Built-in Directives

Introduction

Angular provides a number of built-in *directives*, which are attributes we add to our HTML elements that give us dynamic behavior. In this chapter, we're going to cover each built-in directive and show you examples of how to use them.

By the end of this chapter you'll be able to use the basic built-in directives that Angular offers.

How To Use This Chapter

Instead of building an app step-by-step, this chapter is a tour of the built-in directives in Angular. Since we're early in the book, we won't explain every detail, but we will provide plenty of example code.

Remember: at any time you can reference the sample code for this chapter to get the complete context.

If you'd like to run the examples in this chapter then see the folder `code/built-in-directives` and run:

```
npm install
npm start
```

And then open http://localhost:4200[43] in your browser.

`NgIf`

The `ngIf` directive is used when you want to display or hide an element based on a condition. The condition is determined by the result of the *expression* that you pass into the directive.

If the result of the expression returns a false value, the element will be removed from the DOM.

Some examples are:

[43] http://localhost:4200

```
<div *ngIf="false"></div>         <!-- never displayed -->
<div *ngIf="a > b"></div>         <!-- displayed if a is more than b -->
<div *ngIf="str == 'yes'"></div> <!-- displayed if str is the string "yes" -->
<div *ngIf="myFunc()"></div>      <!-- displayed if myFunc returns truthy -->
```

Note for AngularJS 1.x Users

If you've used AngularJS 1.x, you may have used the `ngIf` directive before. You can think of the Angular 4 version as a direct substitute.

On the other hand, Angular 4 offers no built-in alternative for `ng-show`. So, if your goal is to just change the CSS visibility of an element, you should look into either the `ngStyle` or the `class` directives, described later in this chapter.

`NgSwitch`

Sometimes you need to render different elements depending on a given condition.

When you run into this situation, you could use `ngIf` several times like this:

```
<div class="container">
  <div *ngIf="myVar == 'A'">Var is A</div>
  <div *ngIf="myVar == 'B'">Var is B</div>
  <div *ngIf="myVar != 'A' && myVar != 'B'">Var is something else</div>
</div>
```

But as you can see, the scenario where `myVar` is neither `A` nor `B` is verbose when all we're trying to express is an `else`.

To illustrate this growth in complexity, say we wanted to handle a new value `C`.

In order to do that, we'd have to not only add the new element with `ngIf`, but also change the last case:

```
<div class="container">
  <div *ngIf="myVar == 'A'">Var is A</div>
  <div *ngIf="myVar == 'B'">Var is B</div>
  <div *ngIf="myVar == 'C'">Var is C</div>
  <div *ngIf="myVar != 'A' && myVar != 'B' && myVar != 'C'">Var is something els\
e</div>
</div>
```

For cases like this, Angular introduces the `ngSwitch` directive.

If you're familiar with the `switch` statement then you'll feel very at home.

The idea behind this directive is the same: allow a single evaluation of an expression, and then display nested elements based on the value that resulted from that evaluation.

Once we have the result then we can:

- Describe the known results, using the `ngSwitchCase` directive
- Handle all the other unknown cases with `ngSwitchDefault`

Let's rewrite our example using this new set of directives:

```
<div class="container" [ngSwitch]="myVar">
  <div *ngSwitchCase="'A'">Var is A</div>
  <div *ngSwitchCase="'B'">Var is B</div>
  <div *ngSwitchDefault>Var is something else</div>
</div>
```

Then if we want to handle the new value `C` we insert a single line:

```
<div class="container" [ngSwitch]="myVar">
  <div *ngSwitchCase="'A'">Var is A</div>
  <div *ngSwitchCase="'B'">Var is B</div>
  <div *ngSwitchCase="'C'">Var is C</div>
  <div *ngSwitchDefault>Var is something else</div>
</div>
```

And we don't have to touch the default (i.e. *fallback*) condition.

Having the `ngSwitchDefault` element is optional. If we leave it out, nothing will be rendered when `myVar` fails to match any of the expected values.

