

TEMPLATE SYNTAX

Developer Preview Only - some details may change

Learn how to write templates that display data and consume user events with the help of data binding.

Our Angular application manages what the user sees and does through the interaction of a Component class instance (the *component*) and its user-facing template.

Many of us are familiar with the component/template duality from our 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.

Let's find out what it takes to write a template for our view. We'll cover these basic elements of template syntax:

- [HTML](#)
- [Interpolation](#)
- [Template expressions](#)

- [Template statements](#)
- [Binding syntax](#)
- [Property binding](#)
- [Attribute, class, and style bindings](#)
- [Event binding](#)
- [Two-way data binding with NgModel](#)
- [Built-in directives](#)
- [* and <template>](#)
- [Local template variables](#)
- [Input and output properties](#)
- [Template expression operators](#)

[Run the live example](#)

HTML

HTML is the language of the Angular template. Our [QuickStart](#) application had a template that was pure HTML:

```
<h3>My First Angular Application</h3>
```



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 simply ignored.)

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

We can extend the HTML vocabulary of our templates with components and directives that appear as new elements and attributes. And we are about to learn how to get and set DOM values dynamically through data binding.

Let's turn to the first form of data binding — interpolation — to see how much richer template HTML can be.

Interpolation

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

```
<p>My current hero is {{currentHero.firstName}}</p>
```



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

```
<h3>
  {{title}}
  
</h3>
```



The material 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 this example, Angular evaluates the `title` and `heroImageUrl`

properties and "fills in the blanks", displaying first a bold application title and then a heroic image.

More generally, the material 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 within braces:

```
<!-- "The sum of 1 + 1 is 2" -->  
<p>The sum of 1 + 1 is {{1 + 1}}</p>
```



The expression can invoke methods of the host component, as we do here with `getVal()`:

```
<!-- "The sum of 1 + 1 is not 4" -->  
<p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}</p>
```



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

We appear to be inserting the result between element tags and assigning to attributes. It's convenient to think so, and we rarely suffer for this mistake. But it is not literally true. Interpolation is a special syntax that Angular converts into a [property binding](#), as we'll explain below.

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

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.

We put a template expression within the interpolation braces when we wrote `{{1 + 1}}`. We'll see template expressions again in the [property binding](#) section, appearing in quotes to the right of the `=` symbol as in `[property]="expression"`.

We write 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 (`=`, `+=`, `-=`)
- the `new` operator
- 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 `?`.

Expression context

Perhaps more surprising, template expressions 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`. They are restricted to referencing members of the expression context.

The *expression context* is typically the **component instance**, which is the source of binding values.

When we see *title* wrapped in double-curly braces, `{{title}}`, we know that `title` is a property of the data-bound component. When we see *isUnchanged* in `[disabled]="isUnchanged"`, we know we are referring to that component's `isUnchanged` property.

The component itself is usually the expression *context*, in which case the template expression usually references that component.

The expression context can include objects other than the component. A [local template variable](#) is one such alternative context object.

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. We 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 more often than we might think. Expressions should finish quickly or the user experience may drag, especially on slower devices.

Simplicity

Although it's possible to write quite complex template expressions, we really shouldn't.

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 a `Date` or `Array`), it returns the same object *reference* when called twice in a row.

Template statements

A template **statement** responds to an **event** raised by a binding target such as an element, component, or directive.

We'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*. It's how we update application state from user input. There would be no point to responding to an event otherwise.

Responding to events is the other side of Angular's "unidirectional data flow". We'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 is different than the template expression parser and specifically supports both basic assignment (`=`) and chaining expressions with semicolons (`;`) and commas (`,`).

However, certain JavaScript syntax is not allowed:

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

Statement context

As with expressions, 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`.

Statements are restricted to referencing members of the statement context. The statement context is typically the **component instance** to which we are binding an event.

The `onSave` in `(click)="onSave()"` is sure to be a method of the data-bound component instance.

The statement context may include an object other than the component. A [local template variable](#) is one such alternative context object. We'll frequently see the reserved `$event` symbol in event binding statements, representing the "message" or "payload" of the raised event.

Statement guidelines

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




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

Binding syntax: An overview

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

Angular provides many kinds of data binding, and we'll discuss each of them in this chapter. First we'll take a high-level view of Angular data binding and its syntax.

