Built-in directives

Contents ›

Angular offers two kinds of built-in directives: *attribute* directives and *structural* directives.

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

For more detail, including how to build your own custom directives, see Attribute Directives and Structural Directives.

Built-in attribute directives

Attribute directives listen to and modify the behavior of other HTML elements, attributes, properties, and components. You usually apply them to elements as if they were HTML attributes, hence the name.

Many NgModules such as the `RouterModule` and the `FormsModule` define their own attribute directives. The most common attribute directives are as follows:

- `NgClass`—adds and removes a set of CSS classes.
- `NgStyle`—adds and removes a set of HTML styles.
- `NgModel`—adds two-way data binding to an HTML form element.

`NgClass`

Add or remove several CSS classes simultaneously with `ngClass`.

src/app/app.component.html

```
<!-- toggle the "special" class on/off with a property -->
<div [ngClass]="isSpecial ? 'special' : ''">This div is special</div>
```

> To add or remove a *single* class, use class binding rather than `NgClass`.

Consider a `setCurrentClasses()` component method that sets a component property, `currentClasses`, with an object that adds or removes three classes based on the `true/false` state of three other component properties. Each key of the object is a CSS class name; its value is `true` if the class should be added, `false` if it should be removed.

src/app/app.component.ts

```
currentClasses: {};
/* . . . */
  setCurrentClasses() {
    // CSS classes: added/removed per current state of component properties
    this.currentClasses =  {
      saveable: this.canSave,
      modified: !this.isUnchanged,
      special:  this.isSpecial
    };
  }
```

Adding an `ngClass` property binding to `currentClasses` sets the element's classes accordingly:

src/app/app.component.html

```
<div [ngClass]="currentClasses">This div is initially saveable, unchanged, and special.
</div>
```

Remember that in this situation you'd call `setCurrentClasses()`, both initially and when the dependent properties change.

`NgStyle`

Use `NgStyle` to set many inline styles simultaneously and dynamically, based on the state of the component.

Without `NgStyle`

For context, consider setting a *single* style value with style binding, without `NgStyle`.

src/app/app.component.html

```
<div [style.font-size]="isSpecial ? 'x-large' : 'smaller'">
  This div is x-large or smaller.
</div>
```

However, to set *many* inline styles at the same time, use the `NgStyle` directive.

The following is a `setCurrentStyles()` method that sets a component property, `currentStyles`, with an object that defines three styles, based on the state of three other component properties:

src/app/app.component.ts

```
currentStyles: {};
/* . . . */
  setCurrentStyles() {
    // CSS styles: set per current state of component properties
    this.currentStyles = {
      'font-style':  this.canSave      ? 'italic' : 'normal',
      'font-weight': !this.isUnchanged ? 'bold'   : 'normal',
      'font-size':   this.isSpecial    ? '24px'   : '12px'
    };
  }
```

Adding an `ngStyle` property binding to `currentStyles` sets the element's styles accordingly:

src/app/app.component.html

```
<div [ngStyle]="currentStyles">
  This div is initially italic, normal weight, and extra large (24px).
</div>
```

> Remember to call `setCurrentStyles()`, both initially and when the dependent properties change.

`[(ngModel)]`: Two-way binding

The `NgModel` directive allows you to display a data property and update that property when the user makes changes. Here's an example:

src/app/app.component.html (NgModel example)

```
<label for="example-ngModel">[(ngModel)]:</label>
<input [(ngModel)]="currentItem.name" id="example-ngModel">
```

Import `FormsModule` to use `ngModel`

Before using the `ngModel` directive in a two-way data binding, you must import the `FormsModule` and add it to the NgModule's `imports` list. Learn more about the `FormsModule` and `ngModel` in Forms.

Remember to import the `FormsModule` to make `[(ngModel)]` available as follows:

src/app/app.module.ts (FormsModule import)

```
import { FormsModule } from '@angular/forms'; // <--- JavaScript import from Angular
/* . . . */
@NgModule({
/* . . . */

  imports: [
    BrowserModule,
    FormsModule // <--- import into the NgModule
  ],
/* . . . */
})
export class AppModule { }
```

You could achieve the same result with separate bindings to the `<input>` element's `value` property and `input` event:

src/app/app.component.html

```
<label for="without">without NgModel:</label>
<input [value]="currentItem.name" (input)="currentItem.name=$event.target.value"
id="without">
```

To streamline the syntax, the `ngModel` directive hides the details behind its own `ngModel` input and `ngModelChange` output properties:

src/app/app.component.html

```
<label for="example-change">(ngModelChange)="...name=$event":</label>
<input [ngModel]="currentItem.name" (ngModelChange)="currentItem.name=$event"
id="example-change">
```

The `ngModel` data property sets the element's value property and the `ngModelChange` event property listens for changes to the element's value.

