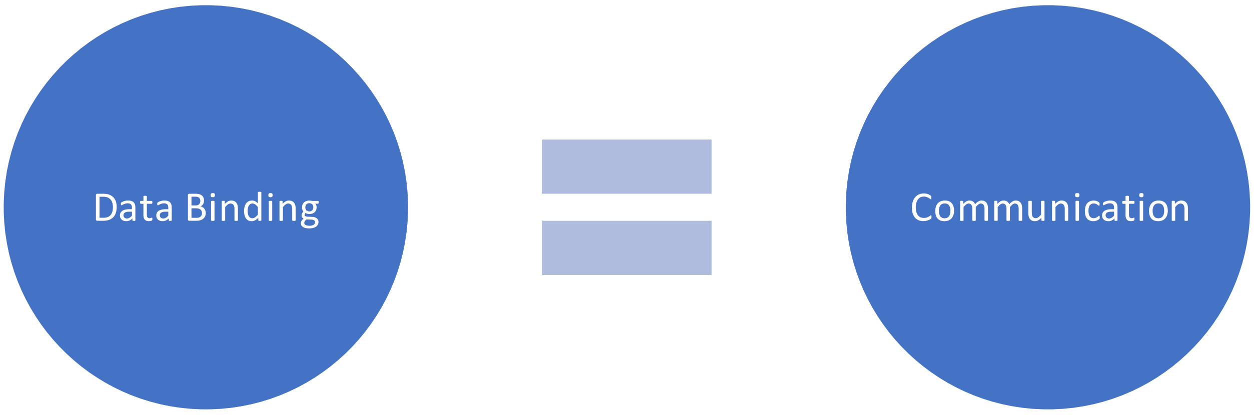
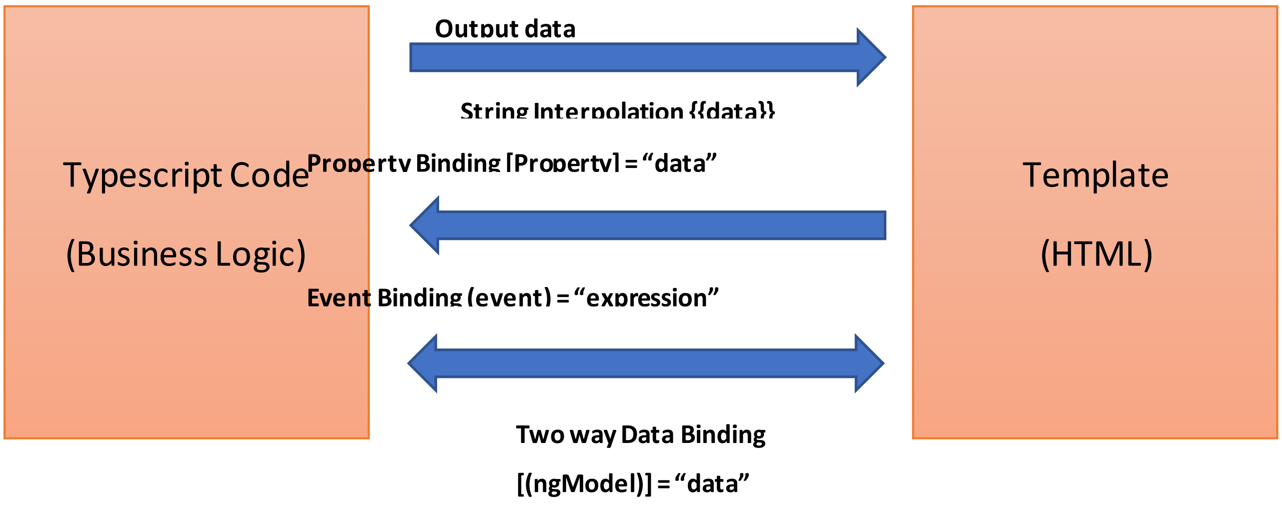
DATA BINDING

In our basic first-app we were working on displaying static data on components. We covered a lot of key aspects of our basic angular and how it starts and how to create and use components.

To output some data dynamically, we use data binding.



Communication between your typescript code of your business logic and the template what the user sees is called Data Binding.


