Understanding Angular Directives

By Jeffry Houser

A DotComIt Whitepaper

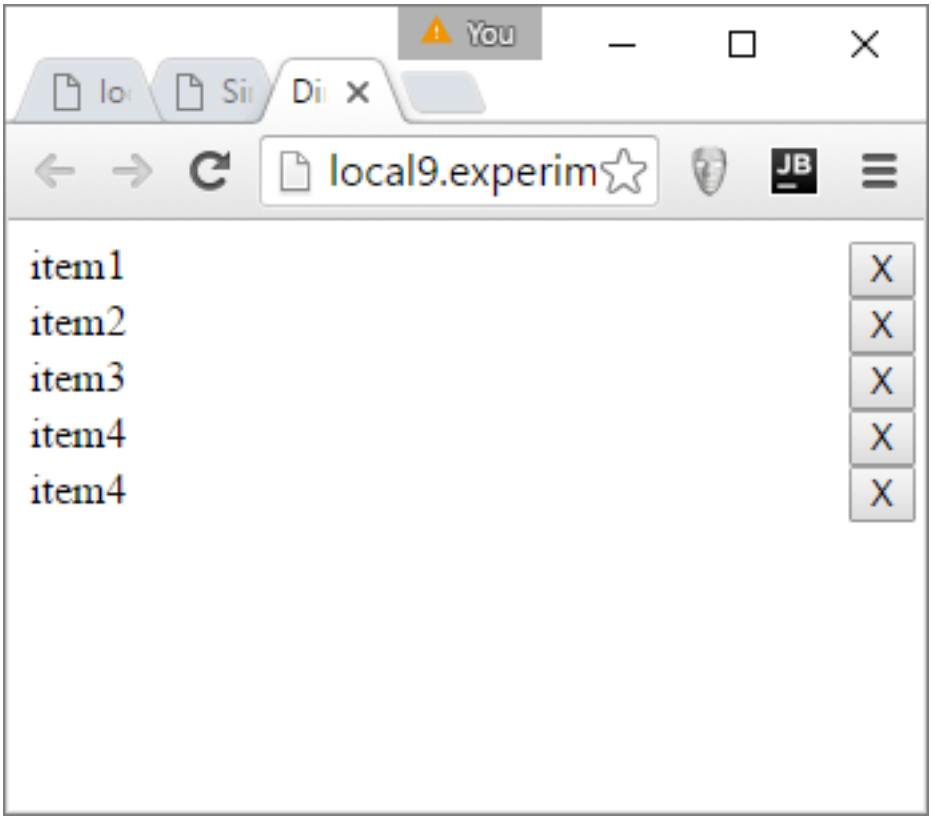
Contents

A Simple Directive

An AngularJS Directive combines HTML and JavaScript code into a single unit that can be easily reused within an Angular application. This whitepaper is intended to explore Angular directives, with a focus on optimizing the directive code for reuse.

Our Directive

The directive we're going to create in this paper will generate a list of items. It will loop over an array of items, and display each item, along with a delete button. The final directive will look something like this:



In this paper, you'll learn how to setup the directive, use external templates for HTML, send values into the directive, run functions when something in the directive happens, and how to call functions inside the directive. This first section will create a basic directive which we will iterate over throughout this whitepaper.

Create the App Infrastructure

The first thing we need to do in any AngularJS project is to import the framework:

```
<script src="//ajax.googleapis.com/ajax/libs/angularjs/1.5.0-beta.1/angular.min.js">
</script>
```

The Angular framework is imported using the HTML Script tag. Next, create the Angular module:

```
<script>
 angular.module('directiveTestApp',[]);
</script>
```

The Angular module is named directiveTestApp and there are no arguments passed into the directive. The rest of our JavaScript code for this article will go inside this script tag.

Every Angular app needs a controller

```
angular.module('directiveTestApp')
       .controller('directiveTestController',
                   ['$scope',
                    function($scope){
                    }
                   ])
```

There is no content in the main controller yet, but we'll come back to that.

Next, create the HTML code for this application:

```
<body ng-app="directiveTestApp">
<div ng-controller="directiveTestController">
</div>
</body>
```

The HTML UI is empty for the moment; but the module name is added to the body tag with the ngApp directive. The directiveTestController is added to a div with the ngController tag. If you have experience with Angular applications, then this approach should be familiar.

One last thing I want to do in the controller is to create the array of items that the directive will loop over. Inside the controller, create a JavaScript array:

```
$scope.objectArray = [
    {itemLabel:'item1'},
    {itemLabel:'item2'},
    {itemLabel:'item3'},
    {itemLabel:'item4'},
    {itemLabel:'item4'}
];
```

Now we are set to create our first directive.

Creating a Simple Directive

An Angular directive is created as part of an Angular Module. A directive function is used. It requires two arguments, similar to how services, controllers, and other Angular elements are created. The first argument is a string that specifies the name of the directive. The second is a function which defines the directive's functionality.

```
angular.module('directiveTestApp').directive('dciTestdirective',function(){
});
```

The directive is named dciTestdirective, and I'll talk more about naming in a second. Right now the directive doesn't do anything, and if you were to try to use it, Angular would return an error. The directive function must return an object that contains properties which define the directive. In this case, we are only going to specify a single parameter, the template:

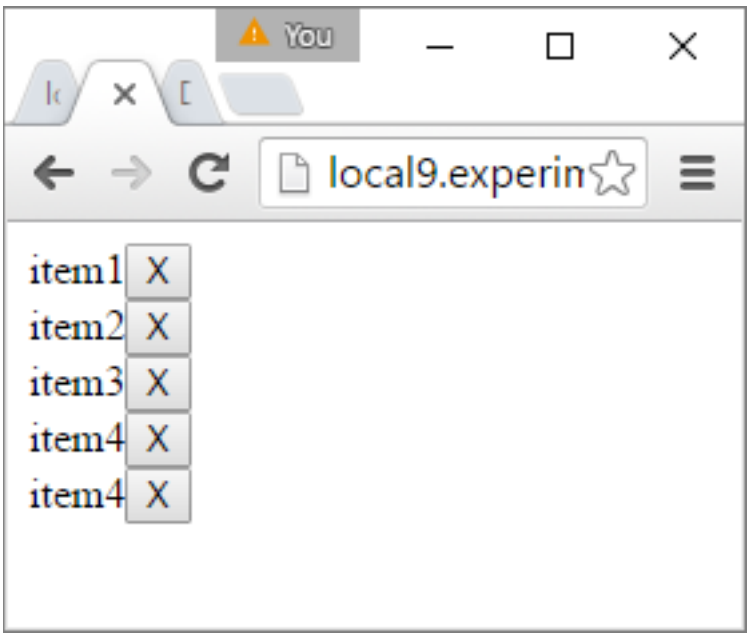
```
return {
  template: "<div ng-repeat='object in objectArray'>
              <span>{{object.itemLabel}}</span>
              <button >X</button>
             </div>"
};
```

The template contains some in-line HTML. It loops over the objectArray; which was created in the app's controller; and displays the itemLabel along with an X button.

Now add the directive to the HTML, inside the controller:

```
<dci-Testdirective></dci-Testdirective>
```

Run the code and you'll see this:



Play with this sample here.

How do you name a Directive?

