AngularJS Views And Directives

By Jakob Jenkov

Connect with me:

Rate article:

+1 52

Share article:

Share Tweet 44

Table of Contents

AngularJS Views and Directives Introduction

In the first text of this tutorial you saw how AngularJS splits an application into views, controllers and models (MVC). This text will dive deeper into how to create AngularJS views.

Before we start, let me first set up a simple AngularJS application which you can use to play around with the examples in this text:

```
<!DOCTYPE html>
<html>
    <head>
        <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.5/angular.min.js
    </head>

    <body ng-app="myapp">

          <div ng-controller="MyController" >
                <span></span>
          </div>

          <script>
                angular.module("myapp", [])
                .controller("MyController", function($scope) {
                  //empty controller function
                });
          </script>

    </body>
</html>
```

AngularJS Directives

AngularJS views mix data from the model into an HTML template. You use AngularJS `directives` to tell AnguluarJS how to mix the data into the HTML template. This text will cover the most commonly used AngularJS directives.

Interpolation Directive

The interpolation directive is one of the most fundamental directives in AngujarJS. The interpolation directive inserts the result of an expression into the HTML template. You mark where to insert the expression using the `{{ }}` notation. Here is an example:

```
<div ng-controller="MyController" >
    <span><b>{{myData.text}}</b></span>
</div>
```

The HTML template is contained within the `div` element with the `ng-controller` attribute. Inside the HTML template is a `span` element, and inside this is an interpolation directive. This directive instructs AngularJS to insert the data value `myData.text` at the given location.

The interpolation directive can also insert data returned from functions of the model object. Here is an example:

```
<div ng-controller="MyController" >
    <span>{{myData.textf()}}</span>
</div>

<script>
  angular.module("myapp", [])
  .controller("MyController", function($scope) {
    $scope.myData = {};
    $scope.myData.textf = function() { return "A text from a function"; };
  });
</script>
```

In this example the interpolation directive `{{myData.textf()}}` will call the `myData.textf()` function on the `$scope` model object, and insert the text returned from that function into the HTML template.

The `textf()` function is inserted into the `$scope.myData` object inside the controller function, as you can see in the example.

ng-bind Directive

The `ng-bind` directive is an alternative to the interpolation directive. You use it by inserting an `ng-bind` attribute into the HTML element you want AngularJS to insert data into. Here is an example:

```
<div ng-controller="MyController" >
  <span ng-bind="myData.textf()"></span>
</div>
```

This will insert the data returned from the `myData.text()` function into the body of the `span` element. Notice how the `{{ }}` are not necessary around the expression inside the `ng-bind` attribute.

Escaping HTML From The Model

If the data obtained from the model contains HTML elements, these are escaped before being inserted into the HTML template. The escaping means that the HTML is displayed as text, and not as HTML.

This is done to prevent HTML injection attacks. For instance, in a chat application somebody might insert a `<script>` element with JavaScript into a chat message. If this element was not escaped, anyone seeing the chat message might have the `<script>` element executed. With the HTML escaping the `<script>` element will just be displayed as text.

You can disable the HTML escaping by using the `ng-bind-html-unsafe` directive, like this:

```
<div ng-controller="MyController" >
  <span ng-bind-html-unsafe="myData.textf()"></span>
</div>
```

You should be really careful when disabling HTML escaping. Make sure that no HTML is displayed which is not trusted.

Conditional Rendering

AngularJS can show or hide HTML depending on the state of data in the model. You do so using a set of AngularJS directives which are created specifically for that purpose. I will cover these directives in the following sections.

ng-show + ng-hide Directives

The `ng-show` and `ng-hide` directives are used to show or hide an HTML element depending on data in the model. These two directives do the same thing, but are each other's opposites. Here are two examples:

```
<div ng-controller="MyController" >
    <span ng-show="myData.showIt"></span>
    <span ng-hide="myData.showIt"></span>
</div>

<script>
  angular.module("myapp", [])
  .controller("MyController", function($scope) {
    $scope.myData = {};
    $scope.myData.showIt = true;
  });
</script>
```

This example creates two `span` elements. One has an `ng-show` directive and the other has an `ng-hide` directive. Both directives look at the `myData.showIt` boolean variable to determine if they should show or hide the `span` element. The `ng-show` directive will show the element if the model value is true, and hide the element if the model value is false. The `ng-hide` directive will do the opposite: Hide the `span` element if the model value is true, and show it if the model value is false.

