Published in ITNEXT

Thomas Laforge Follow
Dec 5 · 4 min read · Listen

Directive Type Checking

Directives are a very powerful tool that we should master to improve our Angular skills. Typescript enforces strict types and helps us make our codebase more resilient. Unfortunately, custom directives are not completely typed out of the box. The purpose of the article is to show you how to add strict typing to your structural directives and your Angular codebase more resilient.

. . .

To support type checking, we will …th two handy type guards provided by Angular:

- **ngTemplateContextGuard:** Declare a custom type for the context of our custom directive
- **ngTemplateGuard_[customInputProperty]:** Narrow the rendered type of an input property.

Before we dive into these static Angular functions we need to understand how Typescript type predicates work. I invite you to read this article if you don't understand or never heard of it, otherwise skip ahead.

. . .

ngTemplateContextGuard

If we need to provide a context to our custom directive, we can ensure it is properly typed in the template using the static function **ngTemplateContextGuard.** It works like a Typescript type guard function and returns a type predicate.

Let's look at an example to better understand the concept.

```
interface DemoUrl {
  url: string;
  video: boolean;
}

// interface declaring the Context of this Directive
interface DemoContext {
  $implicit: number;
  demo: string;
  url: DemoUrl;
}

@Directive({
  selector: '[demo]',
  standalone: true,
})
export class DemoDirective implements OnInit {
  @Input() demo!: string;
  @Input() demoUrl!: DemoUrl;

  constructor(
    private readonly viewContainerRef: ViewContainerRef,
    private readonly templateRef: TemplateRef<DemoContext>
```

```
  ) {}

  ngOnInit(): void {
    const context = {
      $implicit: 1,
      demo: this.demo,
      url: this.demoUrl,
    };
    this.viewContainerRef.createEmbeddedView(this.templateRef, context);
  }

  // Guard to help Typescript correctly type checked
  // the context with which the template will be rendered
  static ngTemplateContextGuard(
    directive: DemoDirective,
    context: unknown
  ): context is DemoContext {
    return true;
  }
}
```

The ngTemplateContextGuard returns true since this directive will always pass a context of type DemoContext to the template.

Now when we use this directive in our template, we get nice correctly typed properties.

> ***Bonus tip:*** *we can write structural directives in three different ways. All are interpreted the same way by the compiler.*

```
<ng-template demo="toto" [demoUrl]="demoUrl" let-version let-demo="demo" let-ur
  {{ url.url }}
</ng-template>

// * is the shorthand for what angular will interpret with ng-template
<div *demo="'toto'; let version; url: demoUrl; let demo = demo; let url = url">
  {{ url.url }}
</div>

// We can use "as" in replacement of "let ... = "
```

```
<div *demo="'toto' as version; url: demoUrl; demo as demo; url as url">
  {{ url.url }}
</div>
```

ngTemplateGuard_[customInputProperty]

This guard is a bit more complex to understand. A structural directive controls how a template will be rendered at runtime. *(NgIf for example, will add a template to the DOM only if the input condition is thrustly.)*

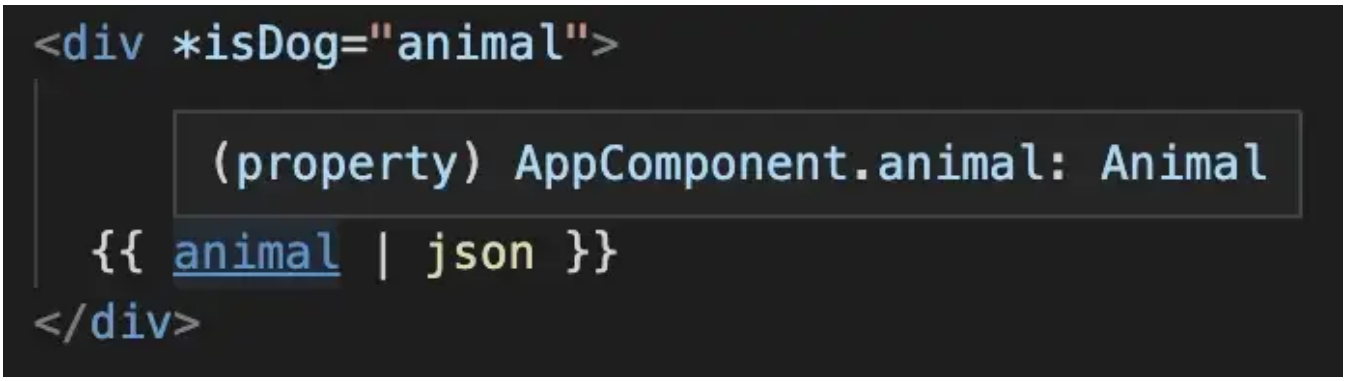
If the input of our custom directive has a complex type and the directive will only render the template when certain condition are met, we can narrow the rendered type with this guard.

Let's show this example to better understand this concept:

```
// Typescript type guard
export const isDog = (animal: Animal): animal is Dog => {
  return (animal as Dog).breed !== undefined;
}

interface Cat {
  name: string;
  type: 'cat';
}

interface Dog {
  name: string;
  race: string;
  type: 'dog';
}

type Animal = Dog | Cat;

@Directive({
  selector: '[isDog]',
  standalone: true,
})
export class DogDirective {
  @Input('isDog') set isDogInput(animal: Animal) {
    if (isDog(animal.type)) {
      this.viewContainerRef.createEmbeddedView(this.templateRef);
    } else {
      this.viewContainerRef.clear();
    }
```

```
  }

  constructor(
    private readonly viewContainerRef: ViewContainerRef,
    private readonly templateRef: TemplateRef<unknown>
  ) {}
```

This directive takes an `Animal` as input and renders the template only if the input animal is of type `Dog`. Even though we are 100% sure that this template will only be rendered when our input is of type `Dog`, we can see that the animal variable inside our template is still of type `Animal`.



without **ngTemplateGuard**

This is where **ngTemplateGuard** comes to the rescue. We use the following function to narrow our type to `Dog` and enjoy better type safety in our template.

```
static ngTemplateGuard_isDog(
  dir: DogDirective,
  state: Animal // input type
): state is Dog { // output type
  return true;
}
```

We can now see the correctly inferred type in our template

with **ngTemplateGuard**

Remark: If we decide to change the name used in template that is bound to our internal input name like *@Input('isDogExternal') isDog*, the guard takes the external value. It means the guard will become *ngTemplateGuard_isDogExternal*.

However aliasing should be avoided unless we have a good reason to do so. Having two names for the same property (external and internal) is confusing. *(Check Angular Style Guide.)*

. . .

That's it! You no longer have any excuses for not strictly typing your custom directives...

If you found this article useful, please consider supporting my work by giving it some claps👏👏 to help it reach a wider audience. Don't forget to share it with your teammates who might also find it useful. Your support would be greatly appreciated.

I hope you learned new Angular concept. You can find me on Medium, Twitter or Github. Don't hesitate to ping me if you have more questions

> 👉 And if you want to accelerate your Angular learning journey, come and check out Angular challenges.

Angular Type Typescript Directive

Thanks to Brecht Billiet and Robin