Output data
String Interpolation {{data}}
Property Binding [Property] = "data"
Typescript Code
(Business Logic)
Template
(HTML)
Event Binding (event) = "expression"
Two way Data Binding
[(ngModel)] = "data"

Whenever you need to communicate properties (variables, objects, arrays, etc..) from the component class to the template, you can use String Interpolation.

The format for defining interpolation in a template is: `{{ propertyName }}`

The easiest way to display a component property is to bind the property name through interpolation. With interpolation, you put the property name in the view template, enclosed in double curly braces: `{{data}}`.

app.component.html

{{data}}

Angular automatically pulls the value of the `data` property from the component and inserts those values into the browser. Angular updates the display when these properties change.

Let's say for some reason, that we want to use the component class to control the value of our ADD ITEM button text. We can do this through property binding.

Open /app.component.ts and add the following property

```
  itemCount: number = 4;

  btnText: string = 'Add an Item';    // Add this line
```

Then, open /app.component.html and set the value property through property binding, to our btnText property:

```
  <!-- From: -->

  <input type="submit" class="btn" value="Add Item">


  <!-- To: -->

  <input type="submit" class="btn" [value]="btnText">
```

Property binding in Angular 5 is one-way, in that communication goes from the component class to the template.

What if we wanted to use our input textfield to both retrieve and set its value from a component class? We can use what's called **ngModel** to create that 2-way data binding.

First, we have to import the **FormsModule** in order to have access to NGMODULE.

Open up /src/app/app.module.ts and at the top within the imports, paste:

```
    import { FormsModule } from '@angular/forms';

    // Other imports removed for brevity

    @NgModule({

      ...

      imports: [

        BrowserModule,

        AppRoutingModule,

        FormsModule              // Add the FormsModule here
```

```
              ],
```

Revisit /app.component.ts and add a third property:

```
        itemCount: number = 4;

          btnText: string = 'Add an Item';

          goalText: string = 'My first life goal';    // Add this
```

In the template file, add this to create a 2-way data binding:

```
    <!-- From: -->

    <input type="text" class="txt" name="item" placeholder="Life goal..">


    <!-- To: -->

    <input type="text" class="txt" name="item" placeholder="Life goal.."
[(ngModel)]="goalText">


    <br><span>{{ goalText }}</span><br>
```

Now that we know how to capture user input, how can we make our ADD AN ITEM button actually work and save the user-submitted data somewhere?

That's where event binding comes in. We can use event binding to capture a variety of user-initiated events to initiate logic in our component class.

Let's define a click event on our Add an Item button:

```
        <!-- From: -->

        <input type="submit" class="btn" value="{{ btnText }}">


        <!-- To: -->

        <input type="submit" class="btn" value="{{ btnText }}"
    (click)="addItem()">
```

Note: We're not even passing in the value of the input field, because we've set it up to work with 2-way data binding.

In the component class, let's create the **addItem()** method and also make some other adjustments:

```
        export class AppComponent implements OnInit {
```

```
  itemCount: number = 4;

  btnText: string = 'Add an Item';

  goalText: string = 'My first life goal';

  goals = [];


  constructor() { }


  ngOnInit() {

    this.itemCount = this.goals.length;

  }


  addItem() {                                   //←-added function here

    this.goals.push(this.goalText);

    this.goalText = '';

    this.itemCount = this.goals.length;

  }


}
```

ADDING DATA TO THE COMPONENT

Right now, our component renders a static template, which means our component isn't very interesting.

Let's imagine that we have an app which will show a list of users and we want to show their names. Before we render the whole list, we first need to render an individual user. So, let's create a new component that will show a user's name. To do this, we will use the `ng generate` command again:

```
> ng generate component user-item
```

Remember that in order to see a component we've created, we need to add it to a template.

Let's add our `app-user-item` tag to `app.component.html` so that we can see our changes as we make them. Modify `app.component.html` to look like this

```
<h1>

{{title}}

    <app-user-item></app-user-item>
    </h1>
```

Then refresh the page and confirm that you see the `user-item works!` text on the page. We want our `UserItemComponent` to show the name of a particular user.

Let's introduce `name` as a new property of our component. By having a `name` property, we will be able to reuse this component for different users (but keep the same markup, logic, and styles).