Notice how the controller function sets the `myData.showIt` to `true`. This means that the example above will show the first span element and hide the second.

The HTML elements (`span` elements in this case) are hidden using the CSS property `display: none;`. That means, that the HTML elements are still present in the DOM. They are just not visible.

ng-switch Directive

The `ng-switch` directive is used if you want to add or remove HTML elements from the DOM based on data in the model. Here is an example:

```
<div ng-controller="MyController" >
    <div ng-switch on="myData.switch">
        <div ng-switch-when="1">Shown when switch is 1</div>
        <div ng-switch-when="2">Shown when switch is 2</div>
        <div ng-switch-default>Shown when switch is anything else than 1 and 2</div>
    </div>
</div>

<script>
    angular.module("myapp", [])
    .controller("MyController", function($scope) {
      $scope.myData = {};
      $scope.myData.switch = 3;
    });
</script>
```

This example contains a `div` element with an `ng-switch` attribute and an `on` attribute. The `on` attribute tells which data in the model to switch on.

Inside the `div` element are three nested `div` elements. The first two nested `div` elements contains an `ng-switch-when` attribute. The value of this attribute tells what value the model data referenced in the `on` attribute of the parent `div` should have, for the nested `div` to be visible. In this example the first nested `div` is visible when `myData.switch` is 1, and the second nested `div` is visible when `myData.switch` is 2.

The `third` nested `div` has an `ng-switch-default` attribute. If no of the other `ng-switch-when` directives are matched, then the `div` with the `ng-switch-default` attribute is shown.

In the example above the controller function sets `myData.switch` to 3. That means that the nested `div` with the `ng-switch-default` attribute will be shown. The two other nested `div` elements will be removed from the DOM completely.

ng-if Directive

The `ng-if` directive can include / remove HTML elements from the DOM, just like the `ng-switch` directive, but it has a simpler syntax. Here is an example:

```
<div ng-controller="MyController" >
    <div ng-if="myData.showIt">ng-if Show it</div>
</div>

<script>
    angular.module("myapp", [])
    .controller("MyController", function($scope) {
        $scope.myData = {};
        $scope.myData.showIt = true;
    });
</script>
```

The main difference between `ng-if` and `ng-show` + `ng-hide` is that `ng-if` removes the HTML element completely from the DOM, whereas the `ng-show` + `ng-hide` just applies the CSS property `display: none;` to the elements.

The ng-include directive can be used to include HTML fragments from other files into the view's HTML template. Here is an example:

```
<div ng-controller="MyController" >
    <div ng-include="'angular-included-fragment.html'"></div>
</div>
```

This example includes the file angular-included-fragment.html into the HTML template inside the div having the ng-include attribute. Notice how the file name is quoted (single quotes).

You can include HTML fragments based on conditions. For instance, you can choose between two files like this:

```
<div ng-controller="MyController" >
    <div ng-include="myData.showIt &&
                        'fragment-1.html' ||
                        'fragment-2.html'"></div>
</div>

<script>
    angular.module("myapp", [])
    .controller("MyController", function($scope) {
        $scope.myData = {};
        $scope.myData.showIt = true;
    });
</script>
```

This example will include fragment-1.html if myData.showIt is true, and fragment-2.html if myData.showIt is false.

ng-repeat Directive

The ng-repeat directive is used to iterate over a collection of items and generate HTML from it. After the initial generation the ng-repeat monitors the items used to generate the HTML for changes. If an item changes, the ng-repeat directive may update the HTML accordingly. This includes reordering and removing DOM nodes.

Here is a simple ng-repeat example:

```
<ol>
   <li ng-repeat="theItem in myData.items">{{theItem.text}}</li>
</ol>

<script>
    angular.module("myapp", [])
    .controller("MyController", function($scope) {
        $scope.myData = {};
        $scope.myData.items = [ {text : "one"}, {text : "two"}, {text : "three"} ];
    });
</script>
```

This example will create an li element for each item in the myData.items array.

