed, rather than reloading the entire list of items.

1. Add a method to the component that returns the value `NgFor` should track. In this example, the value to track is the item's `id`. If the browser has already rendered `id`, Angular keeps track of it and doesn't re-query the server for the same `id`.

1. In the short hand expression, set `trackBy` to the `trackByItems()` method.

Change ids creates new items with new `item.id`s. In the following illustration of the `trackBy` effect, **Reset items** creates new items with the same `item.id`s.

- With no `trackBy`, both buttons trigger complete DOM element replacement.
- With `trackBy`, only changing the `id` triggers element replacement.

{@a ngcontainer}

Hosting a directive without a DOM element

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.

Use `<ng-container>` when there's no single element to host the directive.

Here's a conditional paragraph using `<ng-container>`.

1. Import the `ngModel` directive from `FormsModule`.

2. Add `FormsModule` to the imports section of the relevant Angular module.

3. To conditionally exclude an `<option>`, wrap the `<option>` in an `<ng-container>`.

```
<img src='generated/images/guide/structural-directives/select-ngcontainer-anim.gif' alt="ngc
```

{@a ngSwitch} ## Switching cases with `NgSwitch`

Like the JavaScript `switch` statement, `NgSwitch` displays one element from among several possible elements, based on a switch condition. Angular puts only the selected element into the DOM. `NgSwitch` is a set of three directives:

- `NgSwitch`—an attribute directive that changes the behavior of its companion directives.
- `NgSwitchCase`—structural directive that adds its element to the DOM when its bound value equals the switch value and removes its bound value when it doesn't equal the switch value.
- `NgSwitchDefault`—structural directive that adds its element to the DOM when there is no selected `NgSwitchCase`.

1. On an element, such as a `<div>`, add `[ngSwitch]` bound to an expression that returns the switch value, such as `feature`. Though the `feature` value in this example is a string, the switch value can be of any type.

2. Bind to `*ngSwitchCase` and `*ngSwitchDefault` on the elements for the cases.

1. In the parent component, define `currentItem`, to use it in the `[ngSwitch]` expression.

1. In each child component, add an `item` input property which is bound to the `currentItem` of the parent component. The following two snippets show the parent component and one of the child components. The other child components are identical to `StoutItemComponent`.

```
<img src="generated/images/guide/built-in-directives/ngswitch.gif" alt="Animation of NgSwitc
```

Switch directives also work with built-in HTML elements and web components. For example, you could replace the `<app-best-item>` switch case with a `<div>` as follows.

What's next

For information on how to build your own custom directives, see Attribute Directives and Structural Directives.