Template Syntax

The Angular application manages what the user sees and can do, achieving this through the interaction of a component class instance (the *component*) and its user-facing template.

You may be familiar with the component/template duality from your experience with model-view-controller (MVC) or model-view-viewmodel (MVVM). In Angular, the component plays the part of the controller/viewmodel, and the template represents the view.

This page is a comprehensive technical reference to the Angular template language. It explains basic principles of the template language and describes most of the syntax that you'll encounter elsewhere in the documentation.

Many code snippets illustrate the points and concepts, all of them available in the .

{@a html}

HTML in templates

HTML is the language of the Angular template. Almost all HTML syntax is valid template syntax. The `<script>` element is a notable exception; it is forbidden, eliminating the risk of script injection attacks. In practice, `<script>` is ignored and a warning appears in the browser console. See the Security page for details.

Some legal HTML doesn't make much sense in a template. The `<html>`, `<body>`, and `<base>` elements have no useful role. Pretty much everything else is fair game.

You can extend the HTML vocabulary of your templates with components and directives that appear as new elements and attributes. In the following sections, you'll learn how to get and set DOM (Document Object Model) values dynamically through data binding.

Begin with the first form of data binding—interpolation—to see how much richer template HTML can be.

{@a interpolation}

Interpolation (`{{...}}`)

You met the double-curly braces of interpolation, `{{` and `}}`, early in your Angular education.

You use interpolation to weave calculated strings into the text between HTML element tags and within attribute assignments.

The text between the braces is often the name of a component property. Angular replaces that name with the string value of the corresponding component property. In the example above, Angular evaluates the `title` and `heroImageUrl` properties and "fills in the blanks", first displaying a bold application title and then a heroic image.

More generally, the text between the braces is a **template expression** that Angular first **evaluates** and then **converts to a string**. The following interpolation illustrates the point by adding the two numbers:

The expression can invoke methods of the host component such as `getVal()`, seen here:

Angular evaluates all expressions in double curly braces, converts the expression results to strings, and links them with neighboring literal strings. Finally, it assigns this composite interpolated result to an **element or directive property**.

You appear to be inserting the result between element tags and assigning it to attributes. It's convenient to think so, and you rarely suffer for this mistake. Though this is not exactly true. Interpolation is a special syntax that Angular converts into a property binding, as is explained below.

But first, let's take a closer look at template expressions and statements.

{@a template-expressions}

Template expressions

A template **expression** produces a value. Angular executes the expression and assigns it to a property of a binding target; the target might be an HTML element, a component, or a directive.

The interpolation braces in `{{1 + 1}}` surround the template expression `1 + 1`. In the property binding section below, a template expression appears in quotes to the right of the `=` symbol as in `[property]="expression"`.

You write these template expressions in a language that looks like JavaScript. Many JavaScript expressions are legal template expressions, but not all.

JavaScript expressions that have or promote side effects are prohibited, including:

- assignments (`=` , `+=` , `-=` , ...)
- `new`
- chaining expressions with `;` or `,`
- increment and decrement operators (`++` and `--`)

Other notable differences from JavaScript syntax include:

- no support for the bitwise operators `|` and `&`
- new template expression operators, such as `|` , `?.` and `!` .

{@a expression-context}

Expression context

The *expression context* is typically the *component* instance. In the following snippets, the `title` within double-curly braces and the `isUnchanged` in quotes refer to properties of the `AppComponent` .

An expression may also refer to properties of the *template's* context such as a template input variable (`let hero`) or a template reference variable (`#heroInput`).

The context for terms in an expression is a blend of the *template variables*, the directive's *context* object (if it has one), and the component's *members*. If you reference a name that belongs to more than one of these namespaces, the template variable name takes precedence, followed by a name in the directive's *context*, and, lastly, the component's member names.

The previous example presents such a name collision. The component has a `hero` property and the `*ngFor` defines a `hero` template variable. The `hero` in `{{hero.name}}` refers to the template input variable, not the component's property.

Template expressions cannot refer to anything in the global namespace (except `undefined`). They can't refer to `window` or `document` . They can't call `console.log` or `Math.max` . They are restricted to referencing members of the expression context.

{@a no-side-effects}

{@a expression-guidelines}

Expression guidelines

Template expressions can make or break an application. Please follow these guidelines:

- No visible side effects

- Quick execution
- Simplicity
- Idempotence

The only exceptions to these guidelines should be in specific circumstances that you thoroughly understand.

No visible side effects

A template expression should not change any application state other than the value of the target property.

This rule is essential to Angular's "unidirectional data flow" policy. You should never worry that reading a component value might change some other displayed value. The view should be stable throughout a single rendering pass.

Quick execution

Angular executes template expressions after every change detection cycle. Change detection cycles are triggered by many asynchronous activities such as promise resolutions, http results, timer events, keypresses and mouse moves.

Expressions should finish quickly or the user experience may drag, especially on slower devices. Consider caching values when their computation is expensive.

Simplicity

Although it's possible to write quite complex template expressions, you should avoid them.

A property name or method call should be the norm. An occasional Boolean negation (`!`) is OK. Otherwise, confine application and business logic to the component itself, where it will be easier to develop and test.

Idempotence

An idempotent expression is ideal because it is free of side effects and improves Angular's change detection performance.

In Angular terms, an idempotent expression always returns *exactly the same thing* until one of its dependent values changes.

Dependent values should not change during a single turn of the event loop. If an idempotent expression returns a string or a number, it returns the same string or number when called twice in a row. If the expression returns an object (including an `array`), it returns the same object *reference* when called twice in a row.