`NgModel` and value accessors

The details are specific to each kind of element and therefore the `NgModel` directive only works for an element supported by a ControlValueAccessor that adapts an element to this protocol. Angular provides *value accessors* for all of the basic HTML form elements and the Forms guide shows how to bind to them.

You can't apply `[(ngModel)]` to a non-form native element or a third-party custom component until you write a suitable value accessor. For more information, see the API documentation on DefaultValueAccessor.

You don't need a value accessor for an Angular component that you write because you can name the value and event properties to suit Angular's basic two-way binding syntax and skip `NgModel` altogether. The `sizer` in the Two-way Binding section is an example of this technique.

Separate `ngModel` bindings are an improvement over binding to the element's native properties, but you can streamline the binding with a single declaration using the `[(ngModel)]` syntax:

src/app/app.component.html

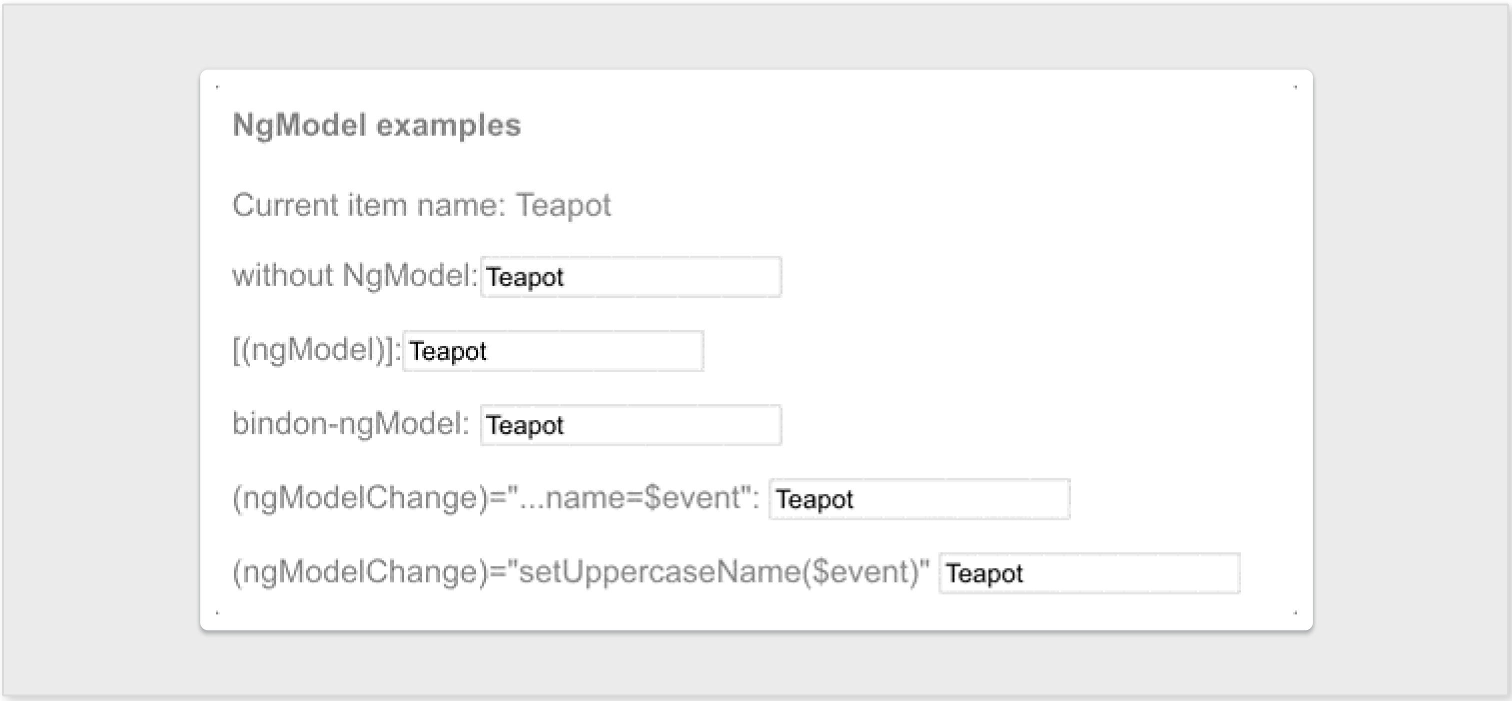
```
<label for="example-ngModel">[(ngModel)]:</label>
<input [(ngModel)]="currentItem.name" id="example-ngModel">
```

