Template expression operators

Contents

The pipe operator (|)

The safe navigation operator (?) and null property paths

The non-null assertion operator (!)

The `$any()` type cast function

The Angular template expression language employs a subset of JavaScript syntax supplemented with a few special operators for specific scenarios. The next sections cover three of these operators:

- pipe
- safe navigation operator
- non-null assertion operator

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

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, change text to uppercase, or filter a list and sort it.

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 (|):

src/app/app.component.html

```
<p>Title through uppercase pipe: {{title | uppercase}}</p>
```

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:

src/app/app.component.html

```
<!-- convert title to uppercase, then to lowercase -->
<p>Title through a pipe chain: {{title | uppercase | lowercase}}</p>
```

And you can also apply parameters to a pipe:

src/app/app.component.html

```
<!-- pipe with configuration argument => "February 25, 1980" -->
<p>Manufacture date with date format pipe: {{item.manufactureDate | date:'longDate'}}
</p>
```

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

src/app/app.component.html

```
<p>Item json pipe: {{item | json}}</p>
```

The generated output would look something like this:

```
{ "name": "Telephone",
  "manufactureDate": "1980-02-25T05:00:00.000Z",
  "price": 98 }
```

> The pipe operator has a higher precedence than the ternary operator (`?:`), which means `a ? b : c | x` is parsed as `a ? b : (c | x)`. Nevertheless, for a number of reasons, the pipe operator cannot be used without parentheses in the first and second operands of `?:`. A good practice is to use parentheses in the third operand too.

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

The Angular safe navigation operator, `?`, guards against `null` and `undefined` values in property paths. Here, it protects against a view render failure if `item` is `null`.

src/app/app.component.html

```
<p>The item name is: {{item?.name}}</p>
```

If `item` is `null`, the view still renders but the displayed value is blank; you see only "The item name is:" with nothing after it.

Consider the next example, with a `nullItem`.

```
The null item name is {{nullItem.name}}
```

Since there is no safe navigation operator and `nullItem` is `null`, JavaScript and Angular would throw a `null` reference error and break the rendering process of Angular:

```
TypeError: Cannot read property 'name' of null.
```

Sometimes however, `null` values in the property path may be OK under certain circumstances, especially when the value starts out null but the data arrives eventually.

With the safe navigation operator, `?`, Angular stops evaluating the expression when it hits the first `null` value and renders the view without errors.

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

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 tell the type checker not to throw an error by applying the postfix non-null assertion operator, `!`.

The Angular non-null assertion operator, `!`, serves the same purpose in an Angular template. For example, you can assert that `item` properties are also defined.

src/app/app.component.html

```
<!-- Assert color is defined, even if according to the `Item` type it could be
undefined. -->
<p>The item's color is: {{item.color!.toUpperCase()}}</p>
```

When the Angular compiler turns your template into TypeScript code, it prevents TypeScript from reporting that `item.color` 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.

The non-null assertion operator, `!`, is optional with the exception that you must use it when you turn on strict null checks.

The `$any()` type cast function

Sometimes a binding expression triggers a type error during AOT compilation and it is not possible or difficult to fully specify the type. To silence the error, you can use the `$any()` cast function to cast the expression to the `any` type as in the following example:

src/app/app.component.html

```
<p>The item's undeclared best by date is: {{$any(item).bestByDate}}</p>
```

When the Angular compiler turns this template into TypeScript code, it prevents TypeScript from reporting that `bestByDate` is not a member of the `item` object when it runs type checking on the template.

The `$any()` cast function also works with `this` to allow access to undeclared members of the component.

src/app/app.component.html

```
<p>The item's undeclared best by date is: {{$any(this).bestByDate}}</p>
```

The `$any()` cast function works anywhere in a binding expression where a method call is valid.

Version 10.0.10-local+sha.b32126c335.