{@a template-statements}

Template statements

A template **statement** responds to an **event** raised by a binding target such as an element, component, or directive. You'll see template statements in the event binding section, appearing in quotes to the right of the `=` symbol as in `(event)="statement"`.

A template statement *has a side effect*. That's the whole point of an event. It's how you update application state from user action.

Responding to events is the other side of Angular's "unidirectional data flow". You're free to change anything, anywhere, during this turn of the event loop.

Like template expressions, template *statements* use a language that looks like JavaScript. The template statement parser differs from the template expression parser and specifically supports both basic assignment (`=`) and chaining expressions (with `;` or `,`).

However, certain JavaScript syntax is not allowed:

- `new`
- increment and decrement operators, `++` and `--`
- operator assignment, such as `+=` and `-=`
- the bitwise operators `|` and `&`
- the template expression operators

Statement context

As with expressions, statements can refer only to what's in the statement context such as an event handling method of the component instance.

The *statement context* is typically the component instance. The *deleteHero* in `(click)="deleteHero()"` is a method of the data-bound component.

The statement context may also refer to properties of the template's own context. In the following examples, the template `$event` object, a template input variable (`let hero`), and a template reference variable (`#heroForm`) are passed to an event handling method of the component.

Template context names take precedence over component context names. In `deleteHero(hero)` above, the `hero` is the template input variable, not the component's `hero` property.

Template statements cannot refer to anything in the global namespace. They can't refer to `window` or `document`. They can't call `console.log` or `Math.max`.

Statement guidelines

As with expressions, avoid writing complex template statements. A method call or simple property assignment should be the norm.

Now that you have a feel for template expressions and statements, you're ready to learn about the varieties of data binding syntax beyond interpolation.

{@a binding-syntax}

Binding syntax: An overview

Data binding is a mechanism for coordinating what users see, with application data values. While you could push values to and pull values from HTML, the application is easier to write, read, and maintain if you turn these chores over to a binding framework. You simply declare bindings between binding sources and target HTML elements and let the framework do the work.

Angular provides many kinds of data binding. This guide covers most of them, after a high-level view of Angular data binding and its syntax.

Binding types can be grouped into three categories distinguished by the direction of data flow: from the *source-to-view*, from *view-to-source*, and in the two-way sequence: *view-to-source-to-view*:

Data direction	Syntax	Type
One-way from data source to view target	{{expression}} [target]="expression" bind-target="expression"	Interpolation Property Attribute Class Style
One-way from view target to data source	(target)="statement" on-target="statement"	Event
Two-way	[(target)]="expression" bindon-target="expression"	Two-way

Binding types other than interpolation have a **target name** to the left of the equal sign, either surrounded by punctuation (`[]` , `()`) or preceded by a prefix (`bind-` , `on-` , `bindon-`).

The target name is the name of a *property*. It may look like the name of an *attribute* but it never is. To appreciate the difference, you must develop a new way to think about template HTML.

A new mental model

With all the power of data binding and the ability to extend the HTML vocabulary with custom markup, it is tempting to think of template HTML as *HTML Plus*.

It really *is* HTML Plus. But it's also significantly different than the HTML you're used to. It requires a new mental model.

In the normal course of HTML development, you create a visual structure with HTML elements, and you modify those elements by setting element attributes with string constants.

You still create a structure and initialize attribute values this way in Angular templates.

Then you learn to create new elements with components that encapsulate HTML and drop them into templates as if they were native HTML elements.

That's HTML Plus.

Then you learn about data binding. The first binding you meet might look like this:

You'll get to that peculiar bracket notation in a moment. Looking beyond it, your intuition suggests that you're binding to the button's `disabled` attribute and setting it to the current value of the component's `isUnchanged` property.

Your intuition is incorrect! Your everyday HTML mental model is misleading. In fact, once you start data binding, you are no longer working with HTML *attributes*. You aren't setting attributes. You are setting the *properties* of DOM elements, components, and directives.

HTML attribute vs. DOM property The distinction between an HTML attribute and a DOM property is crucial to understanding how Angular binding works. **Attributes are defined by HTML. Properties are defined by the DOM (Document Object Model).** * A few HTML attributes have 1:1 mapping to properties. `id` is one example. * Some HTML attributes don't have corresponding properties. `colspan` is one example. * Some DOM properties don't have corresponding attributes. `textContent` is one example. * Many HTML attributes appear to map to properties ... but not in the way you might think! That last category is confusing until you grasp this general rule: **Attributes *initialize* DOM properties and then they are done. Property values can change; attribute values can't.** For example, when the browser renders `<input type="text" value="Bob">`, it creates a

corresponding DOM node with a `value` property *initialized* to "Bob". When the user enters "Sally" into the input box, the DOM element `value` *property* becomes "Sally". But the HTML `value` *attribute* remains unchanged as you discover if you ask the input element about that attribute: `input.getAttribute('value')` returns "Bob". The HTML attribute `value` specifies the *initial* value; the DOM `value` property is the *current* value. The `disabled` attribute is another peculiar example. A button's `disabled` *property* is `false` by default so the button is enabled. When you add the `disabled` *attribute*, its presence alone initializes the button's `disabled` *property* to `true` so the button is disabled. Adding and removing the `disabled` *attribute* disables and enables the button. The value of the *attribute* is irrelevant, which is why you cannot enable a button by writing ` Still Disabled `. Setting the button's `disabled` *property* (say, with an Angular binding) disables or enables the button. The value of the *property* matters. **The HTML attribute and the DOM property are not the same thing, even when they have the same name.**

This fact bears repeating: **Template binding works with *properties* and *events*, not *attributes*.**

A world without attributes
In the world of Angular, the only role of attributes is to initialize element and directive state. When you write a data binding, you're dealing exclusively with properties and events of the target object. HTML attributes effectively disappear.