We can group all bindings into three categories by the direction in which data flows. Each category has its distinctive syntax:

Data direction	Syntax	Binding type
One-way from data source to view target	<pre>{{expression}} [target] = "expression" bind-target = "expression"</pre> 	Interpolation Property Attribute Class Style
One-way from view target to data source	<pre>(target) = "statement" on-target = "statement"</pre> 	Event
Two-way	<pre>[(target)] = "expression" bindon-target = "expression"</pre> 	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-`).

What is that target? Before we can answer that question, we must challenge ourselves to look at template HTML in a new way.

A new mental model

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

Well, *it is HTML Plus*. But it's also significantly different than the HTML we're used to. We really need a new mental model.

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

```
<div class="special">Mental Model</div>
<div><b>{{currentHero.fullName}}</b></div>

<button disabled>Save</button>
```



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

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

```
<div class="special">Mental Model</div>  
<hero-detail></hero-detail>
```



That's HTML Plus.

Now we start to learn about data binding. The first binding we meet might look like this:

```
<!-- Bind button disabled state to `isUnchanged` property -->  
<button [disabled]="isUnchanged">Save</button>
```



We'll get to that peculiar bracket notation in a moment. Looking beyond it, our intuition tells us that we're binding to the button's `disabled` attribute and setting it to the current value of the component's `isUnchanged` property.

Our intuition is wrong! Our everyday HTML mental model is misleading us. In fact, once we start data binding, we are no longer working with HTML *attributes*. We aren't setting attributes. We 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 we might think!

That last category can be especially confusing ... until we understand 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 we discover if we 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 we 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 we cannot enable a button by writing `<button disabled="false">Still Disabled</button>`.

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 is so important, we'll say it again.

Template binding works with *properties* and *events*, not *attributes*.

A WORLD WITHOUT ATTRIBUTES

In the world of Angular 2, the only role of attributes is to initialize element and directive state. When we data bind, we're dealing exclusively with element and directive properties and events. Attributes effectively disappear.

With this model firmly in mind, let's 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:

Binding type	Target	Examples
Property	Element property Component property Directive property	<pre> <hero-detail [hero]="currentHero"></hero- detail> <div [ngClass] = "{selected: isSelected}"></div></pre>

Event

Element event
Component event
Directive event

```
<button (click) =  
  "onSave()">Save</button>  
<hero-detail (deleted)="onHeroDeleted()">  
</hero-detail>  
<div myClick  
  (myClick)="clicked=$event">click me</div>
```



Two-way

Event and property

```
<input [(ngModel)]="heroName">
```



Attribute

Attribute (the exception)

```
<button [attr.aria-  
  label]="help">help</button>
```



Class

class property

```
<div  
  [class.special]="isSpecial">Special</div>
```



Style

style property

```
<button [style.color] = "isSpecial ?  
  'red' : 'green'">
```



Let's descend from the architectural clouds and look at each of these binding types in concrete detail.

Property binding

We write a template **property binding** when we want to set a property of a view element 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:

```
<img [src]="heroImageUrl">
```



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

```
<button [disabled]="isUnchanged">Cancel</button>
```



Another is setting a property of a directive:

```
<div [ngClass]="['special']">NgClass is special</div>
```



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

```
<hero-detail [hero]="selectedHero"></hero-detail>
```



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.

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

Nor can we use property binding to *call* a method on the target element.

If the element raises events we can listen to them with an [event binding](#).

If we must read a target element property or call one of its methods, we'll need a different technique. See the API reference for [viewChild](#) and [contentChild](#).

Binding target

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

```
<img [src]="heroImageUrl">
```



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. We 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:

```
<div [ngClass]="['special']">NgClass is special</div>
```



Technically, Angular is matching the name to a directive `input`, 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.

Template expressions in property binding

As we've already discussed, evaluation of a template expression should have no visible side effects. The expression language itself does its part to keep us safe. We can't assign a value to anything in a property binding expression nor use the increment and decorator operators.

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

The expression could call something like `getFoo()`. Only we know what `getFoo()` does. If `getFoo()` changes something and we happen to be binding to that something, we risk an unpleasant experience. Angular may or may not display the changed value. Angular may detect the change and throw a warning error. Our general advice: stick to data properties and to methods that return values and do no more.

The template expression should evaluate to a value of the type expected by the target property. Most native element properties expect a string. For example, the image `src` should be set to a string that's an URL for the resource providing the image. On the other hand, the `disabled` property of a button expects a Boolean value, so an expression that's assigned to `disabled` should evaluate to `true` or `false`.

