Creating Custom AngularJS Directives Part I – The Fundamentals

By Dan Wahlin

AngularJS provides many directives that can be used to manipulate the DOM, route events to event handler functions, perform data binding, associate controllers/scope with a view, plus much more. Directives such as **ng-click**, **ng-show/ng-hide**, **ng-repeat**,and many others found in the AngularJS core script make it easy to get started using the framework.

Although the built-in directives cover many different scenarios, there may be times when you find yourself needing custom directives and wondering how to get started writing them. In this series of articles I'll provide a step-by-step look at how directives work and how you can get started using them. The overall goal is to move slowly and provide simple, digestible directive examples that get more and more involved as the series progresses.

Writing Custom Directives is Easy Right?

AngularJS directives can be a bit intimidating the first time you see them. They offer many different options, have a few cryptic features (and cryptic is my politically correct term for "what were they thinking here?"), and are generally challenging at first. However, once you understand the pieces and how they fit together you'll find that they're not that bad. I compare it to learning to play an instrument. The first time you play a piano or guitar you aren't very good and feel completely incompetent. However, after putting in the necessary practice time you end up getting pretty good and capable of handling more advanced music. It's the same with directives – they take a little practice but you can do powerful things with them once you get the hang of it. I do look forward to the day where features found in the Polymer project

and Web Components spec replace the directives that we write today. Check out those links if you want a glimpse into the future direction of directives.

Getting Started with Custom Directives

Why would you want to write a custom directive? Let's say that you've been tasked with converting a collection of customers into some type of grid that's displayed in a view. Although you can certainly add DOM functionality into a controller to handle this, doing that would make it hard to test the controller and breaks the separation of concerns – you just don't want to do that in an AngularJS application. Instead, that type of functionality should go into a custom directive. In another case you may have some data binding that occurs across many views and you want to re-use the data binding expressions. While a child view loaded with ng-include (more on this in a moment) could be used, directives also work well in this scenario. There are numerous other ways that custom directives can be used, these examples are only scratching the surface.

Let's start by looking an a very basic example of an AngularJS directive. Assume that we have the following module and controller defined in the application (I'll refer to this module and controller in future articles as well):

```
var app = angular.module('directivesModule', []);


app.controller('CustomersController', ['$scope', function ($scop
e) {
    var counter = 0;
    $scope.customer = {
        name: 'David',
        street: '1234 Anywhere St.'
    };
```

```
    $scope.customers = [
        {
            name: 'David',
            street: '1234 Anywhere St.'
        },
        {
            name: 'Tina',
            street: '1800 Crest St.'
        },
        {
            name: 'Michelle',
            street: '890 Main St.'
        }
    ];

    $scope.addCustomer = function () {
        counter++;
        $scope.customers.push({
            name: 'New Customer' + counter,
            street: counter + ' Cedar Point St.'
        });
    };

    $scope.changeData = function () {
        counter++;
        $scope.customer = {
            name: 'James',
            street: counter + ' Cedar Point St.'
        };
    };
```

```
}]);
```

Let's say that we find ourselves writing out a data binding expression similar to the following over and over in views throughout the app:

```
Name: {{customer.name}}
<br />
Street: {{customer.street}}
```

One technique that can be used to promote re-use is to place the data binding template into a child view (I'll call it **myChildView.html**) and reference it from within a parent view using the **ng-include** directive. This allows**myChildView.html** to be re-used throughout the application wherever it's needed.

```
<div ng-include="'myChildView.html'"></div>
```

While this gets the job done, another technique that can be used is to place the data binding expression into a custom directive. To create a directive you first locate the target module that the directive should be placed in and then call it's **directive()** function. The **directive()** function takes the name of the directive as well as a function. Here's an example of a simple directive that embeds the data binding expression:

```
angular.module('directivesModule').directive('mySharedScope', fu
nction () {
    return {
```

```
        template: 'Name: {{customer.name}}<br /> Street: {{custo
mer.street}}'
    };
});
```

Does this buy us much over using a child view along with the **ng-include** directive? Not at this point. However, directives can do a lot more with just a little code, plus they make it a piece of cake to attach the functionality they offer to an element. An example of attaching the **mySharedScope** directive to a ** ** element is shown next:

```
<div my-shared-scope></div>
```

When the directive is run it'll output the following based on the data in the controller shown earlier:

Name: David
Street: 1234 Anywhere St.

