



- [Sign in](#)

[Dhananjay Kumar](#) » [Understanding scopes in AngularJS custom Directives](#)

Understanding scopes in AngularJS custom Directives

- [Tweet](#)



In this post we will learn about different kinds of scopes in AngularJS custom directives. First we'll start with a high level introduction of directives and then focus on scopes.

Directives

Directives are one of the most important components of AngularJS 1.X, and have the following purposes:

1. Gives special meaning to the existing element
2. Creates a new element
3. Manipulates the DOM

Beyond ng-app, ng-controller, and ng-repeat, there are plenty of built-in directives that come with AngularJS, including:

- ng-maxlength
- ng-minlength
- ng-pattern
- ng-required
- ng-submit
- ng-blur
- ng-change
- ng-checked
- ng-click
- ng-mouse
- ng-bind
- ng-href
- ng-init
- ng-model
- ng-src
- ng-style
- ng-app
- ng-controller
- ng-disabled
- ng-cloak
- ng-hide
- ng-if
- ng-repeat
- ng-show
- ng-switch

- ng-view

Mainly, directives perform either of the following tasks:

- Manipulate DOM
- Iterate through data
- Handle events
- Modify CSS
- Validate data
- Data Binding

Even though there are many built-in directives provided by the Angular team, there are times when you might need to create your own custom directives. A custom directive can be created either as an element, attribute, comment or class. In this post, a very simple custom directive can be created as shown in the listing below:

```
MyApp.directive('helloWorld', function () {  
    return {  
        template: "Hello IG"  
    };  
});
```

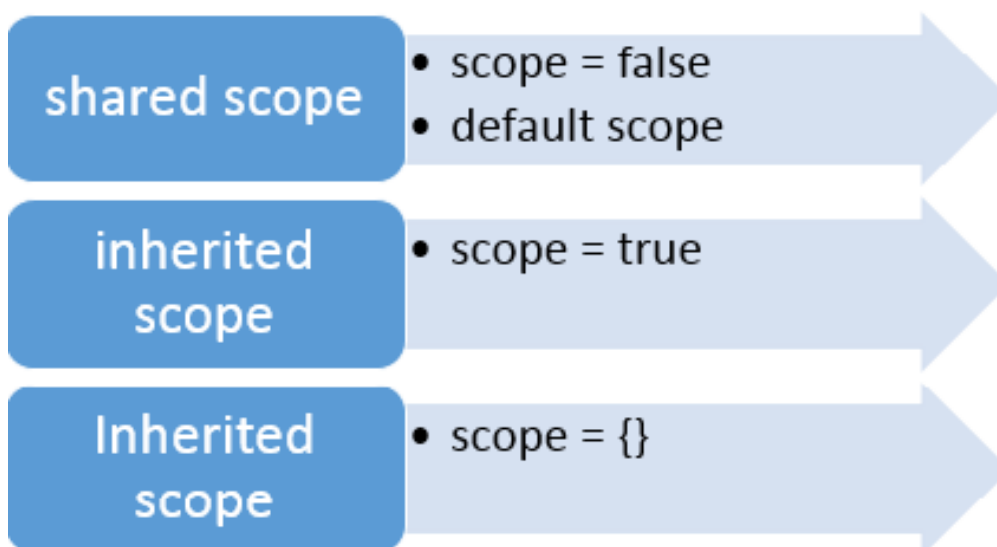
While creating custom directives, it's important to remember:

- The directive name must be in the camel case;
- On the view, the directive can be used by separating the camel case name by using a dash, colon, underscore, or a combination of these.

Scopes in Custom Directives

Scopes enter the scene when we pass data to custom directives. There are three types of scopes:

1. Shared scope
2. Inherited scope
3. Isolated scope



We can create a custom directive with an inherited scope by setting the scope property to true as shown in the listing below:

```

MyApp.directive('studentDirective', function () {
    return {
        template: "
        {{student.name}} is {{student.age}} years old !!
        ",
        replace: true,
        restrict: 'E',
        scope : true ,
        controller: function ($scope) {
            console.log($scope);
        }
    }
});

```

Shared and Inherited Scope

Shared scope and inherited scope are relatively easier to understand. In a shared scope, directives share the scope with the enclosed controller.