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

property binding:

```
<hero-detail [hero]="selectedHero"></hero-detail>
```



This is good news. If `hero` were an attribute, we could not set it to a `Hero` object.

```
<!-- Doesn't work! HeroDetailComponent expects a Hero, not a string -->  
<hero-detail hero="...what do we do here??? ..."></hero-detail>
```



We can't set an attribute to an object. We can only set it to a string. Internally, the attribute may be able to convert that string to an object before setting the like-named element property. That's good to know but not helpful to us when we're trying to pass a significant data object from one component element to another. The power of property binding is its ability to bypass the attribute and set the element property directly with a value of the appropriate type.

Property binding or interpolation?

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

```
  
<img [src]=''' + heroImageUrl">  
  
<div>The title is {{title}}</div>  
<div [textContent]='The title is '+title"></div>
```



Interpolation is a convenient alternative for property binding in many cases. In fact, Angular translates those interpolations into the corresponding property bindings before rendering the view.

There is no technical reason to prefer one form to the other. We lean toward readability, which tends to favor interpolation. We suggest establishing coding style rules for the organization and choosing the form that both conforms to the rules and feels most natural for the task at hand.

Attribute, class, and style bindings

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

Attribute binding

We 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.

We have stressed throughout this chapter 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?

We 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.

We become painfully aware of this fact when we try to write something like this:

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



We 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.

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

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

Here we bind `[attr.colspan]` to a calculated value:

```
<table border=1>  
  <!-- expression calculates colspan=2 -->  
  <tr><td [attr.colspan]="1 + 1">One-Two</td></tr>
```



```
<!-- ERROR: There is no `colspan` property to set!
  <tr><td colspan="{{1 + 1}}">Three-Four</td></tr>
-->

<tr><td>Five</td><td>Six</td></tr>
</table>
```

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:

```
<!-- create and set an aria attribute for assistive technology -->
<button [attr.aria-label]="actionName">{{actionName}} with Aria</button>
```



Class binding

We 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, we 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 we set the attribute without binding:

```
<!-- standard class attribute setting -->  
<div class="bad curly special">Bad curly special</div>
```



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

```
<!-- reset/override all class names with a binding -->  
<div class="bad curly special"  
  [class]="badCurly">Bad curly</div>
```



Finally, we can bind to a specific class name. Angular adds the class when the template expression evaluates to something truthy. It removes the class when the expression is falsey.

```
<!-- toggle the "special" class on/off with a property -->  
<div [class.special]="isSpecial">The class binding is special</div>  
  
<!-- binding to `class.special` trumps the class attribute -->  
<div class="special"  
  [class.special]="!isSpecial">This one is not so special</div>
```



While this is a fine way to toggle a single class name, we generally prefer the [NgClass directive](#) for managing multiple class names at

the same time.

Style binding

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

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

```
<button [style.color] = "isSpecial ? 'red' : 'green'">Red</button>  
<button [style.backgroundColor]="canSave ? 'cyan' : 'grey'" >Save</button>
```



Some style binding styles have unit extension. Here we conditionally set the font size in “em” and “%” units .

```
<button [style.fontSize.em]="isSpecial ? 3 : 1" >Big</button>  
<button [style.fontSize.%]="!isSpecial ? 150 : 50" >Small</button>
```



While this is a fine way to set a single style, we generally prefer the [NgStyle directive](#) when setting several inline styles at the same time.

Event binding

The bindings we've met so far flow data in one direction: *from the 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 the component*.

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

Event binding syntax consists of a **target event** 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 event, calling the component's `onSave()` method whenever a click occurs:

```
<button (click)="onSave()">Save</button>
```



Target event

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

```
<button (click)="onSave()">Save</button>
```



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

```
<button on-click="onSave()">On Save</button>
```



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:

```
<!-- `myClick` is an event on the custom `MyClickDirective` -->  
  
<div myClick (myClick)="clickMessage=$event">click with myClick</div>
```



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 that wants to do something in response to the event, such as take a value from the HTML control and store it in 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 itself. If the target event is a native DOM element event, the `$event` is a [DOM event object](#), with properties such as `target` and `target.value`.