Naming of directives is a subject that should be approached with care. When I first started learning directives, this confused me. The key is to understand that there are two parts to each directive, the prefix and the directive name. When naming a directive with the directive function, the prefix should

start in all lowercase. The name starts with a capital letter. So, in the case of our directive the prefix is dci—which is short for DotComIt, my company. The name is Testdirective.

When Angular parses the DOM, it looks for selectors, such as element names or arguments, using a specific formula that is called normalization. Our directive, named dciTestdirective could be written in any number of ways in the HTML template:

- dci-testdirective
- dci: testdirective
- dci_testdirective
- data-dci-testdirective
- x-dci-testdirective

If Angular finds something in one of the above formats, it runs it through a process called normalization. Then this process occurs:

1. The name is put into all lowercase.
2. x- or data- are stripped from the front of the element or attribute.
3. The delimiters are removed. The dellimiters are the colon (:), dash (-), or underscore (_).
4. The first character after the delimiter is made into uppercase.

When creating a directive, you must name it properly or else Angular won't know how to match the use of the directive to the actual directive code.

For practical purposes, I always use the dash (-) as a separator between the directive prefix and the directive name. I never put use the prefix x- or data- with my directives.

What Next?

The current directive is lacking. Here are some problems:

- The objectArray must be defined in the controller where you use the directive. This is not good use of encapsulation. The directive should have its required parameters passed to it using a defined API.
- The template is created in-line as a single string, which makes it hard to change or edit.
- The in-line template could be styled to improve its look.
- The functionality is not yet complete. The 'x' button does nothing yet, and the directive offers no other way to interact with the data it displays.

We'll iterate over this directive throughout this paper in order to flesh this out.

Using an External Template

I find that inline templates are hard to modify. This becomes a bigger problem as the template grows in complexity and size. No one wants to edit HTML as if it were a large string. Thankfully Angular supports the use of an external template.

Create a Template

First, let's revisit our existing directive:

```
angular.module('directiveTestApp').directive('dciTestdirective',function(){
    return {
        template: "<div ng-repeat='object in objectArray'>
                     <span>{{object.itemLabel}}</span>
                     <button>X</button>
                   </div>"
    };
});
```

This directive loops over an array of objects and displays them. The directive object has one property, named template. Let's move the template contents to an external template.

Create a file named 02ExternalTemplate.html. The 02 in front of the name tells us that this file is part of the second sample for this article series. The text tells us what the file is used for. Populate the file with some cut and paste:

```
<div ng-repeat='object in objectArray'>
   <span>{{object.itemLabel}}</span>
   <button>X</button>
</div>
```

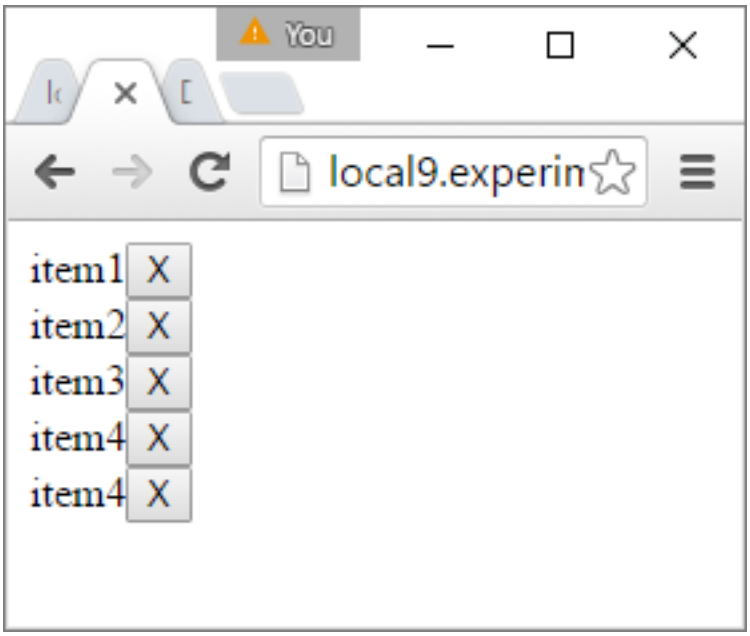
That is simple. Now let's tell the directive how to use the template.

Tell the Directive to use an External Template

The original directive uses the template property on the directive object. We are going to replace it with the templateUrl property. The templateUrl property tells Angular to look at an external file for the template:

```
angular.module('directiveTestApp').directive('dciTestdirective',function(){
    return {
        templateUrl : '02ExternalTemplate.html'
    };
});
```

Run the code as is, and you'll see the same screen you saw before.



Changing the underlying architecture has not changed the HTML page.

Add Some Styles

While we're creating the external template, let's add some styles to make it look a bit nicer. For the sake of this sample, I'll add the styles directly to the external template file. First, I'll show you the styles to create and then we'll modify the HTML to use the styles. First, I want the directive to expand the full width of the container it resides in. I can do this with a style like this:

```
<style>
    .dciTestdirective-width100 { width:100% }
</style>
```

Next, I want the item label to spread as far as it needs; while the x button should take up a smaller portion of the available space. First, the style for the label:

```
.dciTestdirective-horizontal-layout-94{
    display: inline-block;
    vertical-align: top;
    width:94%;
    height:100%;
}
```

The style code can go in the same style block as the width100 style. The display property is set to inline-block so that the two elements reside next to each other.

This is the style for the button:

```
.dciTestdirective-horizontal-layout-4{
    display: inline-block;
```

```
    vertical-align: top;
    width:4%;
    height:100%;
}
```

The style for the button is almost identical to the style for the label. The big differentiator is that the width is set to a different percentage.

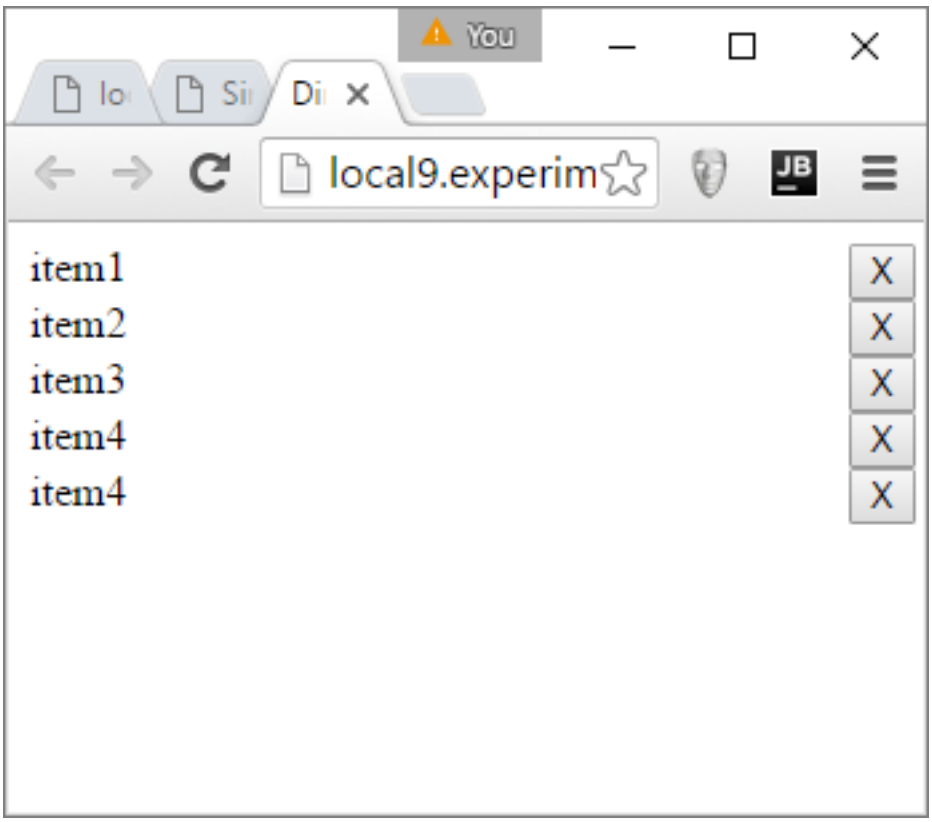
Now look at the modified HTML:

```
<div class='dciTestdirective-width100' ng-repeat='object in objectArray'>
  <span class='dciTestdirective-horizontal-layout-94'>
    {{object.itemLabel}}
  </span>
  <span class='dciTestdirective-horizontal-layout-4'>
    <button >X</button>
  </span>
</div>
```