With this model firmly in mind, read on to learn about binding targets.

Binding targets

The **target of a data binding** is something in the DOM. Depending on the binding type, the target can be an (element | component | directive) property, an (element | component | directive) event, or (rarely) an attribute name. The following table summarizes:

Type	Target	Examples
Property	Element property Component property Directive property	
Event	Element event Component event Directive event	
Two-way	Event and property	
Attribute	Attribute (the exception)	
Class	`class` property	
Style	`style` property	

With this broad view in mind, you're ready to look at binding types in detail.

{@a property-binding}

Property binding (`[property]`)

Write a template **property binding** to set a property of a view element. The binding sets the property to the value of a template expression.

The most common property binding sets an element property to a component property value. An example is binding the `src` property of an image element to a component's `heroImageUrl` property:

Another example is disabling a button when the component says that it `isUnchanged` :

Another is setting a property of a directive:

Yet another is setting the model property of a custom component (a great way for parent and child components to communicate):

One-way *in*

People often describe property binding as *one-way data binding* because it flows a value in one direction, from a component's data property into a target element property.

You cannot use property binding to pull values *out* of the target element. You can't bind to a property of the target element to *read* it. You can only *set* it.

Similarly, you cannot use property binding to *call* a method on the target element. If the element raises events, you can listen to them with an [event binding](guide/template-syntax#event-binding). If you must read a target element property or call one of its methods, you'll need a different technique. See the API reference for [ViewChild](api/core/ViewChild) and [ContentChild](api/core/ContentChild).

Binding target

An element property between enclosing square brackets identifies the target property. The target property in the following code is the image element's `src` property.

Some people prefer the `bind-` prefix alternative, known as the *canonical form*:

The target name is always the name of a property, even when it appears to be the name of something else. You see `src` and may think it's the name of an attribute. No. It's the name of an image element property.

Element properties may be the more common targets, but Angular looks first to see if the name is a property of a known directive, as it is in the following example:

Technically, Angular is matching the name to a directive [input](guide/template-syntax#inputs-outputs), one of the property names listed in the directive's `inputs` array or a property decorated with `@Input()`. Such inputs map to the directive's own properties.

If the name fails to match a property of a known directive or element, Angular reports an “unknown directive” error.

Avoid side effects

As mentioned previously, evaluation of a template expression should have no visible side effects. The expression language itself does its part to keep you safe. You can't assign a value to anything in a property binding expression nor use the increment and decrement operators.

Of course, the expression might invoke a property or method that has side effects. Angular has no way of knowing that or stopping you.

The expression could call something like `getFoo()`. Only you know what `getFoo()` does. If `getFoo()` changes something and you happen to be binding to that something, you risk an unpleasant

experience. Angular may or may not display the changed value. Angular may detect the change and throw a warning error. In general, stick to data properties and to methods that return values and do no more.

Return the proper type

The template expression should evaluate to the type of value expected by the target property. Return a string if the target property expects a string. Return a number if the target property expects a number. Return an object if the target property expects an object.

The `hero` property of the `HeroDetail` component expects a `Hero` object, which is exactly what you're sending in the property binding:

Remember the brackets

The brackets tell Angular to evaluate the template expression. If you omit the brackets, Angular treats the string as a constant and *initializes the target property* with that string. It does *not* evaluate the string!

Don't make the following mistake:

{@a one-time-initialization}

One-time string initialization

You *should* omit the brackets when all of the following are true:

- The target property accepts a string value.
- The string is a fixed value that you can bake into the template.
- This initial value never changes.

You routinely initialize attributes this way in standard HTML, and it works just as well for directive and component property initialization. The following example initializes the `prefix` property of the `HeroDetailComponent` to a fixed string, not a template expression. Angular sets it and forgets about it.

The `[hero]` binding, on the other hand, remains a live binding to the component's `currentHero` property.

{@a property-binding-or-interpolation}

Property binding or interpolation?

You often have a choice between interpolation and property binding. The following binding pairs do the same thing:

Interpolation is a convenient alternative to *property binding* in many cases.

When rendering data values as strings, there is no technical reason to prefer one form to the other. You lean toward readability, which tends to favor interpolation. You suggest establishing coding style rules and choosing the form that both conforms to the rules and feels most natural for the task at hand.

When setting an element property to a non-string data value, you must use *property binding*.

Content security

Imagine the following *malicious content*.

Fortunately, Angular data binding is on alert for dangerous HTML. It *sanitizes* the values before displaying them. It **will not** allow HTML with script tags to leak into the browser, neither with interpolation nor property binding.

Interpolation handles the script tags differently than property binding but both approaches render the content harmlessly.

{@a other-bindings}

Attribute, class, and style bindings

The template syntax provides specialized one-way bindings for scenarios less well suited to property binding.

Attribute binding

You can set the value of an attribute directly with an **attribute binding**.

This is the only exception to the rule that a binding sets a target property. This is the only binding that creates and sets an attribute.

This guide stresses repeatedly that setting an element property with a property binding is always preferred to setting the attribute with a string. Why does Angular offer attribute binding?

You must use attribute binding when there is no element property to bind.

Consider the ARIA, SVG, and table span attributes. They are pure attributes. They do not correspond to element properties, and they do not set element properties. There are no property targets to bind to.

This fact becomes painfully obvious when you write something like this.