Consider this example:

```
<input [value]="currentHero.firstName"  
      (input)="currentHero.firstName=$event.target.value" >
```



We're binding the input box `value` to a `firstName` property, and we're listening for changes by binding 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 `firstName` property, we must get the changed text by following the path `$event.target.value`.

If the event belongs to a directive (remember: components are directives), `$event` has whatever shape the directive chose to produce.

Custom events with EventEmitter

Directives typically raise custom events with an Angular [EventEmitter](#). A 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 that 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 produces `deleted` events with an `EventEmitter`.

HeroDetailComponent.ts (excerpt)

```
deleted = new EventEmitter<Hero>();  
onDelete() {  
  this.deleted.emit(this.hero);
```



```
}
```

When the user clicks a button, the component invokes the `onDelete()` method, which emits a `Hero` object. The `HeroDetailComponent` doesn't know how to delete a hero. Its job is to present information and respond to user actions.

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

```
<hero-detail (deleted)="onHeroDeleted($event)" [hero]="currentHero">
</hero-detail>
```



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

The `onHeroDeleted` method has a side effect: It deletes a hero. Side effects are not just OK, they are 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. It's all good.

Two-way binding with NgModel

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

The `NgModel` directive serves that purpose, as in this example:

```
<input [(ngModel)]="currentHero.firstName">
```



`[()]` = BANANA IN A BOX

To remember that the parentheses go inside the brackets, visualize a *banana in a box*.

Alternatively, we can use the canonical prefix form:

```
<input bindon-ngModel="currentHero.firstName">
```



There's a story behind this construction, a story that builds on the property and event binding techniques we learned previously.

We could have achieved the same result as `NgModel` with separate bindings to the `<input>` element's `value` property and `input` event.

```
<input [value]="currentHero.firstName"
      (input)="currentHero.firstName=$event.target.value" >
```



That's cumbersome. Who can remember what element property to set and what event reports user changes? How do we

extract the currently displayed text from the input box so we can update the data property? Who wants to look that up each time?

That `ngModel` directive hides these onerous details. It wraps the element's `value` property, listens to the `input` event, and raises its own event with the changed text ready for us to consume.

```
<input  
  [ngModel]="currentHero.firstName"  
  (ngModelChange)="currentHero.firstName=$event">
```



That's an improvement, but it could be better.

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

```
<input [(ngModel)]="currentHero.firstName">
```



Internally, Angular maps the term `ngModel` to an `ngModel` input property and an `ngModelChange` output property. That's a specific example of a more general pattern in which Angular matches `[(x)]` to an `x` input property for property binding and an `xChange` output property for event binding.

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

Well, `NgModel` can only set the target property. What if we need to do something more or something different when the user changes the value? Then we need to use the expanded form.

Let's try something silly like forcing the input value to uppercase:

```
<input
  [ngModel]="currentHero.firstName"
  (ngModelChange)="setUpperCaseFirstName($event)">
```



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

NgModel Binding

Result: Hercules

Hercules	without NgModel
Hercules	[(ngModel)]
Hercules	bindon-ngModel
Hercules	(ngModelChange) = "...firstName=Sevent"
Hercules	(ngModelChange) = "setUpperCaseFirstName(Sevent)"

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.

We don't need many of those directives in Angular 2. Quite often we can achieve the same results with the more capable and expressive Angular 2 binding system. Why create a directive to handle a click when we can write a simple binding such as this?

```
<button (click)="onSave()">Save</button>
```



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

This segment reviews some of the most frequently used built-in directives.

NgClass

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

A [class binding](#) is a good way to add or remove a *single* class.

```
<!-- toggle the "special" class on/off with a property -->  
<div [class.special]="isSpecial">The class binding is special</div>
```



The `NgClass` directive may be the better choice when we want to add or remove *many* CSS classes at the same time.

A good way to apply `NgClass` is by binding it 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 component method such as `setClasses` that manages the state of three CSS classes:

```
setClasses() {  
  let classes = {  
    saveable: this.canSave,      // true  
    modified: !this.isUnchanged, // false  
    special: this.isSpecial,     // true  
  }  
  return classes;  
}
```



Now we can add an `NgClass` property binding that calls `setClasses` and sets the element's classes accordingly:

```
<div [ngClass]="setClasses()">This div is saveable and special</div>
```



NgStyle

We can set inline styles dynamically, based on the state of the component. Binding to `NgStyle` lets us set many inline styles

simultaneously.

