Text interpolation

Text interpolation lets you incorporate dynamic string values into your HTML templates. Use interpolation to dynamically change what appears in an application view, such as displaying a custom greeting that includes the user's name.

See the for all of the syntax and code snippets in this guide.

Displaying values with interpolation

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

To illustrate how interpolation works, consider an Angular component that contains a `currentCustomer` variable:

Use interpolation to display the value of this variable in the corresponding component template:

Angular replaces `currentCustomer` with the string value of the corresponding component property. In this case, the value is `Maria`.

In the following example, Angular evaluates the `title` and `itemImageUrl` properties to display some title text and an image.

Template expressions

A template **expression** produces a value and appears within double curly braces, `{{ }}`. Angular resolves the expression and assigns it to a property of a binding target. The target could be an HTML element, a component, or a directive.

Resolving expressions with interpolation

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:

Expressions can also invoke methods of the host component such as `getVal()` in the following example:

With interpolation, Angular performs the following tasks:

1. Evaluates all expressions in double curly braces.
2. Converts the expression results to strings.
3. Links the results to any adjacent literal strings.
4. Assigns the composite to an element or directive property.

Configure the interpolation delimiter with the interpolation option in the `@Component()` metadata.

Syntax

Template expressions are similar to JavaScript. Many JavaScript expressions are legal template expressions, with the following exceptions.

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

- Assignments (`=`, `+=`, `-=`, `...`)
- Operators such as `new`, `typeof`, or `instanceof`
- 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

Interpolated expressions have a context—a particular part of the application to which the expression belongs. Typically, this context is the component instance.

In the following snippet, the expression `recommended` and the expression `itemImageUrl2` refer to properties of the `AppComponent`.

An expression can also refer to properties of the *template's* context such as a template input variable or a template reference variable.

The following example uses a template input variable of `customer`.

This next example features a template reference variable, `#customerInput`.

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.

Preventing name collisions

The context against which an expression evaluates is the union 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, Angular applies the following logic to determine the context:

1. The template variable name.
2. A name in the directive's context.
3. The component's member names.

To avoid variables shadowing variables in another context, keep variable names unique. In the following example, the `AppComponent` template greets the `customer`, Padma.

An `ngFor` then lists each `customer` in the `customers` array.

The `customer` within the `ngFor` is in the context of an `<ng-template>` and so refers to the `customer` in the `customers` array, in this case Ebony and Chiho. This list does not feature Padma because `customer` outside of the `ngFor` is in a different context. Conversely, `customer` in the `<h1>` doesn't include Ebony or Chiho because the context for this `customer` is the class and the class value for `customer` is Padma.

Expression best practices

When using template expressions, follow these best practices:

- **Use short expressions**

  Use property names or method calls whenever possible. Keep application and business logic in the component, where it is accessible to develop and test.

- **Quick execution**

  Angular executes template expressions after every change detection cycle. Many asynchronous activities trigger change detection cycles, such as promise resolutions, HTTP results, timer events, key presses and mouse moves.

  Expressions should finish quickly to keep the user experience as efficient as possible, especially on slower devices. Consider caching values when their computation requires greater resources.

- **No visible side effects**

  According to Angular's unidirectional data flow model, a template expression should not change any application state other than the value of the target property. Reading a component value should not change some other displayed value. The view should be stable throughout a single rendering pass.

  Idempotent expressions reduce side effects

  An idempotent expression 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 if you call it twice consecutively. If the expression returns an object, including an `array`, it returns the same object *reference* if you call it twice consecutively.

There is one exception to this behavior that applies to `*ngFor`. `*ngFor` has `trackBy` functionality that can deal with changing values in objects when iterating over them. See *ngFor with `trackBy` for details.