You can also declare the same `*ngSwitchCase` value for different elements, so you're not limited to matching only a single time. Here's an example:

code/built-in-directives/src/app/ng-switch-example/ng-switch-example.component.html

```html
<h4 class="ui horizontal divider header">
  Current choice is {{ choice }}
</h4>

<div class="ui raised segment">
  <ul [ngSwitch]="choice">
    <li *ngSwitchCase="1">First choice</li>
    <li *ngSwitchCase="2">Second choice</li>
    <li *ngSwitchCase="3">Third choice</li>
    <li *ngSwitchCase="4">Fourth choice</li>
    <li *ngSwitchCase="2">Second choice, again</li>
    <li *ngSwitchDefault>Default choice</li>
  </ul>
</div>

<div style="margin-top: 20px;">
  <button class="ui primary button" (click)="nextChoice()">
    Next choice
  </button>
</div>
```

In the example above when the `choice` is `2`, both the second and fifth `li`s will be rendered.

`NgStyle`

With the `NgStyle` directive, you can set a given DOM element CSS properties from Angular expressions.

The simplest way to use this directive is by doing `[style.<cssproperty>]="value"`. For example:

code/built-in-directives/src/app/ng-style-example/ng-style-example.component.html

```html
<div [style.background-color]="'yellow'">
  Uses fixed yellow background
</div>
```

This snippet is using the `NgStyle` directive to set the `background-color` CSS property to the literal string `'yellow'`.

Another way to set fixed values is by using the `NgStyle` attribute and using key value pairs for each property you want to set, like this:

code/built-in-directives/src/app/ng-style-example/ng-style-example.component.html

```
<div [ngStyle]="{color: 'white', 'background-color': 'blue'}">
  Uses fixed white text on blue background
</div>
```

Notice that in the `ng-style` specification we have single quotes around `background-color` but not around color. Why is that? Well, the argument to `ng-style` is a JavaScript object and `color` is a valid key, without quotes. With `background-color`, however, the dash character isn't allowed in an object key, unless it's a string so we have to quote it.

Generally I'd leave out quoting as much as possible in object keys and only quote keys when we have to.

Here we are setting both the `color` and the `background-color` properties.

But the real power of the `NgStyle` directive comes with using dynamic values.

In our example, we are defining two input boxes with an apply settings button:

code/built-in-directives/src/app/ng-style-example/ng-style-example.component.html

```
<div class="ui input">
  <input type="text" name="color" value="{{color}}" #colorinput>
</div>

<div class="ui input">
  <input type="text" name="fontSize" value="{{fontSize}}" #fontinput>
</div>

<button class="ui primary button" (click)="apply(colorinput.value, fontinput.val\
ue)">
  Apply settings
</button>
```

And then using their values to set the CSS properties for three elements.

On the first one, we're setting the font size based on the input value:

code/built-in-directives/src/app/ng-style-example/ng-style-example.component.html

```
<div>
  <span [ngStyle]="{color: 'red'}" [style.font-size.px]="fontSize">
    red text
  </span>
</div>
```

It's important to note that we have to specify units where appropriate. For instance, it isn't valid CSS to set a `font-size` of `12` - we have to specify a unit such as `12px` or `1.2em`. Angular provides a handy syntax for specifying units: here we used the notation `[style.font-size.px]`.

The `.px` suffix indicates that we're setting the `font-size` property value in pixels. You could easily replace that by `[style.font-size.em]` to express the font size in ems or even in percentage using `[style.font-size.%]`.

The other two elements use the `#colorinput` to set the text and background colors:

code/built-in-directives/src/app/ng-style-example/ng-style-example.component.html

```
<h4 class="ui horizontal divider header">
  ngStyle with object property from variable
</h4>

<div>
  <span [ngStyle]="{color: color}">
    {{ color }} text
  </span>
</div>

<h4 class="ui horizontal divider header">
  style from variable
</h4>

<div [style.background-color]="color"
     style="color: white;">
  {{ color }} background
</div>
```

This way, when we click the **Apply settings** button, we call a method that sets the new values:

code/built-in-directives/src/app/ng-style-example/ng-style-example.component.ts

```
  apply(color: string, fontSize: number): void {
    this.color = color;
    this.fontSize = fontSize;
  }
```

And with that, both the color and the font size will be applied to the elements using the `NgStyle` directive.