The first thing you'll probably notice is that the **mySharedScope** directive is referenced in the view by using **my-shared-scope**. Why is that? The answer is that directive names are defined use a camel-case syntax. For example, when you use the **ngRepeat** directive you use the hyphenated version of the name: **ng-repeat**.

Another interesting thing about this directive is that it inherits the scope from the view by default. If the controller shown earlier (CustomersController) is bound to the view then the **$scope's customer** property is accessible to the directive. This is referred to as **shared**

scope and works great in situations where you know a lot about the parent scope. In situations where you want to re-use a directive you can't rely on knowing the properties of the scope though and will likely use something referred to as **isolate scope**. I'll provide more details on isolate scope in the next article.

Directive Properties

In the previous **mySharedScope** directive a single property named **template** was defined in the object literal returned from the function. This property is responsible for defining the template code (a data binding expression in this case) that should be used to generate HTML. What other properties are available to use?

Custom directives will typically return an object literal that is responsible for defining properties needed by the directive such as templates, a controller (if one is used), DOM manipulation code, and more. Several different properties can be used (you'll find a complete list of them here). Here's a look at a few of the key properties that you may come across and an example of using them:

```
angular.module('moduleName')
    .directive('myDirective', function () {
    return {
        restrict: 'EA', //E = element, A = attribute, C = class, M = comment
        scope: {
            //@ reads the attribute value, = provides two-way binding, & works with functions
            title: '@'          },
        template: '<div>{{ myVal }}</div>',
        templateUrl: 'mytemplate.html',
```

```
        controller: controllerFunction, //Embed a custom control
ler in the directive
        link: function ($scope, element, attrs) { } //DOM manipu
lation
    }
});
```

A short explanation of each of the properties is shown next:

Property	**Description**
restrict	Determines where a directive can be used (as an element, attribute, CSS class, or comment).
scope	Used to create a new child scope or an isolate scope.
template	Defines the content that should be output from the directive. Can include HTML, data binding expressions, and even other directives.
templateUrl	Provides the path to the template that should be used by the directive. It can optionally contain a DOM element id when templates are defined in <script> tags.
controller	Used to define the controller that will be associated with the directive template.
link	Function used for DOM manipulation tasks.

Manipulating the DOM

In addition to performing data binding operations with templates (and there's much more to that story that I'll cover in future articles!), directives can also be used to manipulate the DOM. This is done using the link function shown earlier.

The link function normally accepts 3 parameters (although others can be passed in some situations) including the scope, the element that the directive is associated with, and the

attributes of the target element. An example of a directive that handles click, mouseenter, and mouseleave events on an element is shown next:

```
app.directive('myDomDirective', function () {
    return {
        link: function ($scope, element, attrs) {
            element.bind('click', function () {
                element.html('You clicked me!');
            });
            element.bind('mouseenter', function () {
                element.css('background-color', 'yellow');
            });
            element.bind('mouseleave', function () {
                element.css('background-color', 'white');
            });
        }
    };
});
```

To use the directive you can add the following code into the view:

```
<div my-dom-directive>Click Me!</div>
```

As the mouse enters or leaves the the background color will change between yellow and white (yes, embedded styles are used in this simple example but CSS classes can certainly be used as well). When the target element is clicked the inner HTML is changed to “You clicked

me!". There's of course much, much more you can do when it comes to DOM manipulating but this should help get you started.

Structuring AngularJS Directive Code

Although the **mySharedScope** and **myDomDirective** directives work fine, I like to follow a specific pattern when defining directives and other AngularJS components. Here's an example:

```
(function () {

    var directive = function () {
        return {

        };
    };

    angular.module('moduleName')
        .directive('directiveName', directive);

}());
```

This code wraps everything with an immediately invoked function to pull everything out of the global scope. It then defines the directive functionality in a function assigned to the **directive** variable. Finally, a call is made to the module's **directive()** function and the **directive** variable is passed in. There are several other techniques that can be used to structure code (see a previous blog post I wrote on the subject) but this is the general pattern I like to follow.

Summary

In this first article on AngularJS directives you've seen some of the fundamentals and learned how to create two basic directives. We've barely scratched the surface though! In the next article I'll discuss isolate scope and different properties referred to as **local scope properties** that can be used for data binding and much more.

Check out my AngularJS Custom Directives course for additional information about building your own directives.