The top level div uses the dciTestdirective-width100 style to stretch the box the full width of its container. The label is put in a span and uses dciTestdirective-horizontal-layout-94. The button is put in a different span with the style of dciTestdirective-horizontal-layout-4, giving it 4% of the allotted space.

The button was not previously wrapped in an HTML element. I did this because I wanted to change the size of the space the button was in; but not the actual button size.

The final result should look something like this:



Play with it here!

When creating styles for a directive, I like to explicitly put the directive name in the style name. The intent is to avoid conflicting style names when used in a bigger app, with multiple style sheets.

Find the External Template, Relative to the JavaScript File

When building for reuse I don't want to make assumptions on the location of the directive, relative to the web root. As such, I do not like hard coding the location of the template file. I found a great approach for dynamically finding the external template relative to the JavaScript file.

First, get a hook to the script tag:

```
var scripts = document.getElementsByTagName("script")
```

Then, pull the src attribute to the script tag:

```
var currentScript = scripts[scripts.length-1].src;
```

Now we have the location of the JavaScript file. Next, pull the file name off the currentScript to get the current path:

```
var currentScriptPath =
            currentScript.substring(0, currentScript.lastIndexOf('/') + 1)
```

Finally modify the templateUrl property of the directive:

```
return {
    templateUrl : currentScriptPath  + '02ExternalTemplate.html'
};
```

Rerun the directive and you should see the exact same screenshot as before. I like to put the script processing as part of the directive code before the return object.

Check out the live sample.

What Next?

This section covered two aspects of directive development: moving the HTML code to an external template and adding some simple styles to the directive. For the purposes of this sample, the CSS was kept inline as part of the template file. When building more complex systems, I'll put the styles in their own CSS file. To make use of the directive, the consumer must include the directive JavaScript and the CSS.

Next I'll show you how to pass values into the directive, thus removing the dependency on specific variables defined in the controller.

Passing Values into a Directive

An important aspect of building an encapsulated directive is to be able to pass data into it. This way, you can create multiple instances of a single directive, each using different data. Past entries of this series have relied on a local controller for data access. This entry will show you how to pass parameters into a directive by implementing a concept called isolated scope.

Creating an Isolated Scope

The first step to create an isolated scope is to add the scope variable to the directive object:

```
angular.module('directiveTestApp').directive('dciTestdirective',function(){
    return {
        scope : {
            dataprovider : '=',
            label : '=labelfield'
        },
        templateUrl : currentScriptPath  + '04ExternalTemplate.html'
    };
});
```

The scope variable is an object, which contains name values pairs. This directive contains two arguments:

- **dataprovider**: The dataprovider argument will contain the array which is going to be looped over.
- **labelfield**: The labelfield argument contains the property in the dataprovider's objects that will be used to display items.

I want to look at each scope value individually, and the syntax used to create it. Here is the label definition:

```
label : '=labelfield'
```

The variable name is label. This represents the name of the scope variable inside the directive. You will use the name label to reference this inside the template. The value of the variable is '=labelfield'. This refers to the attribute name that will be used when passing vales into the component. For the purpose of the label, I decided to use a different internal name than the external name.

Here is the dataprovider:

```
dataprovider : '='
```

The variable name is dataprovider. The value of the variable is just an equal sign, '='. This is a short hand way to tell Angular that the name of the variable inside the directive will be the same as the name of the attribute outside of the directive. In both cases, the name dataprovider is used.

I want to point out that the names of the properties are written all in lowercase. The attributes must be written in lowercase in order for Angular to properly sync the external attribute with the internal

attribute. I have lost too many hours because I forget about the case sensitivity of scope variable names.

Referencing the Scope Variables in the template

Let's modify the template to reference the new scope variables. Previously the template was referencing variables in the scope of the controller, like this:

```
<div class='dciTestdirective-width100' ng-repeat='object in objectArray'>
  <span class='dciTestdirective-horizontal-layout-94'>
   {{object.itemLabel}}
  </span>
  <span class='dciTestdirective-horizontal-layout-4'>
   <button >X</button>
  </span>
</div>
```

We can change both the objectArray and the itemLabel values to reference our directive's scope variables:

```
<div class='dciTestdirective-width100' ng-repeat='object in dataprovider'>
   <span class='dciTestdirective-horizontal-layout-94'>
      {{object[label]}}
   </span>
   <span class='dciTestdirective-horizontal-layout-4'>
     <button >X</button>
   </span>
</div>
```

Changing the objectArray to the dataProvider is easy and a simple replacement. Since the item label is not a known value, object property notation cannot be used. In order to access the label, we use associative array notation. That completes the modifications to the template.

Passing Parameters into the Directive

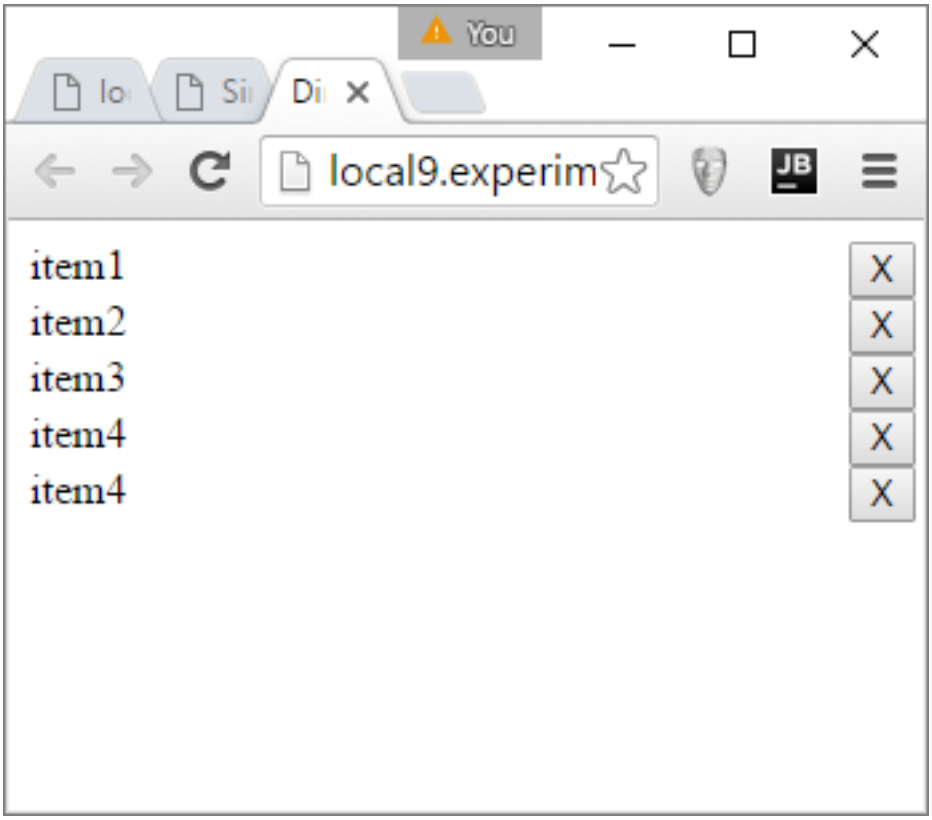
The last step to get this working is to modify the directive to include the parameters in the main HTML page:

```
<dci-testdirective dataprovider="objectArray" labelfield="'itemLabel'">
</dci-testdirective>
```