This `[(ngModel)]` syntax can only *set* a data-bound property. If you need to do something more, you can write the expanded form; for example, the following changes the `<input>` value to uppercase:

src/app/app.component.html

```
<input [ngModel]="currentItem.name" (ngModelChange)="setUppercaseName($event)"
id="example-uppercase">
```

Here are all variations in action, including the uppercase version:


NgModel examples

Current item name: Teapot

without NgModel: Teapot

[(ngModel)]: Teapot

bindon-ngModel: Teapot

(ngModelChange)="...name=$event": Teapot

(ngModelChange)="setUppercaseName($event)" Teapot

Built-in *structural* directives

Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, and manipulating the host elements to which they are attached.

This section is an introduction to the common built-in structural directives:

- `NgIf`—conditionally creates or destroys subviews from the template.
- `NgFor`—repeat a node for each item in a list.
- `NgSwitch`—a set of directives that switch among alternative views.

The deep details of structural directives are covered in the Structural Directives guide, which explains the following:

- Why you prefix the directive name with an asterisk (*).
- Using `<ng-container>` to group elements when there is no suitable host element for the directive.
- How to write your own structural directive.
- That you can only apply one structural directive to an element.

NgIf

You can add or remove an element from the DOM by applying an `NgIf` directive to a host element. Bind the directive to a condition expression like `isActive` in this example.

src/app/app.component.html

```
<app-item-detail *ngIf="isActive" [item]="item"></app-item-detail>
```

Don't forget the asterisk (`*`) in front of `ngIf`. For more information on the asterisk, see the asterisk (*) prefix section of Structural Directives.

When the `isActive` expression returns a truthy value, `NgIf` adds the `ItemDetailComponent` to the DOM. When the expression is falsy, `NgIf` removes the `ItemDetailComponent` from the DOM, destroying that component and all of its sub-components.

Show/hide vs. `NgIf`

Hiding an element is different from removing it with `NgIf`. For comparison, the following example shows how to control the visibility of an element with a class or style binding.

src/app/app.component.html

```
<!-- isSpecial is true -->
<div [class.hidden]="!isSpecial">Show with class</div>
<div [class.hidden]="isSpecial">Hide with class</div>

<p>ItemDetail is in the DOM but hidden</p>
<app-item-detail [class.hidden]="isSpecial"></app-item-detail>

<div [style.display]="isSpecial ? 'block' : 'none'">Show with style</div>
<div [style.display]="isSpecial ? 'none'  : 'block'">Hide with style</div>
```

When you hide an element, that element and all of its descendants remain in the DOM. All components for those elements stay in memory and Angular may continue to check for changes. You could be holding onto considerable computing resources and degrading performance unnecessarily.

`NgIf` works differently. When `NgIf` is `false`, Angular removes the element and its descendants from the DOM. It destroys their components, freeing up resources, which results in a better user experience.

If you are hiding large component trees, consider `NgIf` as a more efficient alternative to showing/hiding.

> For more information on `NgIf` and `ngIfElse`, see the API documentation about NgIf.

Guard against null

Another advantage of `ngIf` is that you can use it to guard against null. Show/hide is best suited for very simple use cases, so when you need a guard, opt instead for `ngIf`. Angular will throw an error if a nested expression tries to access a property of `null`.

The following shows `NgIf` guarding two `<div>`s. The `currentCustomer` name appears only when there is a `currentCustomer`. The `nullCustomer` will not be displayed as long as it is `null`.

src/app/app.component.html

```
<div *ngIf="currentCustomer">Hello, {{currentCustomer.name}}</div>
```

src/app/app.component.html

```
<div *ngIf="nullCustomer">Hello, <span>{{nullCustomer}}</span></div>
```

> See also the safe navigation operator below.