You can also iterate over collections returned from a function call. Here is an example:

```
<ol>
   <li ng-repeat="theItem in myData.getItems()">{{theItem.text}}</li>
</ol>

<script>
    angular.module("myapp", [])
    .controller("MyController", function($scope) {
        $scope.myData = {};
        $scope.myData.items = [ {text : "one"}, {text : "two"}, {text : "three"} ];
        $scope.myData.getItems = function() { return this.items; };
    });
</script>
```

And you can iterate over the properties of a JavaScript object using a slightly different syntax:

```
<ol>
   <li ng-repeat="(name, value) in myData.myObject">{{name}} = {{value}}</li>
</ol>

<script>
    angular.module("myapp", [])
    .controller("MyController", function($scope) {
        $scope.myData = {};
        $scope.myData.myObject = { var1 : "val1", var2 : "val3", var3 : "val3"};
    });
</script>
```

Notice the `(name, value)` part of the `ng-repeat` directive. That signals to AngularJS to iterate over the properties of an object. The `name` parameter will be bound to the property name, and the `value` parameter will be bound to the property value. The `name` and `value` parameters can be output to the HTML template just like any other JavaScript variable or object property, as you can see from the HTML template above.

Special ng-repeat Variables

The `ng-repeat` directive defines a set of special variables which you can use when iterating the collection. These variables are:

- $index
- $first
- $middle
- $last

The `$index` variable contains the index of the element being iterated.

The `$first`, `$middle` and `$last` contain a boolean value depending on whether the current item is the first, middle or last element in the collection being iterated. An item is "middle" if it is not first nor last. You can use these variables to generate different HTML using e.g. the `ng-show` / `ng-hide`, `ng-switch`, `ng-if` and `ng-include` directives described earlier.

Repeating Multiple Elements

So far you have only seen how to repeat a single HTML element using `ng-repeat`. In case you want to repeat more than one HTML element you would have to nest those elements inside a container element, and have the container element have the `ng-repeat` element, like this:

```
<div ng-repeat="(name, value) in myData.myObject">
   <div>{{name}}</li>
   <div>{{value}}</li>
</div>
```

Wrapping the element to be repeated in a root element may not always be possible though. Therefore AngularJS has the `ng-repeat-start` and `ng-repeat-end` directives which mark which element to start and end the repetition with. Here is an example:

```
<ol>
    <li ng-repeat-start="(name, value) in myData.myObject">{{name}}</li>
    <li ng-repeat-end>{{value}}</li>
</ol>
```

This example will repeat both of the `li` elements for each property in `myData.myObject`.

Filtering

Some of the directives covered above support filtering. This section will explain how filtering works.

The `ng-repeat` directive can accept a filter like this:

```
<div ng-repeat="item in myData.items | filter: itemFilter"></div>
```

Notice the `| filter: itemFilter` part of the declaration above. That part is the filter definition. The `| filter:` part tells AngularJS to apply a filter to the `myData.items` array. The `itemFilter` is the name of the filter function. This function has to be present on the `$scope` object, and it has to return either true or false. If the filter function returns true, then the element from the array is used by the `ng-repeat` directive. If the filter function returns false, the element is ignored. Here is an example:

```
<script>
  angular.module("myapp", [])
    .controller("MyController", function($scope) {
      $scope.myData = {};
      $scope.myData.items  = [ {text : "one"}, {text : "two"}, {text : "three"}, {text : "

      $scope.itemFilter = function(item) {
        if(item.text == "two") return false;
          return true;
        }
      }
    });
</script>
```

Formatting Filters

AngularJS comes with a set of built-in formatting filters which can be used in conjunction with the interpolation directive, and with `ng-bind`. Here is a list of the formatting filters:

Filter	Description
date	Formats the variable as a date according to the given date format
currency	Formats the variable as a number with a currency symbol
number	Formats the variable as a number
lowercase	Converts the variable to lowercase
uppercase	Converts the variable to uppercase
json	Converts the variable to a JSON string

Here is a date filter example:

```
<span>{{myData.theDate | date: 'dd-MM-yyyy'}}</span>
```

This example shows the `date` filter which can format a JavaScript date object according to the date format pattern given after the `| date:` part. It is the `myData.theDate` property that will be formatted as a date. Thus, it has to be a JavaScript date object.

Here is a number filter example:

```
<span>{{myData.theNumber | number: 2}}</span>
```

This example formats the `myData.theNumber` variable as a number with 2 decimal places.

Here are an lowercase and uppercase filter example:

```
<span>{{myData.mixedCaseText | lowercase}}</span>
<span>{{myData.mixedCaseText | uppercase}}</span>
```

Array Filters

AngularJS also contains a set of array filters which filters or transforms arrays. These filters are:

Array Filters:

Filter	Description
`limitTo`	Limits the array to the given size, beginning from some index in the array. The `limitTo` filter also works on strings.
`filter`	A general purpose filter.
`orderBy`	Sorts the array based on provided criteria.

