# AngularJS Custom Directives

Jakob Jenkov

Last update: 2015-01-03

## Custom Directive Introduction

AngularJS directives are what controls the rendering of the HTML inside an AngularJS application. Examples of directives are the interpolation directive ( `{{ }}` ), the `ng-repeat` directive and `ng-if` directive.

It is possible to implement your own directives too. This is what AngularJS refers to as "teaching HTML new tricks". This text will show you how to do that.

## Directive Types

You can implement the following types of directives:

- Element directives
- Attribute directives
- CSS class directives
- Comment directives

Of these, AngularJS recommends that you try to use element and attribute directives, and leave the CSS class and comment directives (unless absolutely necessary).

The type of a directive determines how the directive is activated. An element directive is activated when AngularJS finds a matching HTML element in the HTML template. An attribute directive is activated when AngularJS finds a matching HTML element attribute. A CSS class directive is activated when AngularJS finds a matching CSS Class. And, a comment directive is activated when AngularJS finds a matching HTML comment.

## A Basic Directive

You register a directive with a module. Here is an example of how that looks:

```
myapp = angular.module("myapp", []);

myapp.directive('div', function() {
    var directive = {};

    directive.restrict = 'E'; /* restrict this directive to elements */

    directive.template = "My first directive: {{textToInsert}}";

    return directive;
});
```

Notice the call to the `directive()` function on the module. When you call this function you can register a new directive. The first parameter to the `directive()` function call is the name of the directive to register. This name is what you use in your HTML templates when you want to activate the directive. In this example I have used the name '`div`' which means that the directive is activated every time an HTML element named `div` is found in the HTML template.

The second parameter passed to the `directive` function is a factory function. This function should return a directive definition when invoked. AngularJS will invoke this function to obtain a JavaScript object which contains the definition of the directive. If you look inside the function in the above example you will see that it does indeed return a JavaScript object.

The JavaScript object returned from the factory function has two properties: A `restrict` field and a `template` field.

The `restrict` field is used to set if the directive should be activated by a matching HTML element, or an element attribute. By setting `restrict` to `E` you specify that only HTML elements named `div` should activate the directive. By setting `restrict` to `A` you specify that only HTML attributes named `div` should activate the directive. You can also use a value of `AE` which will match both HTML element names and attribute names.

The `template` field is an HTML template that will replace the content of the matched `div` element. It will work as if the content of the matched `div` element had not been there, and instead this HTML template had been located in the same place.

Imagine that your HTML page has this HTML:

```
<div ng-controller="MyController" >
    <div>This div will be replaced</div>
</div>
```

Then the added directive would be activated when AngularJS finds the inner `div` element. Instead of this `div` element, this HTML will be inserted:

```
My first directive: {{textToInsert}}
```

As you can see, this HTML contains an interpolation directive (`{{textToInsert}}`). AngularJS will interpret this HTML again, so that the interpolation directive actually works. The value of the `$scope.textToInsert`property will be inserted at this point in the HTML.

## The template and templateUrl Properties

The easiest way to create your own directives is as shown in the example above. Your directive is intended to generate HTML, and you put that HTML inside the `template` attribute of the directive definition object. Here is the directive definition repeated from earlier, with the `template` string marked in bold:

```
myapp = angular.module("myapp", []);

myapp.directive('div', function() {
    var directive = {};

    directive.restrict = 'E'; /* restrict this directive to elements */
    directive.template = "My first directive: {{textToInsert}}";

    return directive;
});
```

In case that HTML template grows big, it is gets hard to write and maintain the HTML inside a JavaScript string. You can then put the HTML into its own file and have AngularJS load it from that file. You do so by putting the URL of the HTML template file into the `templateUrl` property of the directive definition object. Here is an example:

```
myapp = angular.module("myapp", []);

myapp.directive('div', function() {
    var directive = {};

    directive.restrict = 'E'; /* restrict this directive to elements */
    directive.templateUrl = "/myapp/html-templates/div-template.html";

    return directive;
});
```

AngularJS will now load the HTML template from the URL set in the `templateUrl` property.

Using the separate HTML template file and the `templateUrl` property is especially useful when you create more specialized directives, like a directives showing user info. Here is an example:

```
myapp = angular.module("myapp", []);

myapp.directive('userinfo', function() {
    var directive = {};

    directive.restrict = 'E'; /* restrict this directive to elements */
    directive.templateUrl = "/myapp/html-templates/userinfo-template.html";
    return directive;
});
```

This example creates a directive that is activated whenever AngularJS finds a `<userinfo>` element. AngularJS loads the HTML template found at `/myapp/html-templates/userinfo-template.html`, and interprets that as if it had been located inside the parent HTML file from the beginning.