NgFor

`NgFor` is a repeater directive—a way to present a list of items. You define a block of HTML that defines how a single item should be displayed and then you tell Angular to use that block as a template for rendering each item in the list. The text assigned to `*ngFor` is the instruction that guides the repeater process.

The following example shows `NgFor` applied to a simple `<div>`. (Don't forget the asterisk (*) in front of `ngFor`.)

src/app/app.component.html

```
<div *ngFor="let item of items">{{item.name}}</div>
```

Don't forget the asterisk (`*`) in front of `ngFor`. For more information on the asterisk, see the asterisk (*) prefix section of Structural Directives.

You can also apply an `NgFor` to a component element, as in the following example.

src/app/app.component.html

```
<app-item-detail *ngFor="let item of items" [item]="item"></app-item-detail>
```

***NGFOR MICROSYNTAX**

The string assigned to `*ngFor` is not a template expression. Rather, it's a *microsyntax*—a little language of its own that Angular interprets. The string `"let item of items"` means:

> *Take each item in the `items` array, store it in the local `item` looping variable, and make it available to the templated HTML for each iteration.*

Angular translates this instruction into an `<ng-template>` around the host element, then uses this template repeatedly to create a new set of elements and bindings for each `item` in the list. For more information about microsyntax, see the Structural Directives guide.

Template input variables

The `let` keyword before `item` creates a template input variable called `item`. The `ngFor` directive iterates over the `items` array returned by the parent component's `items` property and sets `item` to the current item from the array during each iteration.

Reference `item` within the `ngFor` host element as well as within its descendants to access the item's properties. The following example references `item` first in an interpolation and then passes in a binding to the `item` property of the `<app-item-detail>` component.

src/app/app.component.html

```
<div *ngFor="let item of items">{{item.name}}</div>
<!-- . . . -->
  <app-item-detail *ngFor="let item of items" [item]="item"></app-item-detail>
```

For more information about template input variables, see Structural Directives.

`*ngFor` with `index`

The `index` property of the `NgFor` directive context returns the zero-based index of the item in each iteration. You can capture the `index` in a template input variable and use it in the template.

The next example captures the `index` in a variable named `i` and displays it with the item name.

src/app/app.component.html

```html
<div *ngFor="let item of items; let i=index">{{i + 1}} - {{item.name}}</div>
```

> `NgFor` is implemented by the `NgForOf` directive. Read more about the other `NgForOf` context values such as `last`, `even`, and `odd` in the NgForOf API reference.

*ngFor with `trackBy`

If you use `NgFor` with large lists, a small change to one item, such as removing or adding an item, can trigger a cascade of DOM manipulations. For example, re-querying the server could reset a list with all new item objects, even when those items were previously displayed. In this case, Angular sees only a fresh list of new object references and has no choice but to replace the old DOM elements with all new DOM elements.

You can make this more efficient with `trackBy`. Add a method to the component that returns the value `NgFor` should track. In this case, that value is the hero's `id`. If the `id` has already been rendered, Angular keeps track of it and doesn't re-query the server for the same `id`.

src/app/app.component.ts

```ts
trackByItems(index: number, item: Item): number { return item.id; }
```

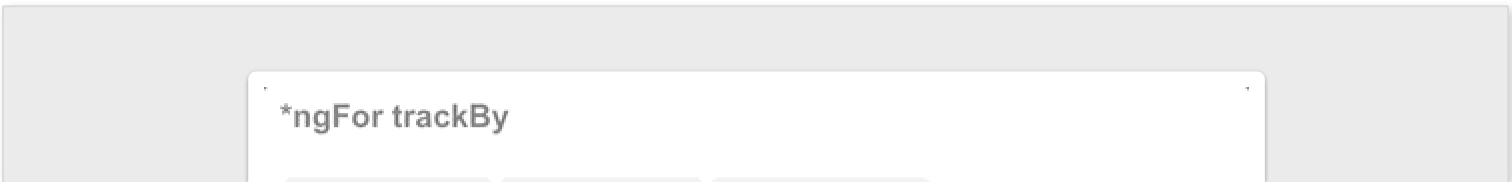
In the microsyntax expression, set `trackBy` to the `trackByItems()` method.