<tr><td colspan="{{1 + 1}}">Three-Four</td></tr>

And you get this error:

Template parse errors: Can't bind to 'colspan' since it isn't a known native property

As the message says, the `<td>` element does not have a `colspan` property. It has the "colspan" *attribute*, but interpolation and property binding can set only *properties*, not attributes.

You need attribute bindings to create and bind to such attributes.

Attribute binding syntax resembles property binding. Instead of an element property between brackets, start with the prefix **`attr`**, followed by a dot (`.`) and the name of the attribute. You then set the attribute value, using an expression that resolves to a string.

Bind `[attr.colspan]` to a calculated value:

Here's how the table renders:

One-Two	
Five	Six

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

Class binding

You can add and remove CSS class names from an element's `class` attribute with a **class binding**.

Class binding syntax resembles property binding. Instead of an element property between brackets, start with the prefix `class`, optionally followed by a dot (`.`) and the name of a CSS class: `[class.class-name]`.

The following examples show how to add and remove the application's "special" class with class bindings. Here's how to set the attribute without binding:

You can replace that with a binding to a string of the desired class names; this is an all-or-nothing, replacement binding.

Finally, you can bind to a specific class name. Angular adds the class when the template expression evaluates

to truthy. It removes the class when the expression is falsy.

While this is a fine way to toggle a single class name, the [NgClass directive](guide/template-syntax#ngClass) is usually preferred when managing multiple class names at the same time.

Style binding

You can set inline styles with a **style binding**.

Style binding syntax resembles property binding. Instead of an element property between brackets, start with the prefix `style`, followed by a dot (`.`) and the name of a CSS style property: `[style.style-property]`.

Some style binding styles have a unit extension. The following example conditionally sets the font size in “em” and “%” units .

While this is a fine way to set a single style, the [NgStyle directive](guide/template-syntax#ngStyle) is generally preferred when setting several inline styles at the same time.
Note that a _style property_ name can be written in either [dash-case](guide/glossary#dash-case), as shown above, or [camelCase](guide/glossary#camelcase), such as `fontSize`.

{@a event-binding}

Event binding (`(event)`)

The bindings directives you've met so far flow data in one direction: **from a component to an element**.

Users don't just stare at the screen. They enter text into input boxes. They pick items from lists. They click buttons. Such user actions may result in a flow of data in the opposite direction: **from an element to a component**.

The only way to know about a user action is to listen for certain events such as keystrokes, mouse movements, clicks, and touches. You declare your interest in user actions through Angular event binding.

Event binding syntax consists of a **target event** name within parentheses on the left of an equal sign, and a quoted template statement on the right. The following event binding listens for the button's click events, calling the component's `onSave()` method whenever a click occurs:

Target event

A **name between parentheses** — for example, `(click)` — identifies the target event. In the following example, the target is the button's click event.

Some people prefer the `on-` prefix alternative, known as the **canonical form**:

Element events may be the more common targets, but Angular looks first to see if the name matches an event property of a known directive, as it does in the following example:

The `myClick` directive is further described in the section on [aliasing input/output properties](guide/template-syntax#aliasing-io).

If the name fails to match an element event or an output property of a known directive, Angular reports an “unknown directive” error.

$event and event handling statements

In an event binding, Angular sets up an event handler for the target event.

When the event is raised, the handler executes the template statement. The template statement typically involves a receiver, which performs an action in response to the event, such as storing a value from the HTML control into a model.

The binding conveys information about the event, including data values, through an **event object named** `$event`.

The shape of the event object is determined by the target event. If the target event is a native DOM element event, then `$event` is a DOM event object, with properties such as `target` and `target.value`.

Consider this example:

This code sets the input box `value` property by binding to the `name` property. To listen for changes to the value, the code binds to the input box's `input` event. When the user makes changes, the `input` event is raised, and the binding executes the statement within a context that includes the DOM event object, `$event`.

To update the `name` property, the changed text is retrieved by following the path `$event.target.value`.

If the event belongs to a directive (recall that components are directives), `$event` has whatever shape the directive decides to produce.

{@a eventemitter}

{@a custom-event}

Custom events with `EventEmitter`

Directives typically raise custom events with an Angular EventEmitter. The directive creates an `EventEmitter` and exposes it as a property. The directive calls `EventEmitter.emit(payload)` to fire an event, passing in a message payload, which can be anything. Parent directives listen for the event by binding to this property and accessing the payload through the `$event` object.

Consider a `HeroDetailComponent` that presents hero information and responds to user actions. Although the `HeroDetailComponent` has a delete button it doesn't know how to delete the hero itself. The best it can do is raise an event reporting the user's delete request.

Here are the pertinent excerpts from that `HeroDetailComponent`:

The component defines a `deleteRequest` property that returns an `EventEmitter`. When the user clicks *delete*, the component invokes the `delete()` method, telling the `EventEmitter` to emit a `Hero` object.

Now imagine a hosting parent component that binds to the `HeroDetailComponent`'s `deleteRequest` event.

When the `deleteRequest` event fires, Angular calls the parent component's `deleteHero` method, passing the *hero-to-delete* (emitted by `HeroDetail`) in the `$event` variable.

Template statements have side effects

The `deleteHero` method has a side effect: it deletes a hero. Template statement side effects are not just OK, but expected.

Deleting the hero updates the model, perhaps triggering other changes including queries and saves to a remote server. These changes percolate through the system and are ultimately displayed in this and other views.

{@a two-way}

Two-way binding (`[(...)]`)

You often want to both display a data property and update that property when the user makes changes.

On the element side that takes a combination of setting a specific element property and listening for an element change event.

