Writing structural directives

This topic demonstrates how to create a structural directive and provides conceptual information on how directives work, how Angular interprets shorthand, and how to add template guard properties to catch template type errors.

For the example application that this page describes, see the .

For more information on Angular's built-in structural directives, such as `NgIf`, `NgForOf`, and `NgSwitch`, see Built-in directives.

{@a unless}

Creating a structural directive

This section guides you through creating an `UnlessDirective` and how to set `condition` values. The `UnlessDirective` does the opposite of `NgIf`, and `condition` values can be set to `true` or `false`. `NgIf` displays the template content when the condition is `true`. `UnlessDirective` displays the content when the condition is `false`.

Following is the `UnlessDirective` selector, `appUnless`, applied to the paragraph element. When `condition` is `false`, the browser displays the sentence.

1. Using the Angular CLI, run the following command, where `unless` is the name of the directive:

```
ng generate directive unless
```

Angular creates the directive class and specifies the CSS selector, `appUnless`, that identifies the directive in a template.

1. Import `Input`, `TemplateRef`, and `ViewContainerRef`.

1. Inject `TemplateRef` and `ViewContainerRef` in the directive constructor as private variables.

The `UnlessDirective` creates an embedded view from the Angular-generated `<ng-template>` and inserts that view in a view container adjacent to the directive's original `<p>` host element.

`TemplateRef` helps you get to the `<ng-template>` contents and `ViewContainerRef` accesses the view container.

1. Add an `appUnless` `@Input()` property with a setter.

Angular sets the `appUnless` property whenever the value of the condition changes.

```
* If the condition is falsy and Angular hasn't created the view previously, the setter cause

* If the condition is truthy and the view is currently displayed, the setter clears the cont
```

The complete directive is as follows:

Testing the directive

In this section, you'll update your application to test the `UnlessDirective`.

1. Add a `condition` set to `false` in the `AppComponent`.

1. Update the template to use the directive. Here, `*appUnless` is on two `<p>` tags with opposite `condition` values, one `true` and one `false`.

The asterisk is shorthand that marks `appUnless` as a structural directive. When the `condition` is falsy, the top (A) paragraph appears and the bottom (B) paragraph disappears. When the `condition` is truthy, the top (A) paragraph disappears and the bottom (B) paragraph appears.

1. To change and display the value of `condition` in the browser, add markup that displays the status and a button.

To verify that the directive works, click the button to change the value of `condition`.

`<img src='generated/images/guide/structural-directives/unless-anim.gif' alt="UnlessDirective`

{@a shorthand} {@a asterisk}

Structural directive shorthand

The asterisk, `*`, syntax on a structural directive, such as `*ngIf`, is shorthand that Angular interprets into a longer form. Angular transforms the asterisk in front of a structural directive into an `<ng-template>` that surrounds the host element and its descendants.

The following is an example of `*ngIf` that displays the hero's name if `hero` exists:

The `*ngIf` directive moves to the `<ng-template>` where it becomes a property binding in square brackets, `[ngIf]`. The rest of the `<div>`, including its class attribute, moves inside the `<ng-template>`.

Angular does not create a real `<ng-template>` element, instead rendering only the `<div>` and a comment node placeholder to the DOM.

```
<!--bindings={
  "ng-reflect-ng-if": "[object Object]"
}-->
<div _ngcontent-c0>Mr. Nice</div>
```

The following example compares the shorthand use of the asterisk in `*ngFor` with the longhand `<ng-template>` form:

Here, everything related to the `ngFor` structural directive applies to the `<ng-template>`. All other bindings and attributes on the element apply to the

`<div>` element within the `<ng-template>`. Other modifiers on the host element, in addition to the `ngFor` string, remain in place as the element moves inside the `<ng-template>`. In this example, the `[class.odd]="odd"` stays on the `<div>`.

The `let` keyword declares a template input variable that you can reference within the template. The input variables in this example are `hero`, `i`, and `odd`. The parser translates `let hero`, `let i`, and `let odd` into variables named `let-hero`, `let-i`, and `let-odd`. The `let-i` and `let-odd` variables become `let i=index` and `let odd=odd`. Angular sets `i` and `odd` to the current value of the context's `index` and `odd` properties.

The parser applies PascalCase to all directives and prefixes them with the directive's attribute name, such as ngFor. For example, the `ngFor` input properties, `of` and `trackBy`, map to `ngForOf` and `ngForTrackBy`. As the `NgFor` directive loops through the list, it sets and resets properties of its own context object. These properties can include, but aren't limited to, `index`, `odd`, and a special property named `$implicit`.

Angular sets `let-hero` to the value of the context's `$implicit` property, which `NgFor` has initialized with the hero for the current iteration.

For more information, see the NgFor API and NgForOf API documentation.

Creating template fragments with `<ng-template>`

Angular's `<ng-template>` element defines a template that doesn't render anything by default. With `<ng-template>`, you can render the content manually for full control over how the content displays.