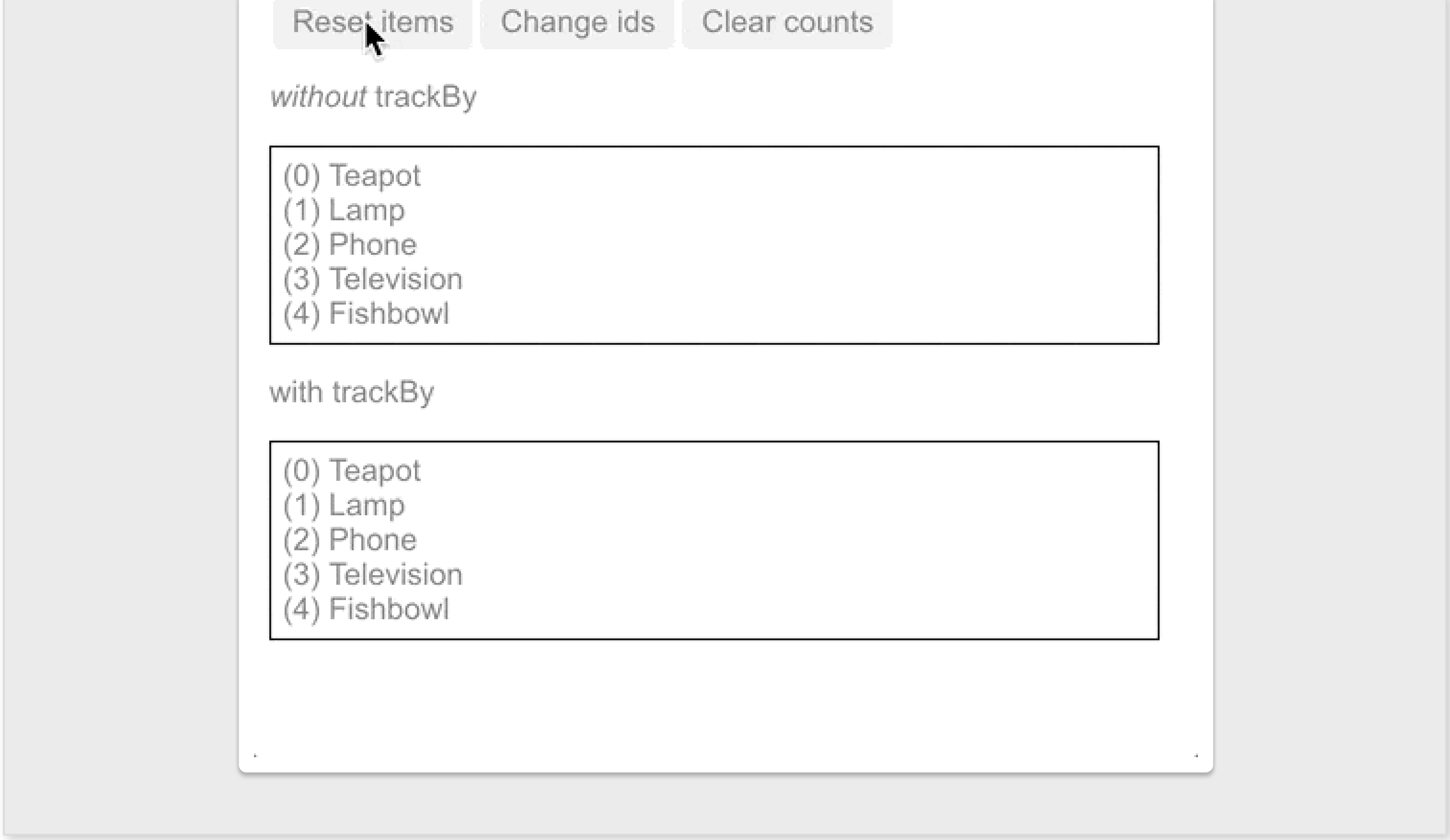
src/app/app.component.html

```html
<div *ngFor="let item of items; trackBy: trackByItems">
  ({{item.id}}) {{item.name}}
</div>
```

Here is an illustration of the `trackBy` effect. "Reset items" creates new items with the same `item.ids`. "Change ids" creates new items with new `item.ids`.