In order to add a name, we'll introduce a property on the `UserItemComponent` class to declare it has a local variable named `name`.

```
export class UserItemComponent implements OnInit {
    name: string; // <-- added name property
    constructor () {
    this.name = 'Felipe'; // set the name
    }
    ngOnInit() {
    }
}
```

1. NAME PROPERTY

On the `UserItemComponent` class we added a property. Notice that the syntax is new relative to ES5 JavaScript. When we write `name: string;` it means that we're declaring the `name` property to be of type `string`.

Being able to assign a type to a variable is what gives TypeScript it's name. By setting the type of this property to `string`, the compiler ensures that `name` variable is a `string` and it will throw an error if we try to assign, say, a `number` to this property.

This syntax is also the way TypeScript defines instance properties. By putting `name: string` in our code like this, we're giving every instance of `UserItemComponent` a property `name`.

2. A CONSTRUCTOR

On the `UserItemComponent` class we defined a constructor, i.e. a function that is called when we create new instances of this class.

In our constructor we can assign our `name` property by using `this.name` When we write:

```
constructor() {
  this.name = 'Felipe'; // set the name
}
```

We're saying that whenever a new `UserItemComponent` is created, set the name to `'Felipe'`.

RENDERING THE TEMPLATE

When we have a property on a component, we can show that value in our template by using two curly brackets `{{ }}` to display the value of the variable in our template. For instance:

/first-app/angular-hello-world/src/app/user-item/user-item.component.html

```
<p>
  Hello {{ name }}
</p>
```

On the `template` notice that we added a new syntax: `{{ name }}`. The brackets are called template tags (or sometimes mustache tags).

Whatever is between the template tags will be expanded as an expression. Here, because the `template` is bound to our Component, the `name` will expand to the value of `this.name` i.e. `'Felipe'`.

WORKING WITH ARRAYS

Now we are able to say "Hello" to a single name, but what if we want to say "Hello" to a collection of names

In Angular we can iterate over a list of objects in our template using the syntax `*ngFor`. The idea is that we want to repeat the same markup for a collection of objects.

Let's create a new component that will render a list of users. We start by generating a new component:

```
> ng generate component user-list
```

And let's replace our <app-user-item> tag with <app-user-list> in our app.component.html file:

/first-app/angular-hello-world/src/app/app.component.html

```
<h1>
    {{title}}
    <app-user-list></app-user-list>
</h1>
```

In the same way we added a `name` property to our `UserItemComponent`, let's add a `names` property to this `UserListComponent`.

However, instead of storing only a single string, let's set the type of this property to an array of strings. An array is notated by the `[]` after the type, and the code looks like this:

code/first-app/angular-hello-world/src/app/user-list/user-list.component.ts

```
export class UserListComponent implements OnInit {
    names: string[];
    constructor() {
    this.names = ['Ari', 'Carlos', 'Felipe', 'Nate'];
    }
    ngOnInit() {
    }
}
```

The first change to point out is the new `string[]` property on our `UserListComponent` class. This syntax means that `names` is typed as an `Array` of `string`s. Another way to write this would be `Array<string>`.

We changed our constructor to set the value of `this.names` to `['Ari', 'Carlos', 'Felipe', 'Nate']`.

Now we can update our template to render this list of names. To do this, we will use `*ngFor`, which will

- iterate over a list of items and
- generate a new tag for each one.

Here's what our new template will look like:

/first-app/angular-hello-world/src/app/user-list/user-list.component.html

```
<ul>
  <li *ngFor="let name of names">Hello {{ name }}</li>
</ul>
```

We updated the template with one `ul` and one `li` with a new `*ngFor="let name of names"` attribute. The `*` character and `let` syntax can be a little overwhelming at first, so let's break it down:

The `*ngFor` syntax says we want to use the `NgFor` directive on this attribute. You can think of `NgFor` akin to a `for` loop; the idea is that we're creating a new DOM element for every item in a collection.

The value states: `"let name of names"`. `names` is our array of names as specified on the `UserListComponent` object. `let name` is called a reference. When we say `"let name of names"` we're saying loop over each element in `names` and assign each one to a local variable called `name`.

The `NgFor` directive will render one `li` tag for each entry found on the `names` array and declare a local variable `name` to hold the current item being iterated. This new variable will then be replaced inside the `Hello {{ name }}` snippet.