The custom directive includes the two attributes we defined in the scope; the dataProvider and the labelfield. The dataProvider attribute references the objectArray; which is defined in the controller. The labelfield uses a hard coded string value which references the itemLabel property in the dataprovider's objects.

You can run the code now, and should see this:

Play with this now!



Even though the underlying architecture of the directive has changed; the actual UI has not.

Using the Directive with a Different Data

One of the benefits of creating the isolated scope is that you can use the directive multiples times with different data. Inside the controller of the main app, create a new array:

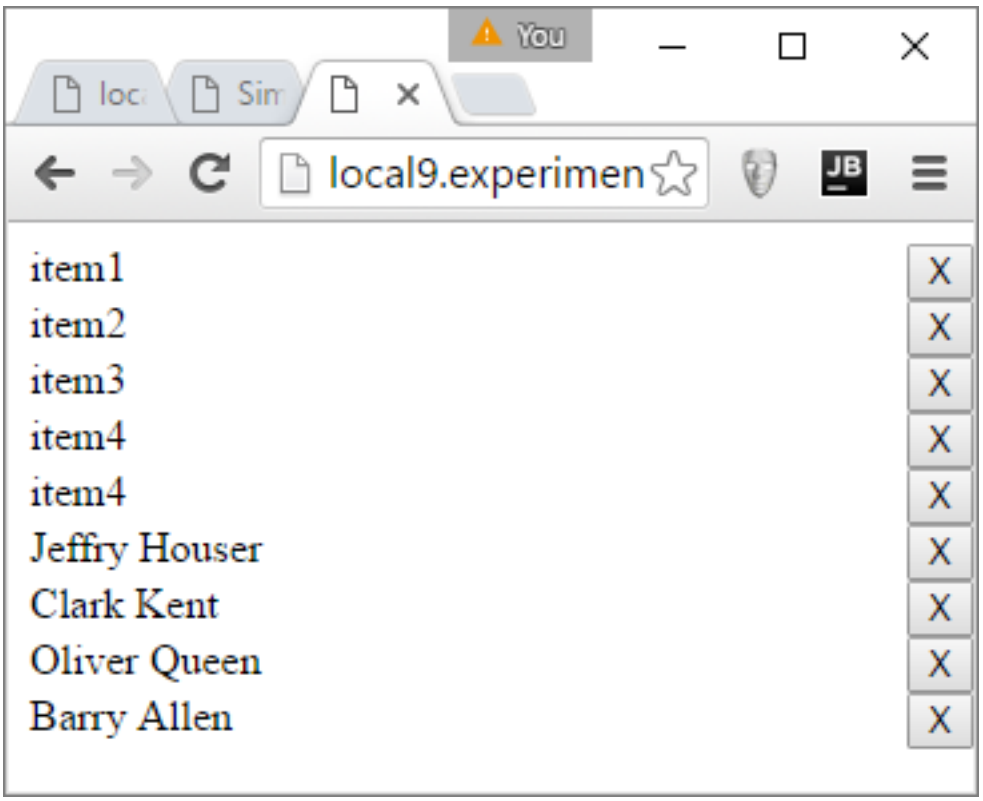
```
$scope.userArray = [
    {fullName:'Jeffry Houser'},
    {fullName:'Clark Kent'},
    {fullName:'Oliver Queen'},
    {fullName:'Barry Allen'},
];
```

This array, instead of being a generic objectArray is now a user array. It contains real fake user data which includes a user's full name.

You can easily add another instance of the dciTestdirective to the main HTML:

```
<dci-testdirective dataprovider="userArray" labelfield="'fullName'">
</dci-testdirective>
```

The directive usage is the same, but the userArray and labelfield attributes both reference the new properties.



Run this code!

What Next?

Giving the directive an isolated scope allows the directive to be easily reusable. I focused on how to get data into a directive; but the same approach is used to get data out of a directive. When the value inside the directive changes the variable outside the directive also changes.

In the next section of this whitepaper, I'm going to show you how to set default values on the scope variables.

Defaulting Directive Arguments

In the previous section, I showed how to use an isolated scope to pass arguments into a directive. This allowed for the same directive to be used multiple times, with different data, within a single view. In the implementation so far, all the directive's arguments must be defined or else an error would be thrown. We did not include any way to default argument values. This article will show you how to use a link function to default argument values.

Creating a link Function

A link function is a way to associate JavaScript code with the directive. It is included as a property on the directive object. Here is a sample link function, defined as part of the directive object:

```
link : function(scope, element, attrs, controller, transcludeFn ){
}
```

There are five arguments to the link function:

- **scope**: The scope variable is, for all intents and purposes, identical to the $scope service you use in most controllers. It is used to share variables and functions between the link function and the view.
- **element**: The element value represents the HTML element that is the directive. It is a JQuery lite object. Primarily you would use this for manipulation of the JavaScript code.
- **attrs**: The attrs is an object that contains name value pairs for each attribute passed into the directive. In the case of dciTestdirective it will include the labelfield and dataprovider.
- **controller**: The controller argument refers to the directive's controller. If the directive has its own controller then this will refer to that controller. Otherwise, it will refer to the controller of the view that contains it. Using controllers in a directive is beyond the scope of this whitepaper.
- **transcludeFn**: The transcludeFn argument refers to the transclude function. Transclusion is beyond the scope of this whitepaper.

Arguments are passed into the link function in a specific order. This is different than many other areas of Angular, such as controllers, that use dependency injection. This is the primary reason the scope argument is not called the $scope argument even though they are essentially the same thing. $scope is passed to a function using dependency injection based on the service name. In a link function, the scope is always passed as the first argument.

Populating the Link Function

The purpose of this link function is to set defaults for the label and dataProvider if they are not predefined. This can be done in JavaScript pretty easy:

```
if(!scope.label){
    scope.label = 'itemLabel';
}
if(!scope.dataprovider){
    scope.dataprovider = [];
}
```

Check if the variable exists and if it doesn't set it to some default value. The default value of the label value is the text itemLabel. The default value of the dataProvider attribute is an empty array.