Angular offers a special *two-way data binding* syntax for this purpose, `[(x)]`. The `[(x)]` syntax combines the brackets of *property binding*, `[x]`, with the parentheses of *event binding*, `(x)`.

[( )] = banana in a box
Visualize a *banana in a box* to remember that the parentheses go _inside_ the brackets.

The `[(x)]` syntax is easy to demonstrate when the element has a settable property called `x` and a corresponding event named `xChange`. Here's a `SizerComponent` that fits the pattern. It has a `size` value property and a companion `sizeChange` event:

The initial `size` is an input value from a property binding. Clicking the buttons increases or decreases the `size`, within min/max values constraints, and then raises (*emits*) the `sizeChange` event with the adjusted size.

Here's an example in which the `AppComponent.fontSizePx` is two-way bound to the `SizerComponent`:

The `AppComponent.fontSizePx` establishes the initial `SizerComponent.size` value. Clicking the buttons updates the `AppComponent.fontSizePx` via the two-way binding. The revised `AppComponent.fontSizePx` value flows through to the *style* binding, making the displayed text bigger or smaller.

The two-way binding syntax is really just syntactic sugar for a *property* binding and an *event* binding. Angular *desugars* the `SizerComponent` binding into this:

The `$event` variable contains the payload of the `SizerComponent.sizeChange` event. Angular assigns the `$event` value to the `AppComponent.fontSizePx` when the user clicks the buttons.

Clearly the two-way binding syntax is a great convenience compared to separate property and event bindings.

It would be convenient to use two-way binding with HTML form elements like `<input>` and `<select>`. However, no native HTML element follows the `x` value and `xChange` event pattern.

Fortunately, the Angular *NgModel* directive is a bridge that enables two-way binding to form elements.

{@a directives}

Built-in directives

Earlier versions of Angular included over seventy built-in directives. The community contributed many more, and countless private directives have been created for internal applications.

You don't need many of those directives in Angular. You can often achieve the same results with the more capable and expressive Angular binding system. Why create a directive to handle a click when you can write a simple binding such as this?

You still benefit from directives that simplify complex tasks. Angular still ships with built-in directives; just not as many. You'll write your own directives, just not as many.

This segment reviews some of the most frequently used built-in directives, classified as either *attribute* directives or *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. They are usually applied to elements as if they were HTML attributes, hence the name.

Many details are covered in the *Attribute Directives* guide. Many NgModules such as the `RouterModule` and the `FormsModule` define their own attribute directives. This section is an introduction to the most commonly used attribute directives:

- `NgClass` - add and remove a set of CSS classes
- `NgStyle` - add and remove a set of HTML styles
- `NgModel` - two-way data binding to an HTML form element

{@a ngClass}

NgClass

You typically control how elements appear by adding and removing CSS classes dynamically. You can bind to the `ngClass` to add or remove several classes simultaneously.

A class binding is a good way to add or remove a *single* class.

To add or remove *many* CSS classes at the same time, the `NgClass` directive may be the better choice.

Try binding `ngClass` to a key:value control object. Each key of the object is a CSS class name; its value is

`true` if the class should be added, `false` if it should be removed.

Consider a `setCurrentClasses` component method that sets a component property, `currentClasses` with an object that adds or removes three classes based on the `true` / `false` state of three other component properties:

Adding an `ngClass` property binding to `currentClasses` sets the element's classes accordingly:

It's up to you to call `setCurrentClasses()`, both initially and when the dependent properties change.

{@a ngStyle}

NgStyle

You can set inline styles dynamically, based on the state of the component. With `NgStyle` you can set many inline styles simultaneously.

A style binding is an easy way to set a *single* style value.

To set *many* inline styles at the same time, the `NgStyle` directive may be the better choice.

Try binding `ngStyle` to a key:value control object. Each key of the object is a style name; its value is whatever is appropriate for that style.

Consider a `setCurrentStyles` component method that sets a component property, `currentStyles` with an object that defines three styles, based on the state of three other component properties:

Adding an `ngStyle` property binding to `currentStyles` sets the element's styles accordingly:

It's up to you to call `setCurrentStyles()`, both initially and when the dependent properties change.

{@a ngModel}

NgModel - Two-way binding to form elements with `[(ngModel)]`

When developing data entry forms, you often both display a data property and update that property when the user makes changes.

Two-way data binding with the `NgModel` directive makes that easy. Here's an example:

FormsModule is required to use *ngModel*

Before using the `ngModel` directive in a two-way data binding, you must import the `FormsModule` and add it to the NgModule's `imports` list. Learn more about the `FormsModule` and `ngModel` in the Forms guide.

Here's how to import the `FormsModule` to make `[(ngModel)]` available.

Inside `[(ngModel)]`

Looking back at the `name` binding, note that you could have achieved the same result with separate bindings to the `<input>` element's `value` property and `input` event.

That's cumbersome. Who can remember which element property to set and which element event emits user changes? How do you extract the currently displayed text from the input box so you can update the data property? Who wants to look that up each time?

That `ngModel` directive hides these onerous details behind its own `ngModel` input and `ngModelChange` output properties.

The `ngModel` data property sets the element's value property and the `ngModelChange` event property listens for changes to the element's value. The details are specific to each kind of element and therefore the `NgModel` directive only works for an element supported by a [ControlValueAccessor](api/forms/ControlValueAccessor) that adapts an element to this protocol. The ` ` box is one of those elements. Angular provides *value accessors* for all of the basic HTML form elements and the [_Forms_](guide/forms) guide shows how to bind to them. You can't apply `[(ngModel)]` to a non-form native element or a third-party custom component until you write a suitable *value accessor*, a technique that is beyond the scope of this guide. You don't need a _value accessor_ for an Angular component that you write because you can name the value and event properties to suit Angular's basic [two-way binding syntax](guide/template-syntax#two-way) and skip `NgModel` altogether. The [`sizer` shown above](guide/template-syntax#two-way) is an example of this technique.