Here is a `limitTo` example:

```
<span>{{myData.theText | limitTo: 3}}</span>
```

This limits the `$scope` `myData.theText` variable to a length of 3 characters. If this filter had been applied to an array, the array would have been limited to 3 elements.

The `filter` filter is a special filter which can do a lot of different things. In its simplest form it simply calls a function on the `$scope` object. This function must return `true` or `false`. True is returned if the filter accepts the value passed to it. False is returned if the filter cannot accept the value. If the filter cannot accept the value, the value is not included in the array resulting from the filtering. Here is an example:

```
<ol>
    <li ng-repeat="item in myData.items | filter:filterArray">
        {{item.text}} : {{$first}}, {{$middle}}, {{$last}}
    </li>
</ol>
<script>
    angular.module("myapp", [])
        .controller("MyController", function($scope) {
            $scope.myData = {};
            $scope.myData.items    =
                [ {text : "one"}, {text : "two"}, {text : "three"}, {text : "four"} ];

            $scope.filterArray = function(item) {
                if(item.text == "two") return false;
                return true;
            }
        } );
</script>
```

This example calls the `filterArray()` function which filters out items which has a `text` property with the value `two`.

Here is an `orderBy` example:

```
<ol>
    <li ng-repeat="item in myData.items | orderBy:sortField:reverse">
        {{item.text}} : {{$first}}, {{$middle}}, {{$last}}
    </li>
</ol>

<script>
    angular.module("myapp", [])
            .controller("MyController", function($scope) {
                $scope.myData = {};
                $scope.myData.items     = [ {text : "one"}, {text : "two"}, {text : "three"
                $scope.sortField = "text";
                $scope.reverse    = true;
            } );
</script>
```

The `orderBy` filter takes a `$scope` variable as parameter. In this example that variable is named `sortField`. The value of this variable is the name of the property on the sorted data objects which is used to sort the data objects. In this example the `sortField` property is set to `text` which means that the data object's `text` property is used to sort the data objects.

The `orderBy` filter can also take a second `$scope` variable as parameter. In this example that variable is named `reverse`. The value of this variable determines if the data objects are to be sorted in their natural order, or the reverse order of that. In this case the `reverse` variable is set to `true`, meaning the data objects will be sorted in reverse order.

Chaining Filters

It is possible to chain filters by simply putting more filters after each other in the filter section. When chaining filters, the output of one filter is used as input for the next filter in the chain. Here is an example:

```
<span>{{myData.theText | limitTo: 5 | uppercase}}</span>
```

This example first limits the string `myData.theText` to 5 characters using the `limitTo` filter, and the converts the 5 characters to uppercase using the `uppercase` filter.

Assigning Filter Output To Variables

It is possible to assign the output of a filter to a temporary variable which you can then refer to later in your view. Here is an example:

```
<ol>
    <li ng-repeat="item in filteredItems = ( myData.items | filter:filterArray) ">
        {{item.text}} : {{$first}}, {{$middle}}, {{$last}}
    </li>
</ol>
<div>{{filteredItems.length}}</div>
```

This example assigns the output of the filtering to the `filteredItems` variable. The example then refers to this variable inside the `{{ }}` directive under the `ol` element.

Implementing Custom Filters

You can implement your own filters in case the AngularJS filters do not suit your needs. Here is an example:

```
<div>Filtered: {{myData.text | myFilter}}</div>


<script>
    var module = angular.module("myapp", []);

    module.filter('myFilter', function() {

        return function(stringValue) {
            return stringValue.substring(0,3);
        };
    });
</script>
```

This example registers a filter with AngularJS which can filter strings. The filter returns the first 3 characters of the string. The filter is registered with the name `myFilter`. It is this name you will have to use when referencing the filter, as you can see in the beginning of the filter.

If your filter needs more input parameters, add more parameters to the filter function, and add the parameters after the filter name and a : when referencing it. Here is an example:

```
<div>Filtered: {{myData.text | myFilter<b>:2:5</b>}}</div>

<script>
    var module = angular.module("myapp", []);

    module.filter('myFilter', function() {

        return function(stringValue, <b>startIndex</b>, <b>endIndex</b>) {
            return stringValue.substring(parseInt(startIndex), parseInt(endIndex));
        };
    });
</script>
```

Notice how the filter reference (`| myfilter:2:5`) now has two values after the filter name, each value separated by a colon. These two values are passed to the filter as parameters. Notice also how the filter function now takes two extra parameters named `startIndex` and `endIndex`. These two parameters are used to determine which part of the string to return as substring from the filter.