We would leave that code as part of the link function's body and leave it as is. However, I am not a fan of 'random' code, so I'll often wrapup the initializing code in a function named onInit(), like this:

```
function onInit(){
    if(!scope.label){
        scope.label = 'itemLabel';
    }
    if(!scope.dataprovider){
        scope.dataprovider = [];
    }
};
onInit();
```

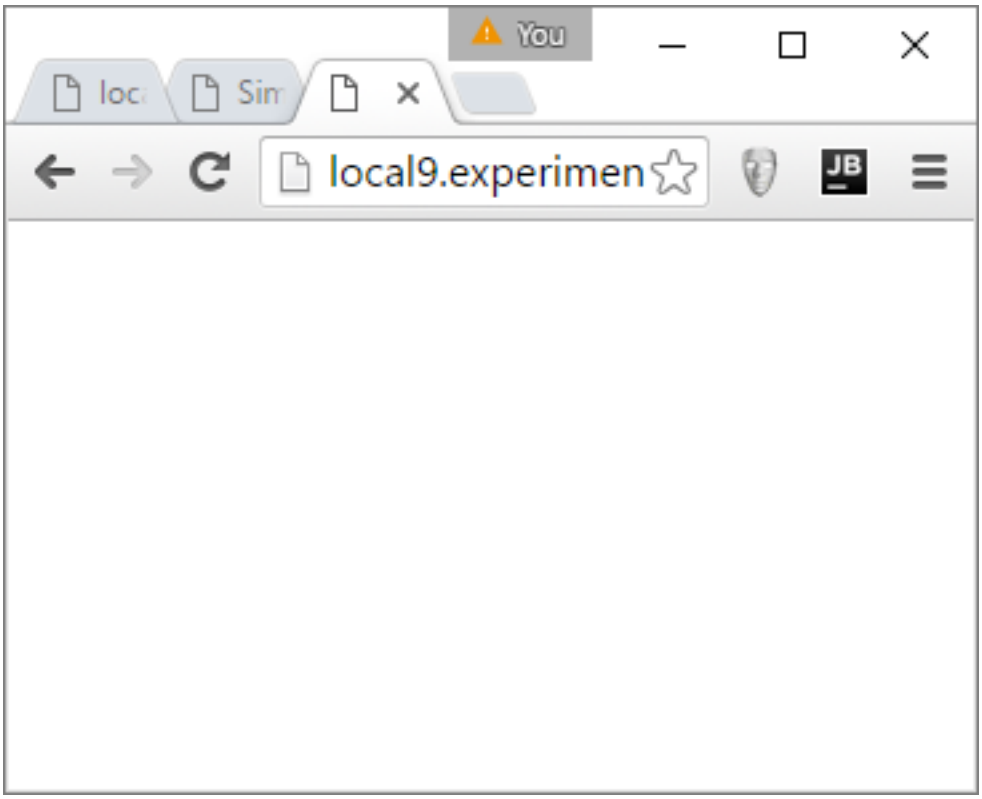
When the link function is initialized the onInit() method will be called, thus initializing the variables.

Using the Modified Directive

The modified directive can be used in three ways. The first is with no attributes set:

```
<dci-testdirective></dci-testdirective>
```

In this case the dataprovider defaults to an empty array and the labelfield property defaults to itemLabel. The result looks like this:

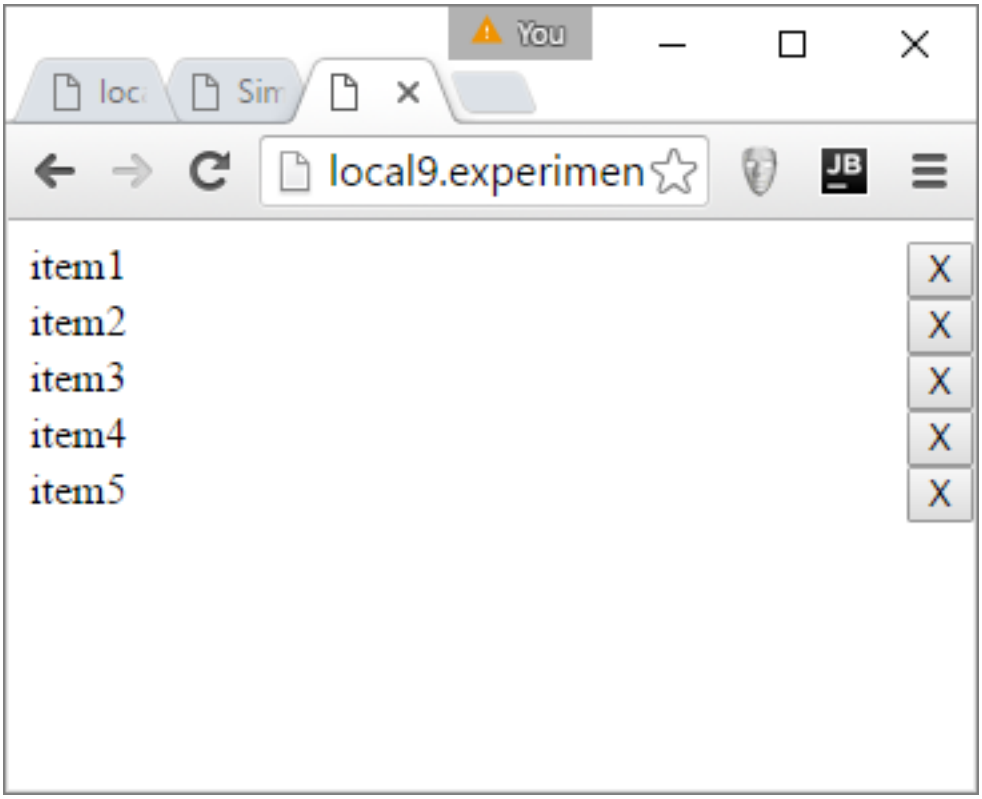


This is pretty boring because nothing is displayed at all.

The second approach is with the dataprovider set, but no labelfield set:

```
<dci-testdirective dataprovider="objectArray" ></dci-testdirective>
```

Here the dataprovider uses the value passed in while the labelfield is defaulted to itemlLabel. The result will look like this:

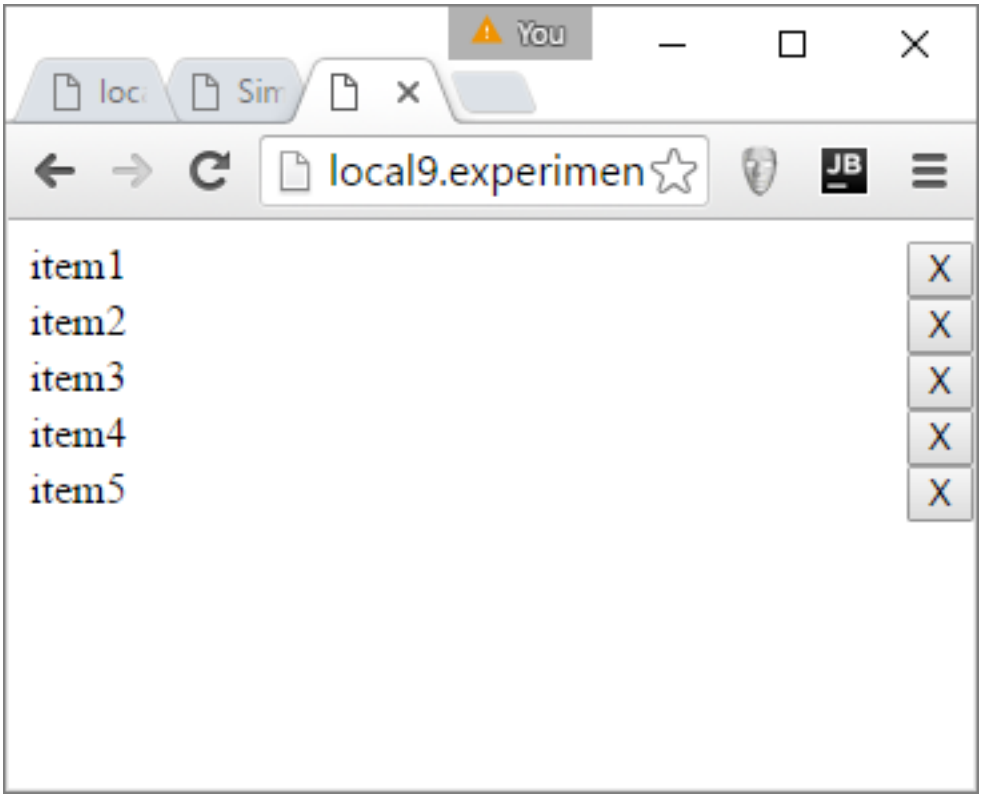


If the dataProvider did not include objects with the property itemLabel then this would have blank labels, but the X buttons would still be shown.

Finally, we can set all properties on the directive:

```
<dci-testdirective dataprovider="objectArray" labelfield="'itemLabel'">
</dci-testdirective>
```

The results here are same as the previous sample.



Play with this sample!

What Next?

The next section of this whitepaper will show how to implement the X button to remove items from the list. This won't cover new directive-specific concepts, but it is important thing to get out of the way before moving on to handling custom events.

Implement the Delete Button

This section will add more functionality to the directive we've been creating in this whitepaper. We will wire up the delete button to remove an item from the dataprovider array. This section will show you how to execute a method within the directive in response to a click event inside the directive's template. I consider this section a transitionary portion of the series because it covers Angular concepts that I consider basic to using Angular.

Wire up the Delete Button

First, we're going to wire up the delete button. Open up the template and add an ngClick to the button:

```
<div class='dciTestdirective-width100' ng-repeat='object in dataprovider'>
  <span class='dciTestdirective-horizontal-layout-94'>
   {{object[label]}}
  </span>
  <span class='dciTestdirective-horizontal-layout-4' >
    <button ng-click="onDeleteRequest(object)">X</button>
  </span>
</div>
```

When the button is clicked it will execute the onDeleteRequest() function inside the directive's link function. The object passed as a parameter refers to the specific item of the dataProvider defined in the ngRepeat that creates the list.

Inside the link function of the component, create the onDeleteRequest() function:

```
scope.onDeleteRequest = function(object){
 console.log('onDeleteRequest');
}
```

This is a placeholder function. At this moment, you can test code and you should see the onDeleteRequest text output to your browser's console.

Implement the Delete Function

The purpose of this delete button is to remove an item from the dataprovider array. To do this, we will loop over the dataprovider until we find the proper object. We can use the JavaScript splice() function to remove the item from the array.

First, create the loop:

```
for (var counter = 0; counter <= scope.dataprovider.length; counter++){
}
```

Inside the look compare the object parameter with the current element of the dataprovider:

```
if(object === scope.dataprovider[counter]){
    scope.dataprovider.splice(counter, 1);
    break;
}
```

If the passed object and the current dataprovider element are the same object; then use the JavaScript splice() method to remove the item from the dataprovider array. The splice() method takes two parameters. The first is the array index where the removal of items starts. The second value is the number of items to remove. In this case; we remove only a single item.

Play with this Sample!

What's Next

This entry did not introduce you to any new directive-specific concepts. Most likely you've used ngClick on buttons, or other elements, plenty of times in your own Angular development adventures. The approach is the same.

The next section will tackle how to bind an external method to an interaction inside of a directive.

Using Event Handlers

This entry in this series will tell you how to create a directive that allows the user to execute their own function based on directive interactions. This is like creating a custom event inside the directive, and a custom handler outside the directive; although I find the the implementation much simpler. We are going to create a API element that responds to the deletion of an item in the directive.

Create the Event

The first thing we need to do is tell the directive that it has an event handler. This is done in the directive's isolated scope:

```
scope : {
    dataprovider : '=',
    label : '=labelfield',
    itemdeleted : '&'
}
```

The dataprovider and the label were covered earlier in this whitepaper. The itemdeleted property is new. You'll notice its value is an ampersand; not an equal sign. This is a way to tell Angular that a function is expected, not a variable. Similar to the label, the itemdeleted could also specify a different internal name, like this:

```
itemdeleted : '&onitemdelete'
```

For the purposes of this sample, I decided not to use a different name externally verse internally.

Next go to the onDeleteRequest() method inside the directive's link function. This method is executed when the X button in the view is clicked. It loops over the array; and uses the JavaScript splice method to remove the item from the array.

```
scope.onDeleteRequest = function(object){
    for (var counter = 0; counter <= scope.dataprovider.length; counter++){
        if(object === scope.dataprovider[counter]){
            scope.dataprovider.splice(counter, 1);
            scope.itemdeleted();
            break;
        }
    }
}
```

After the splice is executed; the itemdeleted() event handler is called. This is done by referencing the scope value inside the link function, and executing the method on it. If no itemdeleted event handler is defined, this silently fails.

Create the Event Handler

We've modified the directive to execute the event handler, but now you need to know how to use it. Go back to the main HTML file and create a handler method inside the directiveTestController:

```
$scope.onItemDelete = function(){
    $scope.itemDeletedMessage = 'item deleted';
}
```

This method creates a $scope variable named itemDeletedMessage and sets it value to itemDeleted. Let's default that $scope variable while we're in the controller:

```
$scope.itemDeletedMessage = '';
```