`NgClass`

The `NgClass` directive, represented by a `ngClass` attribute in your HTML template, allows you to dynamically set and change the CSS classes for a given DOM element.

The first way to use this directive is by passing in an object literal. The object is expected to have the keys as the class names and the values should be a truthy/falsy value to indicate whether the class should be applied or not.

Let's assume we have a CSS class called `bordered` that adds a dashed black border to an element:

code/built-in-directives/src/styles.css

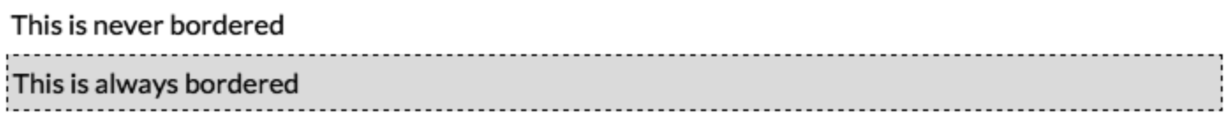
```
.bordered {
  border: 1px dashed black;
  background-color: #eee; }
```

Let's add two `div` elements: one always having the `bordered` class (and therefore always having the border) and another one never having it:

code/built-in-directives/src/app/ng-class-example/ng-class-example.component.html

```
  <div [ngClass]="{bordered: false}">This is never bordered</div>
  <div [ngClass]="{bordered: true}">This is always bordered</div>
```

As expected, this is how those two `div`s would be rendered:


This is never bordered
This is always bordered

Simple class directive usage

Of course, it's a lot more useful to use the `NgClass` directive to make class assignments dynamic.

To make it dynamic we add a variable as the value for the object value, like this:

code/built-in-directives/src/app/ng-class-example/ng-class-example.component.html

```
  <div [ngClass]="{bordered: isBordered}">
   Using object literal. Border {{ isBordered ? "ON" : "OFF" }}
  </div>
```

Alternatively, we can define a `classesObj` object in our component:

code/built-in-directives/src/app/ng-class-example/ng-class-example.component.ts

```
@Component({
  selector: 'app-ng-class-example',
  templateUrl: './ng-class-example.component.html'
})
export class NgClassExampleComponent implements OnInit {
  isBordered: boolean;
  classesObj: Object;
  classList: string[];

  constructor() {
  }

  ngOnInit() {
    this.isBordered = true;
    this.classList = ['blue', 'round'];
    this.toggleBorder();
  }

  toggleBorder(): void {
    this.isBordered = !this.isBordered;
    this.classesObj = {
      bordered: this.isBordered
    };
  }
```

And use the object directly:

code/built-in-directives/src/app/ng-class-example/ng-class-example.component.html

```html
<div [ngClass]="classesObj">
  Using object var. Border {{ classesObj.bordered ? "ON" : "OFF" }}
</div>
```

Again, be careful when you have class names that contains dashes, like `bordered-box`. JavaScript requires that object-literal keys with dashes be quoted like a string, as in:

```html
<div [ngClass]="{'bordered-box': false}">...</div>
```

We can also use a list of class names to specify which class names should be added to the element. For that, we can either pass in an array literal:

code/built-in-directives/src/app/ng-class-example/ng-class-example.component.html

```html
<div class="base" [ngClass]="['blue', 'round']">
  This will always have a blue background and
  round corners
</div>
```

Or assign an array of values to a property in our component:

```javascript
this.classList = ['blue', 'round'];
```

And pass it in:

code/built-in-directives/src/app/ng-class-example/ng-class-example.component.html

```html
<div class="base" [ngClass]="classList">
  This is {{ classList.indexOf('blue') > -1 ? "" : "NOT" }} blue
  and {{ classList.indexOf('round') > -1 ? "" : "NOT" }} round
</div>
```

In this last example, the `[ngClass]` assignment works alongside existing values assigned by the HTML `class` attribute.