A [style binding](#) is an easy way to set a *single* style value.

```
<div [style.fontSize]="isSpecial ? 'x-large' : 'smaller'" >  
  This div is x-large  
</div>
```



The `NgStyle` directive may be the better choice when we want to set *many* inline styles at the same time.

We apply `NgStyle` by binding it 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 component method such as `setStyles` that returns an object defining three styles:

```
setStyles() {  
  let styles = {  
    // CSS property names  
    'font-style': this.canSave      ? 'italic' : 'normal', // italic  
    'font-weight': !this.isUnchanged ? 'bold'   : 'normal', // normal  
    'font-size':   this.isSpecial   ? '24px'   : '8px',     // 24px  
  }  
  return styles;  
}
```



Now we just add an `NgStyle` property binding that calls `setStyles` and sets the element's styles accordingly:

```
<div [ngStyle]="setStyles()">  
  This div is italic, normal weight, and extra large (24px)  
</div>
```



NgIf

We can **add** an element subtree (an element and its children) to the DOM by binding an `NgIf` directive to a truthy expression.

```
<div *ngIf="currentHero">Hello, {{currentHero.firstName}}</div>
```



Don't forget the asterisk (`*`) in front of `ngIf` . For more information, see [* and <template>](#).

Binding to a falsey expression **removes** the element subtree from the DOM.

```
<!-- not displayed because nullHero is falsey.  
  `nullHero.firstName` never has a chance to fail -->  
<div *ngIf="nullHero">Hello, {{nullHero.firstName}}</div>
```



```
<!-- Hero Detail is not in the DOM because isActive is false-->  
<hero-detail *ngIf="isActive"></hero-detail>
```

Visibility and NgIf are not the same

We can show and hide an element subtree (the element and its children) with a [class](#) or [style](#) binding:

```
<!-- isSpecial is true -->  
<div [class.hidden]="!isSpecial">Show with class</div>  
<div [class.hidden]="isSpecial">Hide with class</div>  
  
<!-- HeroDetail is in the DOM but hidden -->  
<hero-detail [class.hidden]="isSpecial"></hero-detail>  
  
<div [style.display]="isSpecial ? 'block' : 'none'">Show with style</div>  
<div [style.display]="isSpecial ? 'none' : 'block'">Hide with style</div>
```



Hiding a subtree is quite different from excluding a subtree with `NgIf`.

When we hide the element subtree, it remains in the DOM. Components in the subtree are preserved, along with their state. Angular may continue to check for changes even to invisible properties. The subtree may tie up substantial memory and computing resources.

When `NgIf` is `false`, Angular **physically removes the element subtree from the DOM**. It **destroys** components in the subtree, along with their state, potentially freeing up substantial resources and resulting in better performance for the user.

The show/hide technique is probably fine for small element trees. We should be wary when hiding large trees; `NgIf` may be the safer choice. Always measure before leaping to conclusions.

NgSwitch

We bind to `NgSwitch` when we want to display *one* element tree (an element and its children) from a set of possible element trees, based on some condition. Angular puts only the *selected* element tree into the DOM.

Here's an example:

```
<span [ngSwitch]="toeChoice">
  <template [ngSwitchWhen]="'Eenie'">Eenie</template>
  <template [ngSwitchWhen]="'Meanie'">Meanie</template>
  <template [ngSwitchWhen]="'Miney'">Miney</template>
  <template [ngSwitchWhen]="'Moe'">Moe</template>
  <template ngSwitchDefault>Other</template>
</span>
```



We bind the parent `NgSwitch` directive to an expression returning a *switch value*. The value is a string in this example, but it can be a value of any type.

The parent `NgSwitch` directive controls a set of child `<template>` elements. Each `<template>` wraps a candidate subtree. A template is either pegged to a “match value” expression or marked as the default template.

At any particular moment, at most one of these templates is in the DOM.

If the template's *match value* equals the switch value, Angular adds the template's subtree to the DOM. If no template is a match and there is a default template, Angular adds the default template's subtree to the DOM. Angular removes and destroys the subtrees of all other templates.

Three collaborating directives are at work here:

1. `ngSwitch`: bound to an expression that returns the switch value
2. `ngSwitchWhen`: bound to an expression returning a match value
3. `ngSwitchDefault`: a marker attribute on the default template

NgFor

`NgFor` is a *repeater* directive — a way to customize data display.

Our goal is to present a list of items. We define a block of HTML that defines how a single item should be displayed. We tell Angular to use that block as a template for rendering each item in the list.