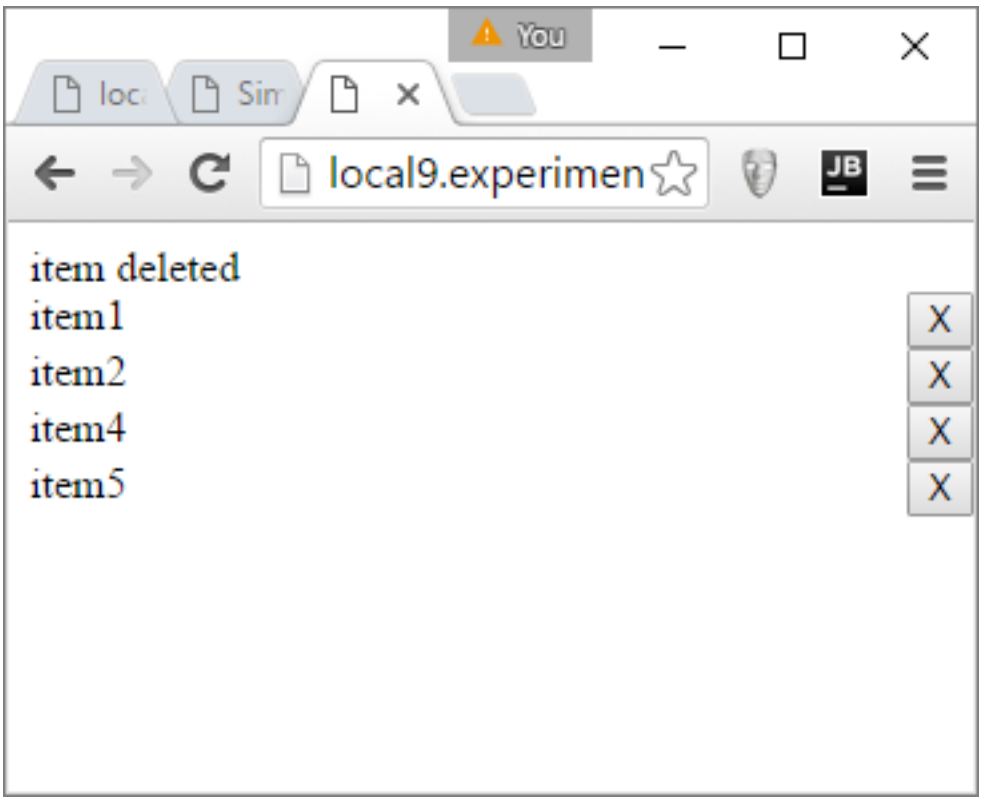
We have the directive all set, and a method to be executed by the directive. Let's put it all together.

In the HTML portion of this sample app, do this:

```
{{itemDeletedMessage}}
<dci-testdirective dataprovider="objectArray" itemdeleted="onItemDelete()">
</dci-testdirective>
```

The scope variable, itemDeletedMessage is shown above the directive. The itemdeleted event handler is created as an attribute on the dciTestdirective. It calls the controller's onItemDelete() scope value.

This is the app with an item deleted:



Play with this sample!

Pass an Argument to the Event Handler

Angular directives also allow us to pass arguments from inside the directive to the event handler method. We'll pass the object that had just been deleted. First, save the object in a variable:

```
var deletedObject = scope.dataprovider[counter];
```

Next, perform the deletion:

```
scope.dataprovider.splice(counter, 1);
```

And finally, perform the handler call:

```
scope.itemdeleted({deletedObject:deletedObject});
```

The value of the itemdeleted method call is an object. Each property on the object represents a parameter that can be passed into the external method. Angular does magic behind the scenes to sync the object of arguments with the actual arguments.

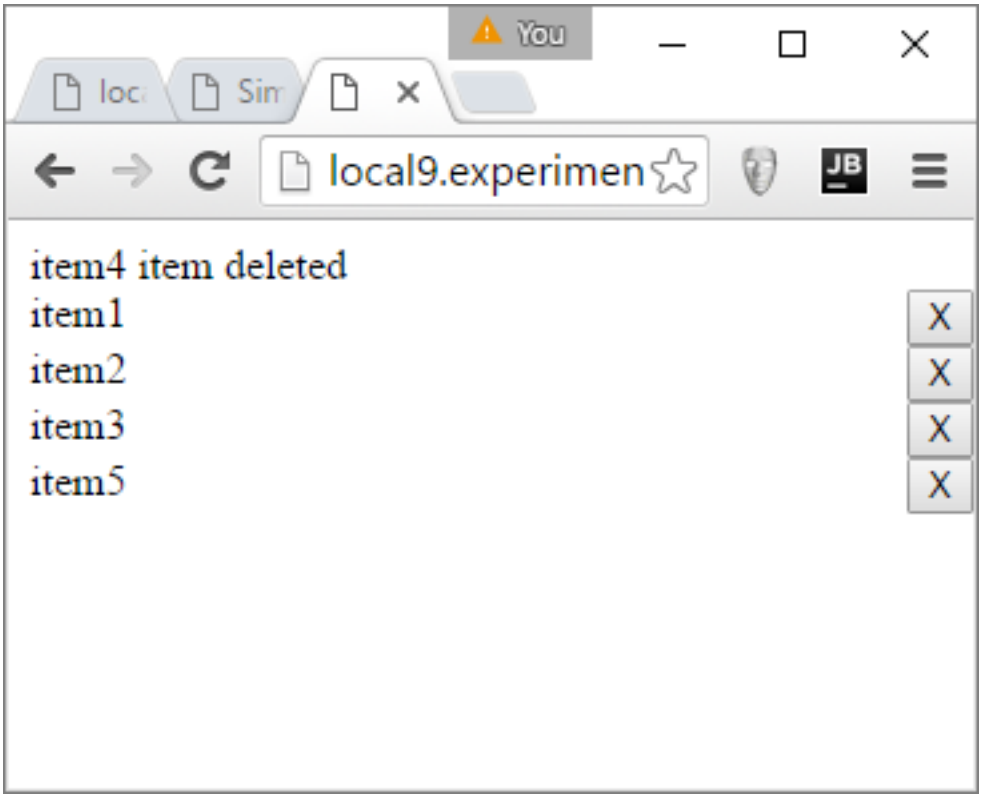
In the HTML directive, add deletedObject as an argument to the itemdeleted function call:

```
<dci-testdirective dataprovider="objectArray"
                   itemdeleted="onItemDelete(deletedObject)">
</dci-testdirective>
```

In the main app's controller, add the argument to the onItemDelete() function::

```
$scope.onItemDelete = function(deletedObject){
    $scope.itemDeletedMessage = deletedObject.itemLabel + ' item deleted';
}
```

The code introspects the deletedObject's label to modify the item deleted message, so the user will know exactly which item has been deleted:



Play with this sample!

What's Next?

So far this whitepaper has covered how to pass data into a directive, how to get data out of a directive, and how to have the directive execute methods outside of the directive. In the next section I'll explain how to execute a method inside the directive.

Call a Method inside a Directive

In this whitepaper discussed how to pass data into a directive. I've shown how to get data out of a directive. We've had the directive execute an external method. Is there a way to trigger a method inside a directive from the outside?

Angular does not inherently provide a way to execute a method inside a directive. There are a few different methods to accomplish this. I am going to demonstrate my preferred work-around. It is a useful approach when you need to do it. This section will create a method inside the directive to delete an item from the list.

Create a Control Wrapper

We previously showed how to use the isolated scope to pass values into a directive. Internal changes to the parameters values will automatically be reflected outside the directive. We'll use this same approach to share internal methods externally.

The method we're going to expose to the directive will be one to delete an item from the dataprovider array. Create this method inside the directive's link function:

```
function deleteItem(object){
    scope.onDeleteRequest(object);
}
```

This function is named deleteItem. It accepts a single argument, representing the item that is going to be deleted, and then calls the onDeleteRequest() method. The onDeleteRequest() method is the listener for the click on the x button inside the directive's view. We're reusing the functionality here.

Now, let's expose this method. First, create a control value in the directive's scope:

```
scope : {
   // other scope variables
   control : '='
},
```

Next go to the directive's onInit() method. Initialize the control parameter:

```
scope.internalControl = scope.control || {};
```

If no control is passed in, then the internalControl turns into an empty object. Otherwise, we now have a link to the control object from both inside and outside the directive.