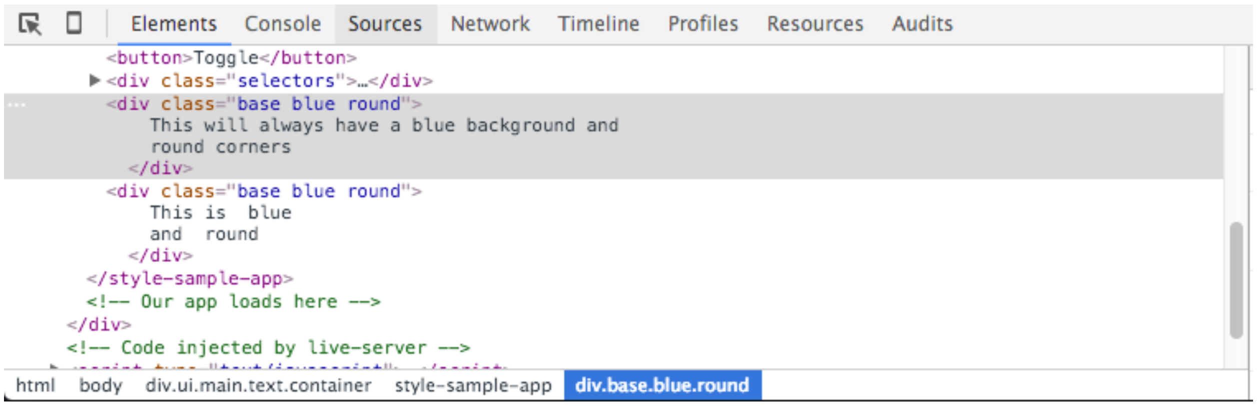
The resulting classes added to the element will always be the set of the classes provided by usual `class` HTML attribute and the result of the evaluation of the `[class]` directive.

In this example:

code/built-in-directives/src/app/ng-class-example/ng-class-example.component.html

```
  <div class="base" [ngClass]="['blue', 'round']">
    This will always have a blue background and
    round corners
  </div>
```

The element will have all three classes: `base` from the `class` HTML attribute and also `blue` and `round` from the `[class]` assignment:


Elements
Console
Sources
Network
Timeline
Profiles
Resources
Audits

```
        <button>Toggle</button>
      ▶ <div class="selectors">…</div>
        <div class="base blue round">
            This will always have a blue background and
            round corners
          </div>
        <div class="base blue round">
            This is  blue
            and  round
          </div>
      </style-sample-app>
      <!-- Our app loads here -->
    </div>
    <!-- Code injected by live-server -->
```

html body div.ui.main.text.container style-sample-app div.base.blue.round

Classes from both the attribute and directive

NgFor

The role of this directive is to **repeat a given DOM element** (or a collection of DOM elements) and pass an element of the array on each iteration.

The syntax is `*ngFor="let item of items"`.

- The `let item` syntax specifies a (template) variable that's receiving each element of the `items` array;
- The `items` is the collection of items from your controller.

To illustrate, we can take a look at the code example. We declare an array of cities on our component controller:

```
this.cities = ['Miami', 'Sao Paulo', 'New York'];
```

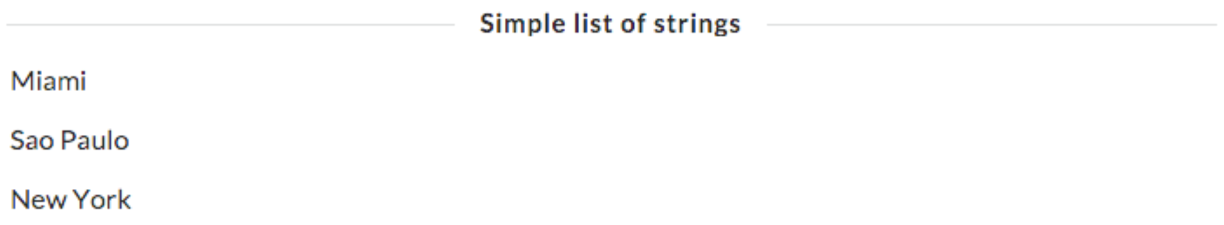
And then, in our template we can have the following HTML snippet:

