Property binding `[property]`

Contents ›

Use property binding to *set* properties of target elements or directive `@Input()` decorators.

> See the live example / download example for a working example containing the code snippets in this guide.

One-way in

Property binding flows a value in one direction, from a component's property into a target element property.

You can't use property binding to read or pull values out of target elements. 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.

If you must read a target element property or call one of its methods, see the API reference for ViewChild and ContentChild.

Examples

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 `itemImageUrl` property:

src/app/app.component.html

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

Here's an example of binding to the `colSpan` property. Notice that it's not `colspan`, which is the attribute, spelled with a lowercase `s`.

src/app/app.component.html

```
<!-- Notice the colSpan property is camel case -->
```

```
<tr><td [colSpan]="2">Span 2 columns</td></tr>
```

For more details, see the MDN HTMLTableCellElement documentation.

For more information about `colSpan` and `colspan`, see the Attribute binding guide.

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

src/app/app.component.html

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

Another is setting a property of a directive:

src/app/app.component.html

```
<p [ngClass]="classes">[ngClass] binding to the classes property making this blue</p>
```

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

src/app/app.component.html

```
<app-item-detail [childItem]="parentItem"></app-item-detail>
```

Binding targets

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.

src/app/app.component.html

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

There's also the `bind-` prefix alternative:

src/app/app.component.html

```
<img bind-src="itemImageUrl">
```

In most cases, the target name is the name of a property, even when it appears to be the name of an attribute. So in this case, `src` is the name of the `<img>` 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:

src/app/app.component.html

```
<p [ngClass]="classes">[ngClass] binding to the classes property making this blue</p>
```

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.

> Though the target name is usually the name of a property, there is an automatic attribute-to-property mapping in Angular for several common attributes. These include `class/className`, `innerHtml/innerHTML`, and `tabindex/tabIndex`.

Avoid side effects

Evaluation of a template expression should have no visible side effects. The expression language itself, or the way you write template expressions, helps to a certain extent; you can't assign a value to anything in a property binding expression nor use the increment and decrement operators.

For example, you could have an expression that invoked a property or method that had side effects. The expression could call something like `getFoo()` where only you know what `getFoo()` does. If `getFoo()` changes something and you happen to be binding to that something, Angular may or may not display the changed value. Angular may detect the change and throw a warning error. As a best practice, stick to properties and to methods that return values and avoid side effects.

Return the proper type

The template expression should evaluate to the type of value that the target property expects. Return a string if the target property expects a string, a number if it expects a number, an object if it expects an object, and so on.

In the following example, the `childItem` property of the `ItemDetailComponent` expects a string, which is exactly what you're sending in the property binding:

src/app/app.component.html

```
<app-item-detail [childItem]="parentItem"></app-item-detail>
```

You can confirm this by looking in the `ItemDetailComponent` where the `@Input` type is set to a string:

src/app/item-detail/item-detail.component.ts (setting the @Input() type)

```
@Input() childItem: string;
```

As you can see here, the `parentItem` in `AppComponent` is a string, which the `ItemDetailComponent` expects:

src/app/app.component.ts

```
parentItem = 'lamp';
```

Passing in an object

The previous simple example showed passing in a string. To pass in an object, the syntax and thinking are the same.

In this scenario, `ItemListComponent` is nested within `AppComponent` and the `items` property expects an array of objects.

src/app/app.component.html

```
<app-item-list [items]="currentItems"></app-item-list>
```

The `items` property is declared in the `ItemListComponent` with a type of `Item` and decorated with `@Input()`:

src/app/item-list.component.ts

```
@Input() items: Item[];
```

In this sample app, an `Item` is an object that has two properties; an `id` and a `name`.

src/app/item.ts

```
export interface Item {
  id: number;
  name: string;
}
```

While a list of items exists in another file, `mock-items.ts`, you can specify a different item in `app.component.ts` so that the new item will render:

src/app.component.ts

```
currentItems = [{
  id: 21,
  name: 'phone'
}];
```

You just have to make sure, in this case, that you're supplying an array of objects because that's the type of `Item` and is what the nested component, `ItemListComponent`, expects.

In this example, `AppComponent` specifies a different `item` object (`currentItems`) and passes it to the nested `ItemListComponent`. `ItemListComponent` was able to use `currentItems` because it matches what an `Item` object is according to `item.ts`. The `item.ts` file is where `ItemListComponent` gets its definition of an `item`.

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:

src/app.component.html

```
<app-item-detail childItem="parentItem"></app-item-detail>
```

Omitting the brackets will render the string `parentItem`, not the value of `parentItem`.

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 put directly 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 `StringInitComponent` to a fixed string, not a template expression. Angular sets it and forgets about it.

src/app/app.component.html

```
<app-string-init prefix="This is a one-time initialized string."></app-string-init>
```

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

Property binding vs. interpolation

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

src/app/app.component.html

```
<p><img src="{{itemImageUrl}}"> is the <i>interpolated</i> image.</p>
<p><img [src]="itemImageUrl"> is the <i>property bound</i> image.</p>

<p><span>"{{interpolationTitle}}" is the <i>interpolated</i> title.</span></p>
<p>"<span [innerHTML]="propertyTitle"></span>" is the <i>property bound</i> title.</p>
```

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, though readability tends to favor interpolation. However, *when setting an element property to a non-string data value, you must use property binding*.

Content security

Imagine the following malicious content.

src/app/app.component.ts

```
evilTitle = 'Template <script>alert("evil never sleeps")</script> Syntax';
```

In the component template, the content might be used with interpolation:

src/app/app.component.html

```
<p><span>"{{evilTitle}}" is the <i>interpolated</i> evil title.</span></p>
```

Fortunately, Angular data binding is on alert for dangerous HTML. In the above case, the HTML displays as is, and the Javascript does not execute. Angular **does not** allow HTML with script tags to leak into the browser, neither with interpolation nor property binding.

In the following example, however, Angular sanitizes the values before displaying them.

src/app/app.component.html

```
<!--
 Angular generates a warning for the following line as it sanitizes them
 WARNING: sanitizing HTML stripped some content (see http://g.co/ng/security#xss).
-->
 <p>"<span [innerHTML]="evilTitle"></span>" is the <i>property bound</i> evil title.
</p>
```

Interpolation handles the `<script>` tags differently than property binding but both approaches render the content harmlessly. The following is the browser output of the `evilTitle` examples.

```
"Template  Syntax" is the interpolated evil title.
"Template alert("evil never sleeps")Syntax" is the property bound evil title.
```

Version 10.0.10-local+sha.b32126c335.