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

```
<div *ngFor="#hero of heroes">{{hero.fullName}}</div>
```



We can also apply an `NgFor` to a component element, as in this example:

```
<hero-detail *ngFor="#hero of heroes" [hero]="hero"></hero-detail>
```



Don't forget the asterisk (`*`) in front of `ngFor` . For more information, see [* and <template>](#).

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

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. In this example, the string `"#hero of heroes"` means:

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

Angular translates this instruction into a new set of elements and bindings. We'll talk about this in the next section.

In the two previous examples, the `ngFor` directive iterates over the `heroes` array returned by the parent component's `heroes` property, stamping out instances of the element to which it is applied. Angular creates a fresh instance of the template for each hero in the array.

The hash (`#`) character before `"hero"` creates a [local template variable](#) called `hero` .

We use this variable within the template to access a hero's properties, as we're doing in the interpolation. We can also pass

the variable in a binding to a component element, as we're doing with `hero-detail`.

NgFor with index

The `ngFor` directive supports an optional `index` that increases from 0 to the length of the array for each iteration. We can capture the index in a local template variable and use it in our template.

The next example captures the index in a variable named `i`, using it to stamp out rows like "1 - Hercules Son of Zeus".

```
<div *ngFor="#hero of heroes, #i=index">{{i + 1}} - {{hero.fullName}}</div>
```



Learn about other special values such as `last`, `even`, and `odd` in the [NgFor API reference](#).

* and <template>

When we reviewed the `NgFor` and `NgIf` built-in directives, we called out an oddity of the syntax: the asterisk (`*`) that appears before the directive name.

The `*` is a bit of **syntactic sugar** that makes it easier to read and write directives that modify HTML layout with the help of templates. `NgFor`, `NgIf`, and `NgSwitch` all add and remove element subtrees that are wrapped in `<template>` tags.

With the [NgSwitch](#) directive we always write the `<template>` tags explicitly. There isn't much choice; we define a different

template for each switch choice and let the directive render the template that matches the switch value.

[NgFor](#) and [NgIf](#), on the other hand, each need only one template: the *template-to-repeat* and the *template-to-include*, respectively.

The `*` prefix syntax is a convenient way to skip the `<template>` wrapper tags and focus directly on the HTML element to repeat or include. Angular sees the `*` and expands the HTML into the `<template>` tags for us.

Expanding `*ngIf`

We can do that expansion ourselves if we wish. Here's some code with `*ngIf`:

```
<hero-detail *ngIf="currentHero" [hero]="currentHero"></hero-detail>
```



Here's the equivalent with `<template>` and `ngIf`:

```
<template [ngIf]="currentHero">
  <hero-detail [hero]="currentHero"></hero-detail>
</template>
```



Notice that the `*` is gone and we have a [property binding](#) to the `ngIf` directive, applied in this case to the `<template>` rather than the application's `hero-detail` component.

The `[hero]="currentHero"` binding remains on the child `<hero-detail>` element inside the template.

REMEMBER THE BRACKETS!

Don't make the mistake of writing `ngIf="currentHero"` ! That syntax assigns the *string* value "currentHero" to `ngIf` . In JavaScript a non-empty string is a truthy value, so `ngIf` would always be `true` and Angular would always display the `hero-detail` ... even when there is no `currentHero` !

Expanding `*ngFor`

A similar transformation applies to `*ngFor` . We can de-sugar the syntax ourselves. First, here's an example with `*ngFor` :

```
<hero-detail *ngFor="#hero of heroes" [hero]="hero"></hero-detail>
```



Here's the same example, slightly expanded:

```
<hero-detail template="ngFor #hero of heroes" [hero]="hero"></hero-detail>
```



And here it is, expanded further:

```
<template ngFor #hero [ngForOf]="heroes">
  <hero-detail [hero]="hero"></hero-detail>
</template>
```



The `NgFor` code is a bit more complex than `NgIf` because a repeater has more moving parts to configure. In this case, we

have to remember the `NgForOf` directive that identifies the list. Using the `*ngFor` syntax is much easier than writing out this expanded HTML ourselves.

Local template variables

A **local template variable** is a vehicle for moving data across element lines.

We've seen `#hero` used to declare a local template variable several times in this chapter, most prominently when writing [NgFor](#) repeaters.