code/built-in-directives/src/app/ng-for-example/ng-for-example.component.html

```
<h4 class="ui horizontal divider header">
  Simple list of strings
</h4>

<div class="ui list" *ngFor="let c of cities">
  <div class="item">{{ c }}</div>
</div>
```

And it will render each city inside the `div` as you would expect:



Result of the ngFor directive usage

We can also iterate through an array of objects like these:

code/built-in-directives/src/app/ng-for-example/ng-for-example.component.ts

```
    this.people = [
      { name: 'Anderson', age: 35, city: 'Sao Paulo' },
      { name: 'John', age: 12, city: 'Miami' },
      { name: 'Peter', age: 22, city: 'New York' }
    ];
```

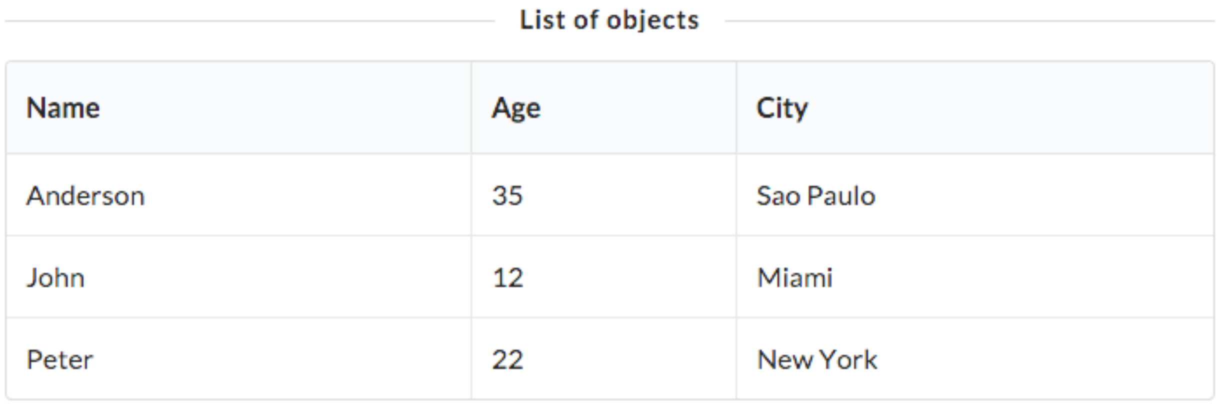
And then render a table based on each row of data:

code/built-in-directives/src/app/ng-for-example/ng-for-example.component.html

```
<h4 class="ui horizontal divider header">
  List of objects
</h4>

<table class="ui celled table">
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
      <th>City</th>
    </tr>
  </thead>
  <tr *ngFor="let p of people">
    <td>{{ p.name }}</td>
    <td>{{ p.age }}</td>
    <td>{{ p.city }}</td>
  </tr>
</table>
```

Getting the following result:



Name	Age	City
Anderson	35	Sao Paulo
John	12	Miami
Peter	22	New York

Rendering array of objects

We can also work with nested arrays. If we wanted to have the same table as above, broken down by city, we could easily declare a new array of objects:

code/built-in-directives/src/app/ng-for-example/ng-for-example.component.ts

```
    this.peopleByCity = [
      {
        city: 'Miami',
        people: [
          { name: 'John', age: 12 },
          { name: 'Angel', age: 22 }
        ]
      },
      {
        city: 'Sao Paulo',
        people: [
          { name: 'Anderson', age: 35 },
          { name: 'Felipe', age: 36 }
      ]
    }
  ];
}
```