## Isolating the $scope From the Directive

In the example above the `userinfo` directive was bound hard to the `$scope` variable because the HTML template referenced the `textToInsert` property directly. Referencing `$scope` variables directly makes it hard to reuse the directive more than once within the same controller, since the `$scope` variables typically have the same values everywhere inside the same controller. For instance, if you wanted to have this HTML in your page:

```
<userinfo></userinfo>
<userinfo></userinfo>
```

Then the two `<userinfo>` elements would be replaced by the same HTML template, which is bound to the same `$scope` variable. The result would be that the two `<userinfo>` elements would be replaced by the exact same HTML code.

To be able to bind the two `<userinfo>` elements to different values in the `$scope` object, you need to bind the HTML template to an *isolate scope*.

An isolate scope is a separate scope object tied to the directive. Here is how you define it in the directive definition object:

```
myapp.directive('userinfo', function() {
    var directive = {};

    directive.restrict = 'E';

    directive.template = "User : {{user.firstName}} {{user.lastName}}";

    directive.scope = {
        user : "=user"
    }

    return directive;
})
```

Notice how the HTML template has two interpolation directives bound to `{{user.firstName}}` and `{{user.lastName}}`. Notice the `user.` part. And notice the `directive.scope` property. The `directive.scope`property is a JavaScript object which contains a property named `user`. The `directive.scope` property is the isolate scope object, and the HTML template is now bound to the `directive.scope.user` object (via the `{{user.firstName}}` and `{{user.lastName}}` interpolation directives).

The `directive.scope.user` property is set to `"=user"`. That means, that the `directive.scope.user` property is bound to the property in the scope property (not in the isolate scope) with the name passed to the `user`attribute of the `<userinfo>` element. It sounds confusing, so look at this HTML example:

```
<userinfo user="jakob"></userinfo>
<userinfo user="john"></userinfo>
```

These two `<userinfo>` element contain a `user` attribute. The value of these attributes contain the names of properties in the `$scope` object which are to be referenced by the isolate scope object's `userinfo` property.

Here is a full example:

```
<userinfo user="jakob"></userinfo>
<userinfo user="john"></userinfo>

<script>
myapp.directive('userinfo', function() {
    var directive = {};

    directive.restrict = 'E';

    directive.template = "User : <b>{{user.firstName}}</b> <b>{{user.lastName}}</b>";

    directive.scope = {
        user : "=user"
    }

    return directive;
});

myapp.controller("MyController", function($scope, $http) {
    $scope.jakob = {};
    $scope.jakob.firstName = "Jakob";
    $scope.jakob.lastName  = "Jenkov";

    $scope.john = {};
    $scope.john.firstName = "John";
    $scope.john.lastName  = "Doe";
});

</script>
```

## The compile() and link() Functions

If you need to do something more advanced inside your directive, something that you cannot do with an HTML template, you can use the `compile()` and `link()` functions instead.

The `compile()` and `link()` functions define how the directive is to modify the HTML that matched the directive.

The `compile()` function is called once for each occurrence of the directive in the HTML page. The `compile()` function can then do any one-time configuration needed of the element containing the directive.

The `compile()` function finishes by returning the `link()` function. The `link()` function is called every time the element is to be bound to data in the `$scope` object.

As mentioned, you add the `compile()` function to the directive definition object, and the `compile()` function has to return the `link()` function when executed. Here is how that looks:

```
<script>
myapp = angular.module("myapp", []);
myapp.directive('userinfo', function() {
    var directive = {};

    directive.restrict = 'E'; /* restrict this directive to elements */


    directive.compile = function(element, attributes) {
        // do one-time configuration of element.

        var linkFunction = function($scope, element, atttributes) {
            // bind element to data in $scope
        }

        return linkFunction;
    }

    return directive;
});
</script>
```

The `compile()` function takes two parameters:
The `element` and `attributes` parameters.

The `element` parameter is a jqLite wrapped DOM element. AngularJS contains a lite version of jQuery to help you do DOM manipulation, so the `element`'s DOM manipulation methods are the same as you know from jQuery.

The `attributes` parameter is a JavaScript object containing properties for all the attributes of the DOM element. Thus, to access an attribute named `type` you would write `attributes.type`.

The `link()` function takes three parameters: The `$scope` parameter, the `element` parameter and the `attributes` parameter.
The `element` and `attributes` parameter is the same as passed to the `compile()`function. The `$scope` parameter is the normal scope object, or an isolate scope in case you have specified one in the directive definition object.