If there is no structural directive and you wrap some elements in an `<ng-template>`, those elements disappear. In the following example, Angular does not render the middle "Hip!" in the phrase "Hip! Hip! Hooray!" because of the surrounding `<ng-template>`.

Structural directive syntax reference

When you write your own structural directives, use the following syntax:

```
*:prefix="( :let | :expression ) (';' | ',')? ( :let | :as | :keyExp )*"
```

The following tables describe each portion of the structural directive grammar:

prefix

HTML attribute key

key

HTML attribute key

local

local variable name used in the template

export

value exported by the directive under a given name

expression

standard Angular expression

keyExp = :key “:”? :expression (“as” :local)? “;”?

let = “let” :local “=” :export “;”?

as = :export “as” :local “;”?

How Angular translates shorthand

Angular translates structural directive shorthand into the normal binding syntax as follows:

Shorthand

Translation

prefix and naked expression

[prefix]=“expression”

keyExp

[prefixKey] “expression” (let-prefixKey=“export”) Notice that the prefix is added to the key

let

let-local=“export”

Shorthand examples

The following table provides shorthand examples:

Shorthand

How Angular interprets the syntax

*ngFor=“let item of [1,2,3]”

<ng-template ngFor let-item [ngForOf]=“[1,2,3]”>

*ngFor=“let item of [1,2,3] as items; trackBy: myTrack; index as i”

<ng-template ngFor let-item [ngForOf]=“[1,2,3]” let-items=“ngForOf” [ngForTrackBy]=“myTrack” let-i=“index”>

*ngIf=“exp”

<ng-template [ngIf]=“exp”>

*ngIf=“exp as value”

<ng-template [ngIf]=“exp” let-value=“ngIf”>

{@a directive-type-checks}

Improving template type checking for custom directives

You can improve template type checking for custom directives by adding template guard properties to your directive definition. These properties help the Angular template type checker find mistakes in the template at compile time, which can avoid runtime errors. These properties are as follows:

- A property `ngTemplateGuard_(someInputProperty)` lets you specify a more accurate type for an input expression within the template.
- The `ngTemplateContextGuard` static property declares the type of the template context.

This section provides examples of both kinds of type-guard property. For more information, see Template type checking.

{@a narrowing-input-types}

Making in-template type requirements more specific with template guards

A structural directive in a template controls whether that template is rendered at run time, based on its input expression. To help the compiler catch template type errors, you should specify as closely as possible the required type of a directive's input expression when it occurs inside the template.

A type guard function narrows the expected type of an input expression to a subset of types that might be passed to the directive within the template at run time. You can provide such a function to help the type-checker infer the proper type for the expression at compile time.

For example, the `NgIf` implementation uses type-narrowing to ensure that the template is only instantiated if the input expression to `*ngIf` is truthy. To provide the specific type requirement, the `NgIf` directive defines a static property `ngTemplateGuard_ngIf: 'binding'`. The `binding` value is a special case for a common kind of type-narrowing where the input expression is evaluated in order to satisfy the type requirement.

To provide a more specific type for an input expression to a directive within the template, add an `ngTemplateGuard_xx` property to the directive, where the suffix to the static property name, `xx`, is the `@Input()` field name. The value of the property can be either a general type-narrowing function based on its return type, or the string `"binding"`, as in the case of `NgIf`.

For example, consider the following structural directive that takes the result of a template expression as an input:

export type Loaded = { type: ‘loaded’, data: T }; export type Loading = { type: ‘loading’ }; export type LoadingState = Loaded | Loading; export class IfLoadedDirective { @Input(‘ifLoaded’) set state(state: LoadingState) {} static ngTemplateGuard_state(dir: IfLoadedDirective, expr: LoadingState): expr is Loaded { return true; }; }

export interface Person { name: string; }

@Component({ template: `<div *ifLoaded="state">{{ state.data }}</div>`, }) export class AppComponent { state: LoadingState; }

In this example, the `LoadingState<T>` type permits either of two states, `Loaded<T>` or `Loading`. The expression used as the directive's `state` input is of the umbrella type `LoadingState`, as it's unknown what the loading state is at that point.

The `IfLoadedDirective` definition declares the static field `ngTemplateGuard_state`, which expresses the narrowing behavior. Within the `AppComponent` template, the `*ifLoaded` structural directive should render this template only when `state` is actually `Loaded<Person>`. The type guard lets the type checker infer that the acceptable type of `state` within the template is a `Loaded<T>`, and further infer that `T` must be an instance of `Person`.

{@a narrowing-context-type}

Typing the directive's context

If your structural directive provides a context to the instantiated template, you can properly type it inside the template by providing a static `ngTemplateContextGuard` function. The following snippet shows an example of such a function.

@Directive({...}) export class ExampleDirective { // Make sure the template checker knows the type of the context with which the // template of this directive will be rendered static ngTemplateContextGuard(dir: ExampleDirective, ctx: unknown): ctx is ExampleContext { return true; };

```
// ...
```

}