Next add the deleteItem() function into the internalControl:

```
scope.internalControl.deleteItem = deleteItem;
```

This completes the changes we need to make to the directive. Let's move onto the main application.

Use the Control Wrapper

Inside the controller for the main app, create our own control object:

```
$scope.dciTestdirectiveControl = {};
```

Inside the HTML, you can pass this object into the directive:

```
<dci-testdirective dataprovider="objectArray"
                   itemdeleted="onItemDelete(deletedObject)"
                   control="dciTestdirectiveControl">
</dci-testdirective>
```

This connects the external control wrapper to the inner control wrapper. The HTML UI will use a button to trigger the method inside the directive.

Add a button under the directive:

```
<button ng-click="onDeleteFirstItemRequest()">Delete First Item</button>
```

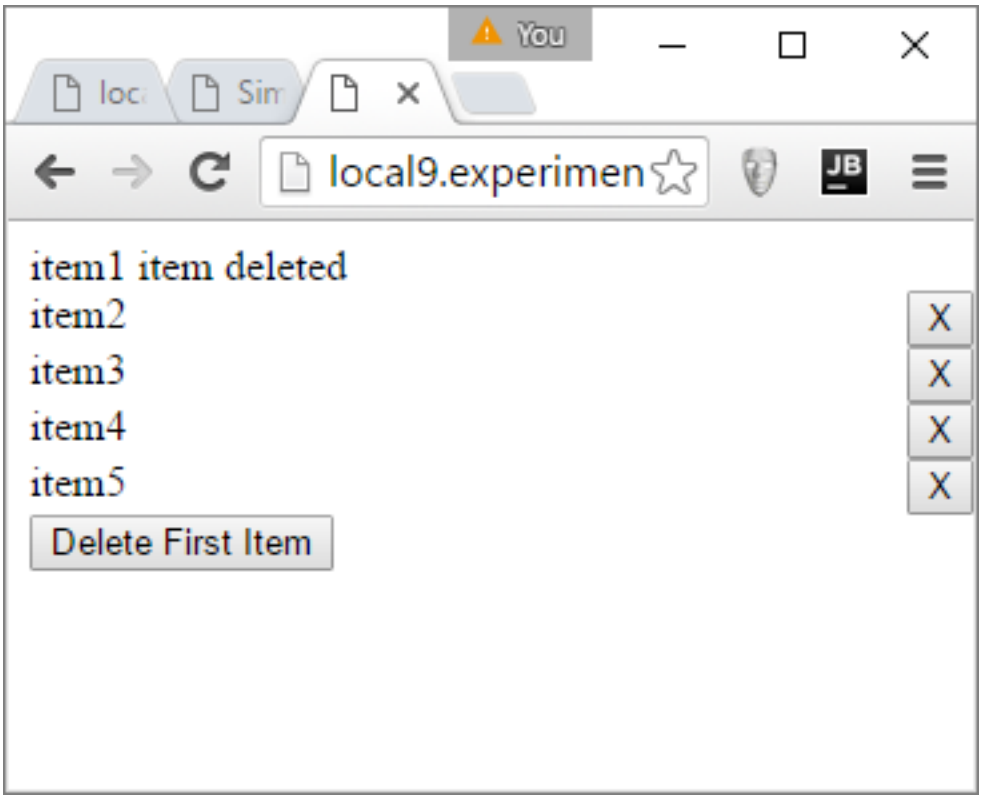
For the sake of this example, we'll always delete the first item in the list.

In the controller's scope, implement the onDeleteFirstItemRequest() method:

```
$scope.onDeleteFirstItemRequest = function(){
    $scope.dciTestdirectiveControl.deleteItem($scope.objectArray[0]);
}
```

This method references the dciTestdirectiveControl wrapper and executes the deleteItem() method on it, passing in the first item in the dataprovider array.

Run the app, click the button and you'll see the first item was removed:



Play with this sample!

What's Next

I have one last topic to cover in this whitepaper about directives. I'm going to show you some things you can do to prepare your directive for reuse between different applications.

Preparing a Directive for Reuse

This article is going to focus on some things you can do to modify your directive for reuse between multiple Angular applications. We've already done some work to make the directive reusable, such as giving the directive an encapsulated scope. This section will focus on making the directive modular.

Restrict

The restrict property defines how a directive can be used. There are four different ways that Angular can find a directive:

- **A**: If the restrict attribute contains an A, then the directive can be used as an attribute name. It would be used like this:

```
<div dci-testdirective dataprovider="objectArray"></div>
```

- **E**: If the restrict attribute contains an E, then the directive can be used as its own element. This is what we've been doing in this whole article series:

```
<dci-testdirective dataprovider="objectArray" ></dci-testdirective>
```

This is my preferred method for using directives.

- **C**: If the restrict property contains a C, then that directive can be used as part of a CSS class:

```
<div class="dci-testdirective" dataprovider="objectArray"></div>
```

- **M**: If the restrict property specifies M that means the directive can be used as an HTML Comment.

```
<!-- directive: dci-testdirective expression -->
```

When using a directive as an HTML comment, there is no easy way to pass in values. The expression mentioned above will be accessible inside the link function as part of the attrs parameter; but the result must be a string; it is not easy to pass in complicated values such as arrays or other dynamic values. Comment directive are the least common and I believe this is why.

If you want your directive to be available in all methods, you can do this:

```
restrict : 'AECM',
```

For most practical purposes, I'll do this:

```
restrict : 'AE',
```

This makes the directive available as an attribute and as an element.

Put Directive in its own Module

The next thing I want to cover is how to put the directive into its own module. This, basically, gives the directive its own space in the world of Angular applications. That space can be passed around to other modules that need to use it.

First, let's review how the directive is currently defined:

```
angular.module('directiveTestApp').directive('dciTestdirective',function(){
});
```

The directive is created using the directive function of an Angular module. The angular module is in this case refers to the main application used throughout this series. We're going to take that directive out of the main application's module and put it in one of its own:

```
angular.module('dciDirectiveCollection',[]).directive('dciTestdirective',
  function(){
});
```

With the directive in its own module, the original application will have no way to access it by default. We need to tell the original a module to reference the new directive-specific module. This is done by adding the directive-specific module as an argument to the main module's definition:

```
angular.module('directiveTestApp',['dciDirectiveCollection']);
```

The directiveTestApp module now has an argument in its options array, referring to the module which contains the directive we created in this whitepaper. The HTML code does not need to change.

Play with the final app.

By implementing directives in their own module, this increases its reusability. With a js include and adding the options argument the directive can be made available to our full application.

Final Thoughts

This series of articles covered everything you need to know, and more, to get started using directives within your Angular applications. Still, I didn't cover everything. Here are a few areas that you can look into for further study:

- **Controllers**: Directives can have controllers. These are often used for communication between nested directives, but could also be used similar to the link function.
- **Transclusion**: An Angular directive can wrap HTML. This is useful because the wrapped HTML is treated as part of the source controller, not part of the directive. It can access scope variables and methods from the controller even though visually it is displayed as part of the directive.

Such topics will have to wait for another day. Thanks for reading!