The `compile()` and `link()` function names are actually confusing. They are inspired by compiler terms. I can see the resemblance, but a compiler parses an input once, and creates an output. A directive configures an HTML element and then updates that HTML subsequently whenever the `$scope` object changes.

A better name for the `compile()` function would have been something like `create()`, `init()` or `configure()`. Something that signals that this function is only called once.

A better name for the `link()` function would have been something like `bind()` or `render()`, which signals that this function is called whenever the directive needs to bind data to it, or to re-render it.

Here is a full example that shows a directive that uses both a `compile()` and `link()` function:

```
<div ng-controller="MyController" >
    <userinfo >This will be replaced</userinfo>
</div>

<script>
    myapp = angular.module("myapp", []);
    myapp.directive('userinfo', function() {
        var directive = {};

        directive.restrict = 'E'; /* restrict this directive to elements */

        directive.compile = function(element, attributes) {
            element.css("border", "1px solid #cccccc");

            var linkFunction = function($scope, element, attributes) {
                element.html("This is the new content: " + $scope.firstName);
                element.css("background-color", "#ffff00");
            }

            return linkFunction;
        }

        return directive;
    })
    myapp.controller("MyController", function($scope, $http) {
        $scope.cssClass = "notificationDiv";

        $scope.firstName = "Jakob";

        $scope.doClick = function() {
            console.log("doClick() called");
        }
    });
</script>
```

The `compile()` function sets a border on the HTML element. This is only executed once because the `compile()` function is only executed once.

The `link()` function replaces the content of the HTML element, and sets the background color to yellow.

There is no particular reason why the border was set in the `compile()` function, and the background color in the `link()` function. Both could have been set in the `compile()` function, or both in the `link()` function. If set in the `compile()` function they would only have been set once (which is often what you want). If set in the `link()` function they would be set every time the HTML element is bound to data in the `$scope` object. This might be useful if you needed to set the border and background color differently depending on data in the `$scope` object.

### Setting Only a link() Function

Sometimes you do not need the `compile()` step for your directive. You only need th `link()` function. In that case you can set the `link()` function directly on the directive definition object. Here is the example from before, with only a link function:

```
<div ng-controller="MyController" >
    <userinfo >This will be replaced</userinfo>
</div>

<script>
    myapp = angular.module("myapp", []);
    myapp.directive('userinfo', function() {
        var directive = {};

        directive.restrict = 'E'; /* restrict this directive to elements */

        directive.link = function($scope, element, attributes) {
                element.html("This is the new content: " + $scope.firstName);
                element.css("background-color", "#ffff00");
        }

        return directive;
    })
    myapp.controller("MyController", function($scope, $http) {
        $scope.cssClass = "notificationDiv";

        $scope.firstName = "Jakob";

        $scope.doClick = function() {
            console.log("doClick() called");
        }
    });
</script>
```

Notice how the `link()` function does the same as the `link()` function returned in the previous example.

## Directives Which Wraps Elements Via Transclusion

The examples we have seen so far all set the content of the element matching the directive themselves, either via JavaScript code or an HTML template. But what if you wanted a directive to wrap elements inserted into the directive body by the developer? For instance:

```
<mytransclude>This is a transcluded directive {{firstName}}</mytransclude>
```

The directive is marked by the `<mytransclude>` element. But the content inside it is set by the developer. Thus, this part of the HTML should not be replaced by the directive's HTML template. We actually want that part of the HTML to be processed by AngularJS. This processing is called "transclusion".

In order to make AngularJS process the HTML inside a directive, you have to set the `transclude` property of the directive definition object to `true`. You will also have to tell AngularJS what part of the directive's HTML template that is to contain the transcluded HTML. You do so by inserting the `ng-transclude` attribute (a directive, really) into the HTML element in the HTML template where you want the transcluded HTML inserted.

Here is an AngularJS directive that shows how to use transclusion:

```
<mytransclude>This is a transcluded directive {{firstName}}</mytransclude>

<script>
    myapp = angular.module("myapp", []);
    myapp.directive('mytransclude', function() {
        var directive = {};

        directive.restrict = 'E'; /* restrict this directive to elements */
        directive.transclude = true;
        directive.template = "<div class='myTransclude' ng-transclude></div>";

        return directive;
    });
    myapp.controller("MyController", function($scope, $http) {
        $scope.firstName = "Jakob";
    });
</script>
```

Notice the HTML inside the `<mytransclude>` element. This HTML code contains the interpolation directive `{{firstName}}`. We want AngularJS to process this HTML for us so that interpolation directive is executed. To achieve that I have set the `transclude` property to `true` on the directive definition object. I have also inserted an `ng-transclude` attribute into the HTML template. This attribute tells AngularJS what element to insert the transcluded HTML into.