Creating Custom AngularJS Directives Part 6 - Using Controllers

By Dan Wahlin

Up to this point in the AngularJS directives article series you've learned about many key aspects of directives but haven't seen anything about how controllers fit into the picture. Although controllers are typically associated with routes and views, they can also be embedded in AngularJS directives. In fact, there are many scenarios where custom directives can take advantage of controllers to minimize code and simplify maintenance. While using controllers in directives is certainly optional, if you'd prefer to build directives using similar techniques that you use now to build views then you'll find controllers are essential in many cases. By using controllers, directives start to feel like "child views".

In this article I'll walk through the process of integrating controllers into directives and show the role that they can play. Let's start off by looking at a directive that doesn't use a controller and talk through the pros and cons of the approach.

A Directive without a Controller

Directives provide several different ways to render HTML, collect data, and perform additional tasks. In situations where a directive is performing a lot of DOM manipulation, using the **link** function makes sense. Here's a simple example of the **link** function in action:

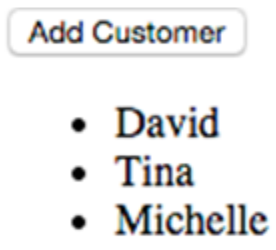
```
(function() {


  var app = angular.module('directivesModule');
  app.directive('domDirective', function () {
      return {
          restrict: 'A',
          link: function ($scope, element, attrs) {
              element.on('click', function () {
                  element.html('You clicked me!');
              });
              element.on('mouseenter', function () {
                  element.css('background-color', 'yellow');
              });
              element.on('mouseleave', function () {
                  element.css('background-color', 'white');
              });
          }
      };
  });

}());
```

Adding a controller into this directive doesn't make much sense given that the goal is to handle events and manipulate the DOM. Although it would be possible to accomplish the same task using a view in the directive (along with built-in AngularJS directives such as ng-click) and controller there's really no reason to add a controller into this directive if DOM manipulation is the overall end goal.

In cases where you're manipulating the DOM, integrating data into the generated HTML, handling events, and more, adding a controller can minimize the amount of code you write and simplify the overall process in some cases. To make this more clear, let's look at an example of a directive then renders a list of items and provides a button that can be used to add items to the list. Here's the simple output that the directive renders:



There are several different ways to handle rendering this type of UI. A typical DOM-centric approach would use the **link** function to handle everything as shown in the following directive. Keep in mind there are many ways to do this and the overall goal is to demonstrate DOM-centric code (as simply as possible):

```
(function() {

  var app = angular.module('directivesModule');

  app.directive('isolateScopeWithoutController', function () {

      var link = function (scope, element, attrs) {

              //Create copy of original data that's passed in
              var items = angular.copy(scope.datasource);

              function init() {
                  var html = '<button id="addItem">Add Item' +
                        '</button><div></div>';
```

```
            element.html(html);

            element.on('click', function(event) {
                if (event.srcElement.id === 'addItem') {
                    addItem();
                    event.preventDefault();
                }
            });
        }

        function addItem() {
            //Call external function passed in with &
            scope.add();

            //Add new customer to the local collection
            items.push({
                name: 'New Directive Customer'
            });

            render();
        }

        function render() {
            var html = '<ul>';
            for (var i=0,len=items.length;i<len;i++) {
                html += '<li>' + items[i].name + '</li>'
            }
            html += '</ul>';

            element.find('div').html(html);
```

```
                }

                init();
                render();
        };



        return {
            restrict: 'EA',
            scope: {
                datasource: '=',
                add: '&',
            },
            link: link
        };
    });

}());
```

Although this code gets the job done, it’s more along the lines of a jQuery plugin and takes what I refer to as a “control-oriented” approach where tag names and/or IDs are prevalent in the code. All of the DOM manipulation is handled manually which is fine and maybe even preferred in some cases (for performance reasons for example), but it’s definitely not the normal way we build Angular apps. The DOM manipulation code is mixed in with the scope which starts to get messy especially as the directive grows in size.

As the button is clicked the **addItem()** function is called which handles calling an isolate scope property (**add**) and invoking the **render()** function which renders a <ul> tag and multiple <li> tags. There’s nothing wrong with this approach per se, but I’m not a fan of having a lot of separate strings embedded in the JavaScript since they can cause a maintenance nightmare over time. While a small directive like this is fairly easy to maintain, the code can get more challenging as the directive has additional features added.

There's also a more subtle issue at play in this code. When **scope.add()** is called the invoked parent scope function will need to use **$scope.$apply()** to update any properties in the parent scope since the call to **add** is being made from vanilla JavaScript rather than from within the context of AngularJS (something that's outside the scope of this article, but definitely important to consider). Finally, the directive doesn't resemble the "child view" concept that was mentioned at the beginning of the article – it's just a bunch of code. How can a controller help out in this example? Let's take a look.

Adding a Controller and View into a Directive

The directive shown in the previous section gets the job done but what if you could write it much like you'd write a standard AngularJS view and use a more data-oriented approach as opposed to the control-oriented approach that the DOM takes? By using a controller and view in a directive the development process feel more along the lines of what you do everyday in AngularJS applications.