And then we could use `NgFor` to render one `h2` for each city:

code/built-in-directives/src/app/ng-for-example/ng-for-example.component.html

```
<div *ngFor="let item of peopleByCity">
  <h2 class="ui header">{{ item.city }}</h2>
```

And use a nested directive to iterate through the people for a given city:

code/built-in-directives/src/app/ng-for-example/ng-for-example.component.html

```
<table class="ui celled table">
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
      <th>City</th>
    </tr>
  </thead>
  <tr *ngFor="let p of people">
    <td>{{ p.name }}</td>
    <td>{{ p.age }}</td>
    <td>{{ p.city }}</td>
  </tr>
</table>
```

Resulting in the following template code:

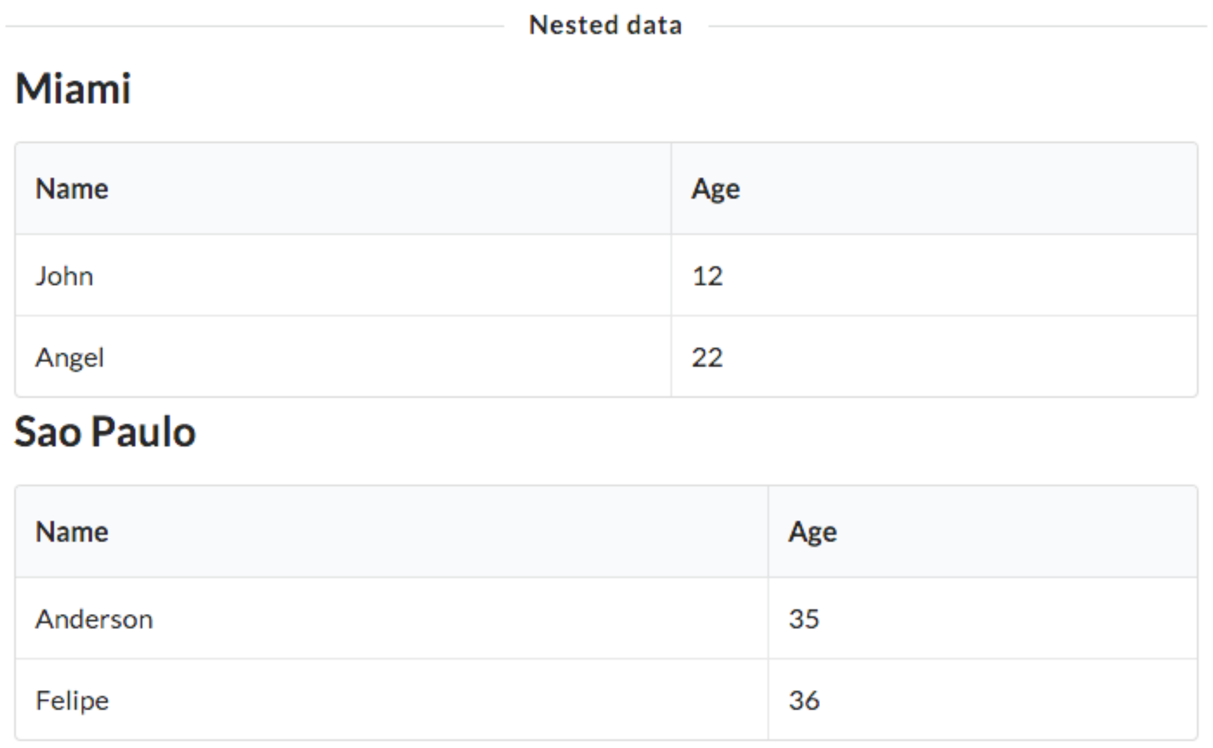
code/built-in-directives/src/app/ng-for-example/ng-for-example.component.html

```html
<h4 class="ui horizontal divider header">
  Nested data
</h4>

<div *ngFor="let item of peopleByCity">
  <h2 class="ui header">{{ item.city }}</h2>

  <table class="ui celled table">
    <thead>
      <tr>
        <th>Name</th>
        <th>Age</th>
      </tr>
    </thead>
    <tr *ngFor="let p of item.people">
      <td>{{ p.name }}</td>
      <td>{{ p.age }}</td>
    </tr>
  </table>
</div>
```

And it would render one table for each city:



Nested data

Miami

Name	Age
John	12
Angel	22

Sao Paulo

Name	Age
Anderson	35
Felipe	36

Rendering nested arrays

Getting an index

There are times that we need the index of each item when we're iterating an array.

