NgIf DIRECTIVE

A structural directive that conditionally includes a template based on the value of an expression coerced to Boolean. When the expression evaluates to true, Angular renders the template provided in a `then` clause, and when false or null, Angular renders the template provided in an optional `else` clause. The default template for the `else` clause is blank.

See more...

Exported from

- `CommonModule`

Selectors

`[ngIf]`

Properties

Property	Description
`@Input()` `ngIf: T`	***Write-Only*** The Boolean expression to evaluate as the condition for showing a template.

`@Input()`
`ngIfThen:`
`TemplateRef<NgIfContext<T>>`
	Write-Only
	A template to show if the condition expression evaluates to true.

`@Input()`
`ngIfElse:`
`TemplateRef<NgIfContext<T>>`
	Write-Only
	A template to show if the condition expression evaluates to false.

Description

A shorthand form of the directive, `*ngIf="condition"`, is generally used, provided as an attribute of the anchor element for the inserted template. Angular expands this into a more explicit version, in which the anchor element is contained in an `<ng-template>` element.

Simple form with shorthand syntax:

```
<div *ngIf="condition">Content to render when condition
is true.</div>
```

Simple form with expanded syntax:

```
<ng-template [ngIf]="condition"><div>Content to render
when condition is
true.</div></ng-template>
```

Form with an "else" block:

```
<div *ngIf="condition; else elseBlock">Content to
render when condition is true.</div>
<ng-template #elseBlock>Content to render when
condition is false.</ng-template>
```

Shorthand form with "then" and "else" blocks:

```
<div *ngIf="condition; then thenBlock else elseBlock">
</div>
<ng-template #thenBlock>Content to render when
condition is true.</ng-template>
<ng-template #elseBlock>Content to render when
condition is false.</ng-template>
```

Form with storing the value locally:

```
<div *ngIf="condition as value; else elseBlock">
{{value}}</div>
<ng-template #elseBlock>Content to render when value is
null.</ng-template>
```

The `*ngIf` directive is most commonly used to conditionally show an inline template, as seen in the following example. The default `else` template is blank.

```
@Component({
  selector: 'ng-if-simple',
  template: `
    <button (click)="show = !show">{{show ? 'hide' :
'show'}}</button>
    show = {{show}}
    <br>
    <div *ngIf="show">Text to show</div>
`
})
export class NgIfSimple {
  show = true;
}
```

Showing an alternative template using `else`

To display a template when `expression` evaluates to false, use an `else` template binding as shown in the following example. The `else` binding points to an `<ng-template>` element labeled `#elseBlock`. The template can be defined anywhere in the component view, but is typically placed right after `ngIf` for readability.

```
@Component({
  selector: 'ng-if-else',
  template: `
    <button (click)="show = !show">{{show ? 'hide' :
'show'}}</button>
    show = {{show}}
    <br>
    <div *ngIf="show; else elseBlock">Text to
show</div>
    <ng-template #elseBlock>Alternate text while
primary text is hidden</ng-template>
`
})
export class NgIfElse {
  show = true;
}
```

Using an external `then` template

In the previous example, the then-clause template is specified inline, as the content of the tag that contains the `ngIf` directive. You can also specify a template that is defined externally, by referencing a labeled `<ng-template>` element. When you do this, you can change which template to use at runtime, as shown in the following example.

```
@Component({
  selector: 'ng-if-then-else',
  template: `
    <button (click)="show = !show">{{show ? 'hide' :
'show'}}</button>
    <button (click)="switchPrimary()">Switch
Primary</button>
    show = {{show}}
    <br>
    <div *ngIf="show; then thenBlock; else
elseBlock">this is ignored</div>
    <ng-template #primaryBlock>Primary text to
show</ng-template>
    <ng-template #secondaryBlock>Secondary text to
show</ng-template>
    <ng-template #elseBlock>Alternate text while
primary text is hidden</ng-template>
`
})
export class NgIfThenElse implements OnInit {
  thenBlock: TemplateRef<any>|null = null;
  show = true;

  @ViewChild('primaryBlock', {static: true})
primaryBlock: TemplateRef<any>|null = null;
  @ViewChild('secondaryBlock', {static: true})
secondaryBlock: TemplateRef<any>|null = null;

  switchPrimary() {
    this.thenBlock = this.thenBlock ===
this.primaryBlock ? this.secondaryBlock :
this.primaryBlock;
  }

  ngOnInit() {
    this.thenBlock = this.primaryBlock;
```

```
    }
  }
```