Here's an example of converting the directive shown earlier into a cleaner version (in my opinion anyway) that relies on a controller and a simple view:

```
(function() {

  var app = angular.module('directivesModule');

  app.directive('isolateScopeWithController', function () {
```

```
var controller = ['$scope', function ($scope) {

      function init() {
          $scope.items = angular.copy($scope.datasource);
      }

      init();

      $scope.addItem = function () {
          $scope.add();

          //Add new customer to directive scope
          $scope.items.push({
              name: 'New Directive Controller Item'
          });
      };
  }],

  template ='<button ng-click="addItem()">Add Item</button><ul>' +
            '<li ng-repeat="item in items">{{ ::item.name }}' +
            '</li></ul>';

  return {
      restrict: 'EA', //Default in 1.3+
      scope: {
          datasource: '=',
          add: '&',
      },
            controller: controller,
            template: template
        };
    });

}());
```

The directive could be used in one of the following ways:

```
Attribute: <div isolate-scope-with-controller datasource="customers"
                add="addCustomer()"></div>


Element: <isolate-scope-with-controller datasource="customers"
           add="addCustomer()">
         </isolate-scope-with-controller>
```

Looking through the directive code you can see that it’s very similar to the approach you’d take for writing a normal view with a controller. I’d argue that it looks like you’re writing a “child view” as mentioned at the beginning of this article since the code is focused more on the data and less on the “controls” in the view. The view takes advantage of AngularJS directives to handle various control rendering tasks which eliminates all of the DOM code that had to be written before.

The view is defined using the **template** property and the controller is defined using the **controller** property. Keep in mind that the view can be loaded from a file using the **templateUrl** property or from the**$templateCache** as well – it doesn’t have to be embedded directly in the directive. The **templateUrl** or**$templateCache** options are really useful when a view has a lot of HTML in it that you don’t want embedded in the directive.

As mentioned, the code in the view leverages existing AngularJS directives such as **ng-click** and **ng-repeat** and also uses **{{ … }}** data binding expressions. This eliminates the DOM code shown earlier in the DOM-centric/control-oriented directive. The controller has the **$scope** injected as you'd expect and uses it to define an**items** property which is consumed by **ng-repeat** in the view to generate <li> tags. As the button in the view is clicked the **addItem()** function on the **$scope** is invoked which calls the **add** isolate scope property and adds a new item object into the local collection (since angular.copy() is used items added into the local collection won't show up in the parent scope). Because **addItem()** is called using **ng-click**, the parent scope call that is made ($scope.add()) won't need to worry about using **$scope.$apply()** as mentioned in the earlier section.

In situations where a directive is being written with raw performance in mind then the DOM-centric approach shown earlier may be preferred I realize since you'd be purposely taking over control of the HTML that's generated and avoiding the use of Angular directives. If you ever attend one of my conference sessions or training classes you'll often hear me say, "Use the right tool for the right job". I've never believed that "one size fits all" and know that each situation and application is unique.

This thought process definitely applies to directives since there are many different ways to write them. In many situations I'm happy with how AngularJS performs and know about the pitfalls to avoid so I prefer the controller/view type of directive whenever possible. It makes maintenance much easier down the road since you can leverage existing Angular directives in the directive's view and modify the view using a controller and scope. If, however, I was trying to maximize performance and eliminate the use of directives such as ng-repeat then going the DOM-centric route with the **link** function might be a better choice. Again, choose the right tool for the right job.

Using controllerAs in a Directive

If you're a fan of the controllerAs syntax you may be wondering if the same style can be used inside of directives. The answer is "yes"! When you define a Directive Definition Object (DDO) in a directive you can add a**controllerAs** property. Starting with Angular 1.3 you'll also need to add a **bindToController** property as well to ensure that properties are bound to the controller rather than to the scope. Here's an example of the previous directive that has been converted to use the controllerAs syntax:

```
(function() {

  var app = angular.module('directivesModule');

  app.directive('isolateScopeWithControllerAs', function () {

      var controller = function () {

              var vm = this;

              function init() {
                  vm.items = angular.copy(vm.datasource);
              }

              init();

              vm.addItem = function () {
                  vm.add();

                  //Add new customer to directive scope
                  vm.items.push({
                      name: 'New Directive Controller Item'
                  });
              };
      };

      var template = '<button ng-click="vm.addItem()">Add Item' +
                     '</button>' +
                     '<ul><li ng-repeat="item in vm.items">' +
                     '{{ ::item.name }}</li></ul>';

      return {
          restrict: 'EA', //Default for 1.3+
          scope: {
              datasource: '=',
              add: '&',
          },
          controller: controller,
          controllerAs: 'vm',
          bindToController: true, //required in 1.3+ with controllerAs
          template: template
      };
  });

}());
```

Notice that a controller alias of **vm** (short for "ViewModel") has been assigned to the **controllerAs** property and that the alias is used in the controller code and in the view.

The **bindToController** property is set to **true** to ensure that properties are bound to the controller instead of the scope. While this code is very similar to the initial controller example shown earlier, it allows you to use "dot" syntax in the view (vm.customers for example) which is a recommended approach.

Conclusion

Controllers can be used to cleanup directives in many scenarios. Although using a controller isn't always necessary, you'll find that by levering the "child view" concept in directives your code can be kept more maintainable and easier to work with.

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