- With no `trackBy`, both buttons trigger complete DOM element replacement.
- With `trackBy`, only changing the `id` triggers element replacement.


*ngFor trackBy



without trackBy

(0) Teapot
(1) Lamp
(2) Phone
(3) Television
(4) Fishbowl

with trackBy

(0) Teapot
(1) Lamp
(2) Phone
(3) Television
(4) Fishbowl

Built-in directives use only public APIs; that is, they do not have special access to any private APIs that other directives can't access.

The `NgSwitch` directives

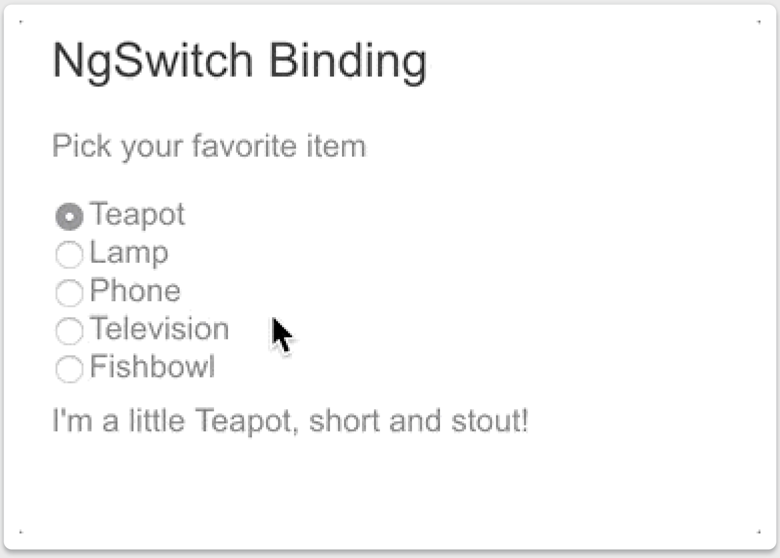
NgSwitch is like the JavaScript `switch` statement. It displays one element from among several possible elements, based on a switch condition. Angular puts only the selected element into the DOM.

`NgSwitch` is actually a set of three, cooperating directives: `NgSwitch`, `NgSwitchCase`, and `NgSwitchDefault` as in the following example.

src/app/app.component.html

```
<div [ngSwitch]="currentItem.feature">
  <app-stout-item    *ngSwitchCase="'stout'"    [item]="currentItem"></app-stout-
item>
  <app-device-item   *ngSwitchCase="'slim'"     [item]="currentItem"></app-device-
item>
  <app-lost-item     *ngSwitchCase="'vintage'"  [item]="currentItem"></app-lost-
item>
  <app-best-item     *ngSwitchCase="'bright'"   [item]="currentItem"></app-best-
item>
<!-- . . . -->
  <app-unknown-item  *ngSwitchDefault           [item]="currentItem"></app-unknown-
```

```
item>
</div>
```



`NgSwitch` is the controller directive. Bind it to an expression that returns the *switch value*, such as `feature`. Though the `feature` value in this example is a string, the switch value can be of any type.

Bind to `[ngSwitch]`. You'll get an error if you try to set `*ngSwitch` because `NgSwitch` is an *attribute* directive, not a *structural* directive. Rather than touching the DOM directly, it changes the behavior of its companion directives.

Bind to `*ngSwitchCase` **and** `*ngSwitchDefault`. The `NgSwitchCase` and `NgSwitchDefault` directives are *structural* directives because they add or remove elements from the DOM.

- `NgSwitchCase` adds its element to the DOM when its bound value equals the switch value and removes its bound value when it doesn't equal the switch value.
- `NgSwitchDefault` adds its element to the DOM when there is no selected `NgSwitchCase`.

The switch directives are particularly useful for adding and removing *component elements*. This example switches among four `item` components defined in the `item-switch.components.ts` file. Each component has an `item` input property which is bound to the `currentItem` of the parent component.

Switch directives work as well with native elements and web components too. For example, you could replace the `<app-best-item>` switch case with the following.

src/app/app.component.html

```
<div *ngSwitchCase="'bright'"> Are you as bright as {{currentItem.name}}?</div>
```