We can get the index by appending the syntax `let idx = index` to the value of our `ngFor` directive, separated by a semi-colon. When we do this, ng2 will assign the current index into the variable we provide (in this case, the variable `idx`).

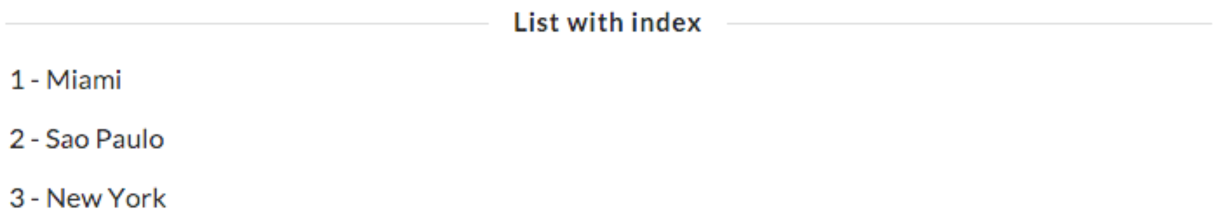
> Note that, like JavaScript, the index is always zero based. So the index for first element is 0, 1 for the second and so on…

Making some changes to our first example, adding the `let num = index` snippet like below:

code/built-in-directives/src/app/ng-for-example/ng-for-example.component.html

```
<div class="ui list" *ngFor="let c of cities; let num = index">
  <div class="item">{{ num+1 }} - {{ c }}</div>
</div>
```

It will add the position of the city before the name, like this:


List with index

1 - Miami

2 - Sao Paulo

3 - New York

Using an index

`NgNonBindable`

We use `ngNonBindable` when we want tell Angular **not** to compile or bind a particular section of our page.

Let's say we want to render the literal text `{{ content }}` in our template. Normally that text will be *bound* to the value of the `content` variable because we're using the `{{ }}` template syntax.

So how can we render the exact text `{{ content }}`? We use the `ngNonBindable` directive.

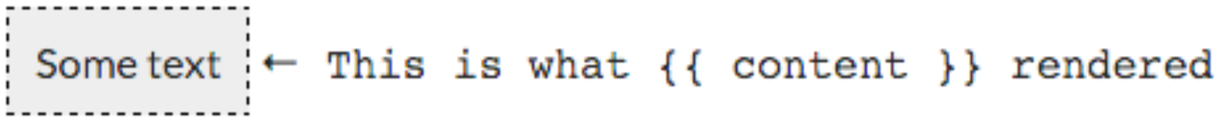
Let's say we want to have a `div` that renders the contents of that `content` variable and right after we want to point that out by outputting *<- this is what {{ content }} rendered* next to the actual value of the variable.

To do that, here's the template we'd have to use:

code/built-in-directives/src/app/ng-non-bindable-example/ng-non-bindable-example.component.html

```html
<div class='ngNonBindableDemo'>
  <span class="bordered">{{ content }}</span>
  <span class="pre" ngNonBindable>
    &larr; This is what {{ content }} rendered
  </span>
</div>
```

And with that `ngNonBindable` attribute, ng2 will not compile within that second `span`'s context, leaving it intact:


Some text ← This is what {{ content }} rendered

Result of using ngNonBindable

Conclusion

Angular has only a few core directives, but we can combine these simple pieces to create dynamic, powerful apps. However, all of these directives help us **output** dynamic data, they don't let us **accept user interaction**.

In the next chapter we'll learn how to let our **user input data using forms**.