Storing a conditional result in a variable

You might want to show a set of properties from the same object. If you are waiting for asynchronous data, the object can be undefined. In this case, you can use `ngIf` and store the result of the condition in a local variable as shown in the following example.

```
@Component({
  selector: 'ng-if-as',
  template: `
    <button (click)="nextUser()">Next User</button>
    <br>
    <div *ngIf="userObservable | async as user; else
loading">
      Hello {{user.last}}, {{user.first}}!
    </div>
    <ng-template #loading let-user>Waiting... (user is
{{user|json}})</ng-template>
`
})
export class NgIfAs {
  userObservable = new Subject<{first: string, last:
string}>();
  first = ['John', 'Mike', 'Mary', 'Bob'];
  firstIndex = 0;
  last = ['Smith', 'Novotny', 'Angular'];
  lastIndex = 0;

  nextUser() {
    let first = this.first[this.firstIndex++];
    if (this.firstIndex >= this.first.length)
this.firstIndex = 0;
    let last = this.last[this.lastIndex++];
    if (this.lastIndex >= this.last.length)
this.lastIndex = 0;
    this.userObservable.next({first, last});
  }
}
```

This code uses only one `AsyncPipe`, so only one subscription is created. The conditional statement stores the result of `userStream|async` in the local variable `user`. You can then bind the local `user` repeatedly.

The conditional displays the data only if `userStream` returns a value, so

you don't need to use the safe-navigation-operator (`?.`) to guard against null values when accessing properties. You can display an alternative template while waiting for the data.

Shorthand syntax

The shorthand syntax `*ngIf` expands into two separate template specifications for the "then" and "else" clauses. For example, consider the following shorthand statement, that is meant to show a loading page while waiting for data to be loaded.

```
<div class="hero-list" *ngIf="heroes else loading">
 ...
</div>

<ng-template #loading>
 <div>Loading...</div>
</ng-template>
```

You can see that the "else" clause references the `<ng-template>` with the `#loading` label, and the template for the "then" clause is provided as the content of the anchor element.

However, when Angular expands the shorthand syntax, it creates another `<ng-template>` tag, with `ngIf` and `ngIfElse` directives. The anchor element containing the template for the "then" clause becomes the content of this unlabeled `<ng-template>` tag.

```
<ng-template [ngIf]="heroes" [ngIfElse]="loading">
 <div class="hero-list">
  ...
 </div>
</ng-template>

<ng-template #loading>
 <div>Loading...</div>
</ng-template>
```

The presence of the implicit template object has implications for the nesting of structural directives. For more on this subject, see Structural Directives.

Static properties

Property	Description
`static ngTemplateGuard_ngIf: 'binding'`	Assert the correct type of the expression bound to the `ngIf` input within the template. The presence of this static field is a signal to the Ivy template type check compiler that when the `NgIf` structural directive renders its template, the type of the expression bound to `ngIf` should be narrowed in some way. For `NgIf`, the binding expression itself is used to narrow its type, which allows the strictNullChecks feature of TypeScript to work with `NgIf`.

Static methods

ngTemplateContextGuard()

Asserts the correct type of the context for the template that `NgIf` will render.

```
static ngTemplateContextGuard<T>(dir: NgIf<T>, ctx: any):
ctx is NgIfContext<Exclude<T, false | 0 | '' | null |
undefined>>
```

Parameters

dir	`NgIf<T>`
ctx	`any`

Returns

`ctx is NgIfContext<Exclude<T, false | 0 | '' | null | undefined>>`

The presence of this method is a signal to the Ivy template type-check compiler that the `NgIf` structural directive renders its template with a specific context type.