Separate `ngModel` bindings is an improvement over binding to the element's native properties. You can do better.

You shouldn't have to mention the data property twice. Angular should be able to capture the component's data property and set it with a single declaration, which it can with the `[(ngModel)]` syntax:

Is `[(ngModel)]` all you need? Is there ever a reason to fall back to its expanded form?

The `[(ngModel)]` syntax can only *set* a data-bound property. If you need to do something more or something different, you can write the expanded form.

The following contrived example forces the input value to uppercase:

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

{@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.

The deep details of structural directives are covered in the *Structural Directives* guide where you'll learn:

- why you *prefix the directive name with an asterisk (*)*.
- to use `<ng-container>` to group elements when there is no suitable host element for the directive.
- how to write your own structural directive.
- that you can only apply one structural directive to an element.

This section is an introduction to the common structural directives:

- `NgIf` - conditionally add or remove an element from the DOM
- `NgSwitch` - a set of directives that switch among alternative views
- NgForOf - repeat a template for each item in a list

{@a ngIf}

NgIf

You can add or remove an element from the DOM by applying an `NgIf` directive to that element (called the *host element*). Bind the directive to a condition expression like `isActive` in this example.

Don't forget the asterisk (`*`) in front of `ngIf`.

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

Show/hide is not the same thing

You can control the visibility of an element with a class or style binding:

Hiding an element is quite different from removing an element with `NgIf`.

When you hide an element, that element and all of its descendents remain in the DOM. All components for those elements stay in memory and Angular may continue to check for changes. You could be holding onto considerable computing resources and degrading performance, for something the user can't see.

When `NgIf` is `false`, Angular removes the element and its descendents from the DOM. It destroys their components, potentially freeing up substantial resources, resulting in a more responsive user experience.

The show/hide technique is fine for a few elements with few children. You should be wary when hiding large component trees; `NgIf` may be the safer choice.

Guard against null

The `ngIf` directive is often used to guard against null. Show/hide is useless as a guard. Angular will throw an error if a nested expression tries to access a property of `null`.

Here we see `NgIf` guarding two `<div>`s. The `currentHero` name will appear only when there is a `currentHero`. The `nullHero` will never be displayed.

See also the [_safe navigation operator_](guide/template-syntax#safe-navigation-operator "Safe navigation operator (?.)") described below.

{@a ngFor}

NgForOf

`NgForOf` is a *repeater* directive — a way to present a list of items. You define a block of HTML that defines how a single item should be displayed. You tell Angular to use that block as a template for rendering each item in the list.

Here is an example of `NgForOf` applied to a simple `<div>`:

You can also apply an `NgForOf` to a component element, as in this example:

Don't forget the asterisk (`*`) in front of `ngFor`.

The text assigned to `*ngFor` is the instruction that guides the repeater process.

{@a microsyntax}

*ngFor microsyntax

The string assigned to `*ngFor` is not a template expression. It's a *microsyntax* — a little language of its own that Angular interprets. The string `"let hero of heroes"` means:

> *Take each hero in the `heroes` array, store it in the local `hero` looping variable, and make it available to the templated HTML for each iteration.*

Angular translates this instruction into a `<ng-template>` around the host element, then uses this template repeatedly to create a new set of elements and bindings for each `hero` in the list.

Learn about the *microsyntax* in the *Structural Directives* guide.

{@a template-input-variable}

{@a template-input-variables}

Template input variables

The `let` keyword before `hero` creates a *template input variable* called `hero`. The `NgForOf` directive iterates over the `heroes` array returned by the parent component's `heroes` property and sets `hero` to the current item from the array during each iteration.

You reference the `hero` input variable within the `NgForOf` host element (and within its descendants) to access the hero's properties. Here it is referenced first in an interpolation and then passed in a binding to the `hero` property of the `<hero-detail>` component.

Learn more about *template input variables* in the *Structural Directives* guide.

*ngFor with *index*

The `index` property of the `NgForOf` directive context returns the zero-based index of the item in each iteration. You can capture the `index` in a template input variable and use it in the template.

The next example captures the `index` in a variable named `i` and displays it with the hero name like this.

`NgFor` is implemented by the `NgForOf` directive. Read more about the other `NgForOf` context values such as `last`, `even`, and `odd` in the [NgForOf API reference](api/common/NgForOf).

{@a trackBy}

*ngFor with *trackBy*

The `NgForOf` directive may perform poorly, especially with large lists. A small change to one item, an item removed, or an item added can trigger a cascade of DOM manipulations.

For example, re-querying the server could reset the list with all new hero objects.

Most, if not all, are previously displayed heroes. *You* know this because the `id` of each hero hasn't changed. But Angular sees only a fresh list of new object references. It has no choice but to tear down the old DOM elements and insert all new DOM elements.

Angular can avoid this churn with `trackBy`. Add a method to the component that returns the value `NgForOf` *should* track. In this case, that value is the hero's `id`.

In the microsyntax expression, set `trackBy` to this method.

Here is an illustration of the *trackBy* effect. "Reset heroes" creates new heroes with the same `hero.id`s. "Change ids" creates new heroes with new `hero.id`s.