In [* and <templates>](#), we learned how Angular expands an `*ngFor` on a component tag into a `<template>` that wraps the component.

```
<template ngFor #hero [ngForOf]="heroes">
  <hero-detail [hero]="hero"></hero-detail>
</template>
```



The hash (`#`) prefix to "hero" means that we're defining a `hero` variable.

Folks who don't like using the `#` character can use its canonical alternative, the `var-` prefix. For example, we can declare the our `hero` variable using either `#hero` or `var-hero`.

We define `hero` on the outer `<template>` element, where it becomes the current hero item as Angular iterates through the list of heroes.

The `hero` variable appears again in the binding on the inner `<hero-detail>` component element. That's how each instance of the `<hero-detail>` gets its hero.

Referencing a local template variable

We can reference a local template variable on the same element, on a sibling element, or on any child elements.

Here are two other examples of creating and consuming a local template variable:

```
<!-- phone refers to the input element; pass its `value` to an event handler -->
<input #phone placeholder="phone number">
<button (click)="callPhone(phone.value)">Call</button>

<!-- fax refers to the input element; pass its `value` to an event handler -->
<input var-fax placeholder="fax number">
<button (click)="callFax(fax.value)">Fax</button>
```



How a variable gets its value

The value assigned to a variable depends upon the context.

When a directive is present on the element, as it is in the earlier `NgFor` `<hero-detail>` component example, the directive sets the value. Accordingly, the `NgFor` directive sets the `hero` variable to a hero item from the `heroes` array.

When no directive is present, as in phone and fax examples, Angular sets the variable's value to the element on which it was defined. We defined these variables on the `input` elements. We're passing those `input` element objects across to the button elements, where they're used in arguments to the `call` methods in the event bindings.

NgForm and local template variables

Let's look at one final example: a form, the poster child for local template variables.

The HTML for a form can be quite involved, as we saw in the [Forms](#) chapter. The following is a *simplified* example — and it's not simple at all.

```
<form (ngSubmit)="onSubmit(theForm)" #theForm="ngForm">
  <div class="form-group">
    <label for="name">Name</label>
    <input class="form-control" required ngControl="firstName"
      [(ngModel)]="currentHero.firstName">
  </div>
  <button type="submit" [disabled]="!theForm.form.valid">Submit</button>
</form>
```



A local template variable, `theForm`, appears three times in this example, separated by a large amount of HTML.

```
<form (ngSubmit)="onSubmit(theForm)" #theForm="ngForm">
  <!-- . . . -->
  <button type="submit" [disabled]="!theForm.form.valid">Submit</button>
</form>
```



What is the value of `theForm`?

It would be the [HTMLFormElement](#) if Angular hadn't taken it over. It's actually `ngForm`, a reference to the Angular built-in `NgForm` directive that wraps the native `HTMLFormElement` and endows it with additional superpowers such as the ability to track the validity of user input.

This explains how we can disable the submit button by checking `theForm.form.valid` and pass an object with rich information to the parent component's `onSubmit` method.

Input and output properties

So far, we've focused mainly on binding to component members within template expressions and statements that appear on the *right side of the binding declaration*. A member in that position is a data binding **source**.

This section concentrates on binding to **targets**, which are directive properties on the *left side of the binding declaration*. These directive properties must be declared as **inputs** or **outputs**.

Remember: All **components** are **directives**.

We're drawing a sharp distinction between a data binding **target** and a data binding **source**.

The *target* of a binding is to the *left* of the `=`. The *source* is on the *right* of the `=`.

The *target* of a binding is the property or event inside the binding punctuation: `[]`, `()` or `[]()`. The *source* is either inside quotes (`"`) or within an interpolation (`{}`).

Every member of a **source** directive is automatically available for binding. We don't have to do anything special to access a directive member in a template expression or statement.

We have *limited* access to members of a **target** directive. We can only bind to properties that are explicitly identified as *inputs* and *outputs*.

In the following example, `iconUrl` and `onSave` are members of a component that are referenced within quoted syntax to the right of the `=`.

```
<img [src]="iconUrl"/>
<button (click)="onSave()">Save</button>
```



They are *neither inputs nor outputs* of the component. They are data sources for their bindings.

Now look at `HeroDetailComponent` when it is the **target of a binding**.

```
<hero-detail [hero]="currentHero" (deleted)="onHeroDeleted($event)">
</hero-detail>
```



Both `HeroDetailComponent.hero` and `HeroDetailComponent.deleted` are on the **left side** of binding declarations.

