Interpolation and template expressions

Contents

Interpolation allows you to incorporate calculated strings into the text between HTML element tags and within attribute assignments. Template expressions are what you use to calculate those strings.

See the live example / download example for all of the syntax and code snippets in this guide.

Interpolation {{...}}

Interpolation refers to embedding expressions into marked up text. By default, interpolation uses as its delimiter the double curly braces, {{ and }}.

In the following snippet, `{{ currentCustomer }}` is an example of interpolation.

src/app/app.component.html

```
<h3>Current customer: {{ currentCustomer }}</h3>
```

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.

src/app/app.component.html

```
<p>{{title}}</p>
<div><img src="{{itemImageUrl}}"></div>
```

In the example above, Angular evaluates the `title` and `itemImageUrl` properties and fills in the blanks, first displaying some title text and then an 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 two numbers:

src/app/app.component.html

```
<!-- "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 such as `getVal()` in the following example:

src/app/app.component.html

```
<!-- "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 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. However, interpolation is a special syntax that Angular converts into a *property binding*.

> If you'd like to use something other than `{{` and `}}`, you can configure the interpolation delimiter via the interpolation option in the `Component` metadata.

Template expressions

A template **expression** produces a value and appears within the double curly braces, `{{ }}`. Angular executes the expression and assigns it to a property of a binding target; the target could 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, a template expression appears in quotes to the right of the = symbol as in `[property]="expression"`.

In terms of syntax, template expressions are similar to JavaScript. Many JavaScript expressions are legal template expressions, with a few exceptions.

You can't use JavaScript expressions that have or promote side effects, including:

- Assignments (`=`, `+=`, `-=`, `...`)
- Operators such as `new`, `typeof`, `instanceof`, etc.
- Chaining expressions with `;` or `,`
- The increment and decrement operators `++` and `--`
- Some of the ES2015+ operators

Other notable differences from JavaScript syntax include:

- No support for the bitwise operators such as `|` and `&`

- New template expression operators, such as `|`, `?.` and `!`

Expression context

The *expression context* is typically the *component* instance. In the following snippets, the `recommended` within double curly braces and the `itemImageUrl2` in quotes refer to properties of the `AppComponent`.

src/app/app.component.html

```
<h4>{{recommended}}</h4>
<img [src]="itemImageUrl2">
```

An expression may also refer to properties of the *template's* context such as a template input variable,

`let customer`, or a template reference variable, `#customerInput`.

src/app/app.component.html (template input variable)

```
<ul>
  <li *ngFor="let customer of customers">{{customer.name}}</li>
</ul>
```

src/app/app.component.html (template reference variable)

```
<label>Type something:
  <input #customerInput>{{customerInput.value}}
</label>
```

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 `customer` property and the `*ngFor` defines a `customer` template variable.

> The `customer` in `{{customer.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`. Additionally, they can't call `console.log()` or `Math.max()` and they are restricted to referencing members of the expression context.

Expression guidelines

When using template expressions follow these guidelines:

- Simplicity
- Quick execution
- No visible side effects

Simplicity

Although it's possible to write complex template expressions, it's a better practice to avoid them.

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

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, key presses 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.

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.

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.

> There is one exception to this behavior that applies to `*ngFor`. `*ngFor` has `trackBy` functionality that can deal with referential inequality of objects when iterating over them. See *ngFor with `trackBy` for details.

Press Kit	Report Issues	Twitter	日本語版
Blog	Code of Conduct	GitHub	한국어
Usage Analytics		Contribute	