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

{@a ngSwitch}

The *NgSwitch* directives

NgSwitch is like the JavaScript `switch` statement. It can display *one* element from among several possible elements, based on a *switch condition*. Angular puts only the *selected* element into the DOM.

NgSwitch is actually a set of three, cooperating directives: `NgSwitch`, `NgSwitchCase`, and `NgSwitchDefault` as seen in this example.

`NgSwitch` is the controller directive. Bind it to an expression that returns the *switch value*. The `emotion` value in this example is a string, but the switch value can be of any type.

Bind to `[ngSwitch]`. You'll get an error if you try to set `*ngSwitch` because `NgSwitch` is an *attribute* directive, not a *structural* directive. It changes the behavior of its companion directives. It doesn't touch the DOM directly.

Bind to `*ngSwitchCase` and `*ngSwitchDefault`. The `NgSwitchCase` and `NgSwitchDefault` directives are *structural* directives because they add or remove elements from the DOM.

- `NgSwitchCase` adds its element to the DOM when its bound value equals the switch value.
- `NgSwitchDefault` adds its element to the DOM when there is no selected `NgSwitchCase`.

The switch directives are particularly useful for adding and removing *component elements*. This example switches among four "emotional hero" components defined in the `hero-switch.components.ts` file. Each component has a `hero` input property which is bound to the `currentHero` of the parent component.

Switch directives work as well with native elements and web components too. For example, you could replace the `<confused-hero>` switch case with the following.

{@a template-reference-variable}

{@a ref-vars}

{@a ref-var}

Template reference variables (`#var`)

A **template reference variable** is often a reference to a DOM element within a template. It can also be a reference to an Angular component or directive or a web component.

Use the hash symbol (#) to declare a reference variable. The `#phone` declares a `phone` variable on an `<input>` element.

You can refer to a template reference variable *anywhere* in the template. The `phone` variable declared on this `<input>` is consumed in a `<button>` on the other side of the template

How a reference variable gets its value

In most cases, Angular sets the reference variable's value to the element on which it was declared. In the previous example, `phone` refers to the *phone number* `<input>` box. The phone button click handler passes the *input* value to the component's `callPhone` method. But a directive can change that behavior and set the value to something else, such as itself. The `NgForm` directive does that.

The following is a *simplified* version of the form example in the Forms guide.

A template reference variable, `heroForm`, appears three times in this example, separated by a large amount of HTML. What is the value of `heroForm`?

If Angular hadn't taken it over when you imported the `FormsModule`, it would be the HTMLFormElement.

The `heroForm` is actually a reference to an Angular NgForm directive with the ability to track the value and validity of every control in the form.

The native `<form>` element doesn't have a `form` property. But the `NgForm` directive does, which explains how you can disable the submit button if the `heroForm.form.valid` is invalid and pass the entire form control tree to the parent component's `onSubmit` method.

Template reference variable warning notes

A template *reference* variable (`#phone`) is *not* the same as a template *input* variable (`let phone`) such as you might see in an `*ngFor` . Learn the difference in the *Structural Directives* guide.

The scope of a reference variable is the *entire template*. Do not define the same variable name more than once in the same template. The runtime value will be unpredictable.

You can use the `ref-` prefix alternative to `#` . This example declares the `fax` variable as `ref-fax` instead of `#fax` .

{@a inputs-outputs}

Input and Output properties

An *Input* property is a *settable* property annotated with an `@Input` decorator. Values flow *into* the property when it is data bound with a property binding

An *Output* property is an *observable* property annotated with an `@Output` decorator. The property almost always returns an Angular `EventEmitter` . Values flow *out* of the component as events bound with an event binding.

You can only bind to *another* component or directive through its *Input* and *Output* properties.

Remember that all **components** are **directives**. The following discussion refers to _components_ for brevity and because this topic is mostly a concern for component authors.

Discussion

You are usually binding a template to its *own component class*. In such binding expressions, the component's property or method is to the *right* of the (`=`).

The `iconUrl` and `onSave` are members of the `AppComponent` class. They are *not* decorated with

`@Input()` or `@Output`. Angular does not object.

You can always bind to a public property of a component in its own template. It doesn't have to be an *Input* or *Output* property

A component's class and template are closely coupled. They are both parts of the same thing. Together they *are* the component. Exchanges between a component class and its template are internal implementation details.

Binding to a different component

You can also bind to a property of a *different* component. In such bindings, the *other* component's property is to the *left* of the (`=`).

In the following example, the `AppComponent` template binds `AppComponent` class members to properties of the `HeroDetailComponent` whose selector is `'app-hero-detail'`.

The Angular compiler *may* reject these bindings with errors like this one:

Uncaught Error: Template parse errors: Can't bind to 'hero' since it isn't a known property of 'app-hero-detail'

You know that `HeroDetailComponent` has `hero` and `deleteRequest` properties. But the Angular compiler refuses to recognize them.

The Angular compiler won't bind to properties of a different component unless they are Input or Output properties.

There's a good reason for this rule.

It's OK for a component to bind to its *own* properties. The component author is in complete control of those bindings.

But other components shouldn't have that kind of unrestricted access. You'd have a hard time supporting your component if anyone could bind to any of its properties. Outside components should only be able to bind to the component's public binding API.

Angular asks you to be *explicit* about that API. It's up to *you* to decide which properties are available for binding by external components.

TypeScript *public* doesn't matter

You can't use the TypeScript *public* and *private* access modifiers to shape the component's public binding API.

All data bound properties must be TypeScript _public_ properties. Angular never binds to a TypeScript _private_ property.

Angular requires some other way to identify properties that *outside* components are allowed to bind to. That *other way* is the `@Input()` and `@Output()` decorators.