`HeroDetailComponent.hero` is inside brackets; it is the target of a property binding. `HeroDetailComponent.deleted` is

inside parentheses; it is the target of an event binding.

Declaring input and output properties

Target properties must be explicitly marked as inputs or outputs.

When we peek inside `HeroDetailComponent`, we see that these properties are marked with decorators as input and output properties.

```
@Input() hero: Hero;  
@Output() deleted = new EventEmitter<Hero>();
```



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

```
@Component({  
  inputs: ['hero'],  
  outputs: ['deleted'],  
})
```

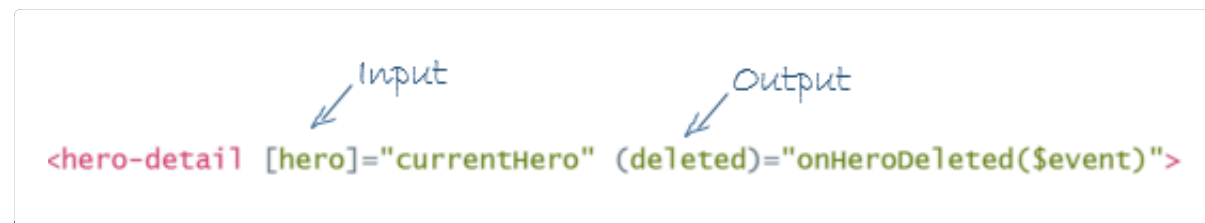


We can specify an input/output property either with a decorator or in a metadata array. **Don't do both!**

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.deleted` 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 we want the public name of an input/output property to 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 we apply a directive with a `myClick` selector to a `<div>` tag, we expect to bind to an event property that is also called `myClick`.

```
<div myClick (myClick)="clickMessage=$event">click with myClick</div>
```



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

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

```
@Output('myClick') clicks = new EventEmitter<string>(); // @Output(alias) propertyName = ...
```



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

```
@Directive({  
  outputs: ['clicks:myClick'] // propertyName:alias  
})
```



Template expression operators

The template expression language employs a subset of JavaScript syntax supplemented with a few special operators for specific scenarios. We'll cover two of these operators: *pipe* and *Elvis*.

The pipe operator (|)

The result of an expression might require some transformation before we're ready to use it in a binding. For example, we might want to 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 (|)**:

```
<!-- Force title to uppercase -->  
<div>{{ title | uppercase }}</div>
```



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

We can chain expressions through multiple pipes:

```
<!-- Pipe chaining: force title to uppercase, then to lowercase -->  
<div>{{ title | uppercase | lowercase }}</div>
```



And we can configure them too:

```
<!-- pipe with configuration argument => "February 25, 1970" -->  
<div>Birthdate: {{currentHero?.birthdate | date:'longDate'}}</div>
```



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

```
<div>{{currentHero | json}}</div>
```



```
<!-- Output:
  { "firstName": "Hercules", "lastName": "Son of Zeus",
    "birthdate": "1970-02-25T08:00:00.000Z",
    "url": "http://www.imdb.com/title/tt0065832/",
    "rate": 325, "id": 1 }
-->
```

The Elvis operator (?.) and null property paths

The Angular **Elvis 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.

```
The current hero's name is {{currentHero?.firstName}}
```



Let's elaborate on the problem and this particular solution.

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

```
The title is {{ title }}
```



The view still renders but the displayed value is blank; we 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 where we're displaying the `firstName` of a null hero.

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



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

```
TypeError: Cannot read property 'firstName' of null in [null]
```



Worse, the *entire view disappears*.

We could claim that this is reasonable behavior if we believed that the `hero` property must never be null. If it must never be null and yet it is null, **we've made 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 we know the data will arrive eventually.

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

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

We could code around that problem with [NgIf](#).

```
<!--No hero, div not displayed, no error -->  
<div *ngIf="nullHero">The null hero's name is {{nullHero.firstName}}</div>
```



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

```
The null hero's name is {{nullHero && nullHero.firstName}}
```



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 Elvis 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.**

```
<!-- No hero, no problem! -->  
The null hero's name is {{nullHero?.firstName}}
```



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

Summary

We've completed our survey of template syntax. Now it's time to put that knowledge to work as we write our own components and directives.

Next Step

[Pipes](#)