`NgFor` repeats the element that the `ngFor` is called. That is, we put it on the `li` tag and not the `ul` tag because we want to repeat the list element (`li`) and not the list itself (`ul`).

Note that the capitaliz ation here isn't a typo: `NgFor` is the capitalization of the class that implements the logic and `ngFor` is the "selector" for the attribute we want to use

BUILT-IN DIRECTIVES

Directives are instructions to the DOM. “Components” are such kind of instructions in the DOM. Once we place our selector of our component somewhere in out template, we are instructing angular to add content of our component template and business logic in our typescript code in that place where we use the selector.

“Components” are directives with templates, But there are also directives without template. We can create custom directives as well as can use built in directives.

Angular provides a number of built-in directives, which are attributes we add to our HTML elements that give us dynamic behavior.

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

```
<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 -->
```

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
else
```

```
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:

```
<h4>

  Current choice is {{ choice }}

</h4>

<div>

    <ul [ngSwitch]="choice">

      <li *ngSwitchCase="1">First choice</li>
```

```
            <li *ngSwitchCase="2">Second choice</li> 9

            <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:

```
<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:

```
<div [ngStyle]="{color: 'white', 'background-color': 'blue'}">

  Uses fixed white text on blue background

</div>
```

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

```
<div>
```

```
    <input type="text" name="color" value="{{color}}" #colorinput>
  </div>
  <div>
    <input type="text" name="fontSize" value="{{fontSize}}" #fontinput>
  </div>
  <button (click)="apply(colorinput.value, fontinput.value)">
          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:

```
  <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:

```
  <h4>
          ngStyle with object property from variable
  </h4>
  <div>
      <span [ngStyle]="{color: color}">
        {{ color }} text
      </span>
  </div>
  <h4>
          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:

```
apply(color: string, fontSize: number): void {

    this.color = color;

    this.fontSize = fontSize;

}
```

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>
```

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

```
<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:

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

```
<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:

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

And pass it in:

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

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:

```
<h4>
        Simple list of strings
</h4>
```

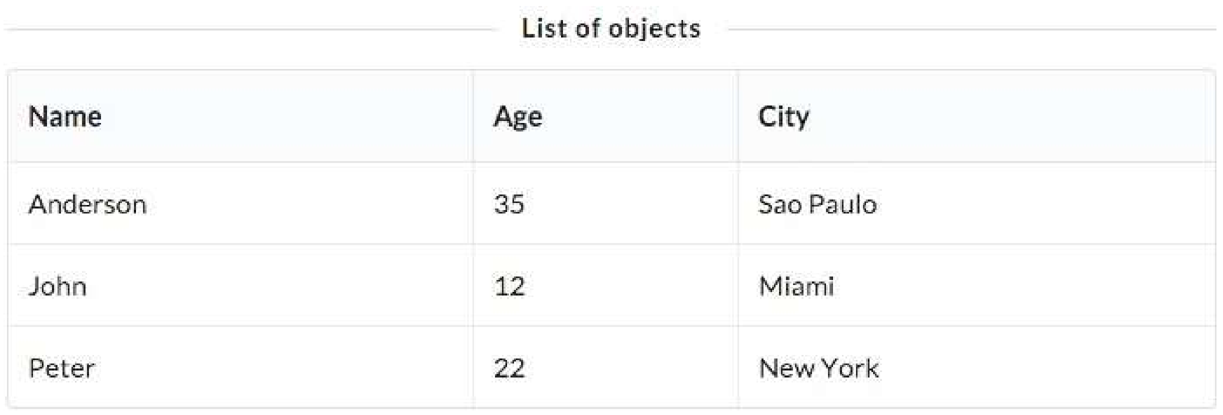
```
<div *ngFor="let c of cities">
  {{ c }}
</div>
```

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

```
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

```
<h4>
  List of objects
</h4>
<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>
```



List of objects

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

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

```
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 }
                       ]
               }
         ];


<h4>
  Nested data
</h4>
<div *ngFor="let item of peopleByCity">
         <h2>{{ 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>
```

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, ng 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.

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

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

```
<div class='ngNonBindableDemo'>

<span class="bordered">{{ content }}</span>

<span class="pre" ngNonBindable>

  &larr; This is what {{ content }} rendered

</span>

</div>
```