Declaring Input and Output properties

In the sample for this guide, the bindings to `HeroDetailComponent` do not fail because the data bound properties are annotated with `@Input()` and `@Output()` decorators.

Alternatively, you can identify members in the `inputs` and `outputs` arrays of the directive metadata, as in this example:

Input or output?

Input properties usually receive data values. *Output* properties expose event producers, such as `EventEmitter` objects.

The terms *input* and *output* reflect the perspective of the target directive.

`HeroDetailComponent.hero` is an **input** property from the perspective of `HeroDetailComponent` because data flows *into* that property from a template binding expression.

`HeroDetailComponent.deleteRequest` is an **output** property from the perspective of `HeroDetailComponent` because events stream *out* of that property and toward the handler in a template binding statement.

Aliasing input/output properties

Sometimes the public name of an input/output property should be different from the internal name.

This is frequently the case with attribute directives. Directive consumers expect to bind to the name of the directive. For example, when you apply a directive with a `myClick` selector to a `<div>` tag, you expect to bind to an event property that is also called `myClick`.

However, the directive name is often a poor choice for the name of a property within the directive class. The directive name rarely describes what the property does. The `myClick` directive name is not a good name for a property that emits click messages.

Fortunately, you can have a public name for the property that meets conventional expectations, while using a different name internally. In the example immediately above, you are actually binding *through the* `myClick` *alias* to the directive's own `clicks` property.

You can specify the alias for the property name by passing it into the input/output decorator like this:

You can also alias property names in the `inputs` and `outputs` arrays. You write a colon-delimited (`:`) string with the directive property name on the *left* and the public alias on the *right*:

{@a expression-operators}

Template expression operators

The template expression language employs a subset of JavaScript syntax supplemented with a few special operators for specific scenarios. The next sections cover two of these operators: *pipe* and *safe navigation operator*.

{@a pipe}

The pipe operator (|)

The result of an expression might require some transformation before you're ready to use it in a binding. For example, you might display a number as a currency, force text to uppercase, or filter a list and sort it.

Angular pipes are a good choice for small transformations such as these. Pipes are simple functions that accept an input value and return a transformed value. They're easy to apply within template expressions, using the **pipe operator (`|`)**:

The pipe operator passes the result of an expression on the left to a pipe function on the right.

You can chain expressions through multiple pipes:

And you can also apply parameters to a pipe:

The `json` pipe is particularly helpful for debugging bindings:

The generated output would look something like this

{ "id": 0, "name": "Hercules", "emotion": "happy", "birthdate": "1970-02-25T08:00:00.000Z", "url": "http://www.imdb.com/title/tt0065832/", "rate": 325 }

{@a safe-navigation-operator}

The safe navigation operator (`?.`) and null property paths

The Angular **safe navigation operator (`?.`)** is a fluent and convenient way to guard against null and undefined values in property paths. Here it is, protecting against a view render failure if the `currentHero` is null.

What happens when the following data bound `title` property is null?

The view still renders but the displayed value is blank; you see only "The title is" with nothing after it. That is reasonable behavior. At least the app doesn't crash.

Suppose the template expression involves a property path, as in this next example that displays the `name` of a null hero.

The null hero's name is {{nullHero.name}}

JavaScript throws a null reference error, and so does Angular:

TypeError: Cannot read property 'name' of null in [null].

Worse, the *entire view disappears*.

This would be reasonable behavior if the `hero` property could never be null. If it must never be null and yet it is null, that's a programming error that should be caught and fixed. Throwing an exception is the right thing to do.

On the other hand, null values in the property path may be OK from time to time, especially when the data are null now and will arrive eventually.

While waiting for data, the view should render without complaint, and the null property path should display as blank just as the `title` property does.

Unfortunately, the app crashes when the `currentHero` is null.

You could code around that problem with *ngIf.

You could try to chain parts of the property path with `&&`, knowing that the expression bails out when it encounters the first null.

These approaches have merit but can be cumbersome, especially if the property path is long. Imagine guarding against a null somewhere in a long property path such as `a.b.c.d`.

The Angular safe navigation operator (`?.`) is a more fluent and convenient way to guard against nulls in property paths. The expression bails out when it hits the first null value. The display is blank, but the app keeps rolling without errors.

It works perfectly with long property paths such as `a?.b?.c?.d` .

{@a non-null-assertion-operator}

The non-null assertion operator (`!`)

As of Typescript 2.0, you can enforce strict null checking with the `--strictNullChecks` flag. TypeScript then ensures that no variable is *unintentionally* null or undefined.

In this mode, typed variables disallow null and undefined by default. The type checker throws an error if you leave a variable unassigned or try to assign null or undefined to a variable whose type disallows null and undefined.

The type checker also throws an error if it can't determine whether a variable will be null or undefined at runtime. You may know that can't happen but the type checker doesn't know. You tell the type checker that it can't happen by applying the post-fix *non-null assertion operator (!)*.

The *Angular* **non-null assertion operator (`!`)** serves the same purpose in an Angular template.

For example, after you use *ngIf to check that `hero` is defined, you can assert that `hero` properties are also defined.

When the Angular compiler turns your template into TypeScript code, it prevents TypeScript from reporting that `hero.name` might be null or undefined.

Unlike the *safe navigation operator*, the **non-null assertion operator** does not guard against null or undefined. Rather it tells the TypeScript type checker to suspend strict null checks for a specific property expression.

You'll need this template operator when you turn on strict null checks. It's optional otherwise.

back to top

Summary

You've completed this survey of template syntax. Now it's time to put that knowledge to work on your own components and directives.