Let us assume that we have a controller as shown in the listing below:

```

MyApp.controller('StudentController', ['$scope', function ($scope) {
    console.log($scope);
    $scope.student = {
        name: "dj",
        age: 32,
        subject: [
            "math",
            "geography"
        ]
    }

    $scope.setGrade = function (student) {
        student.grade = "A+"
    }

}]);

```

Next let's go ahead and create a custom directive as shown in the listing below:

```

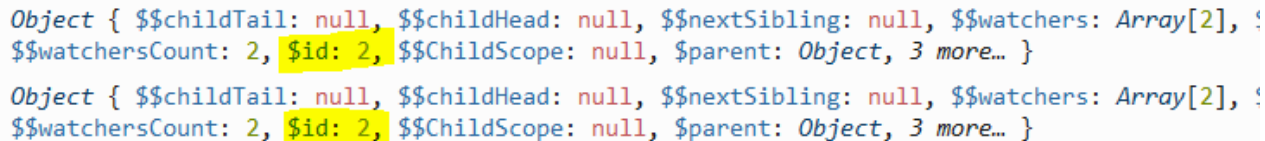
MyApp.directive('studentDirective', function () {
    return {
        template: "
        {{student.name}} is {{student.age}} years old !!
        ",
        replace: true,
        restrict: 'E',
        controller: function ($scope) {
            console.log($scope);
        }
    }
});

```

Here we can use the studentdirective on the view as shown in the listing below:

```
<div ng-controller="StudentController">
    <student-directive> student-directive>
</div>
```

In the above snippet we are using the directive inside the div, where the ng-controller directive is set to StudentsController. Since we have not set any value for the scope property in the directive, by default that works in the shared scope mode. Directives can access the properties attached to the controller scope. Any changes to the properties in the directive would be reflected to the controller and vice versa. You'll also notice that I'm printing the scope id for both the controller and the directive, and both id's must be the same.



```
Object { $$childTail: null, $$childHead: null, $$nextSibling: null, $$watchers: Array[2], $$watchersCount: 2, $id: 2, $$ChildScope: null, $parent: Object, 3 more... }
Object { $$childTail: null, $$childHead: null, $$nextSibling: null, $$watchers: Array[2], $$watchersCount: 2, $id: 2, $$ChildScope: null, $parent: Object, 3 more... }
```

There is one problem with the shared scope: we cannot pass data explicitly to the directive, the directive directly takes the data from the enclosed controller.

In the inherited scope, the directive inherits the scope of the controller. Let's take a look at how to create a directive with inherited scope below:

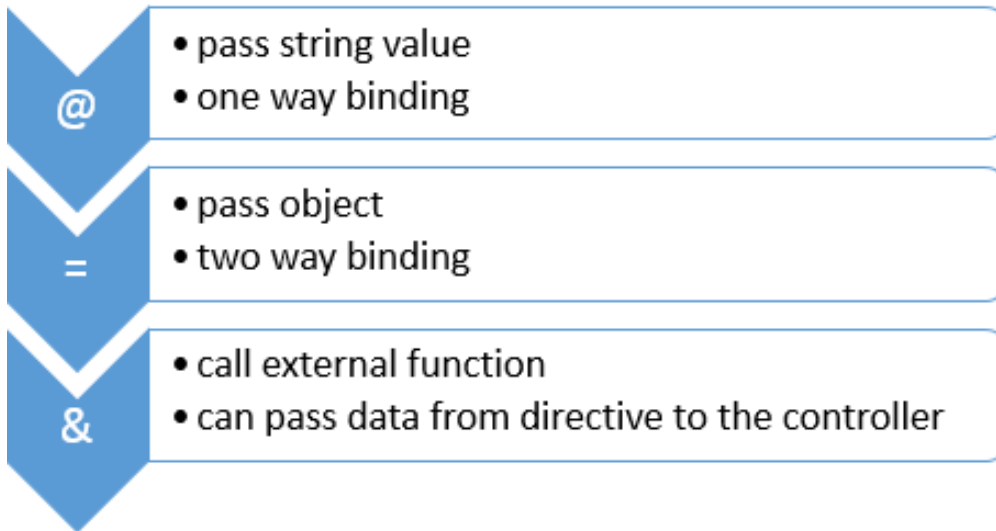
```
MyApp.directive('studentDirective', function () {
    return {
        template: "
        {{student.name}} is {{student.age}} years old !!
        ",
        replace: true,
        restrict: 'E',
        scope : true ,
        controller: function ($scope) {
            console.log($scope);
        }
    }
});
```

Inherited scope is very useful in nested custom directives.

Isolated Scope

In Isolated scope, the directive does not share a scope with the controller; both directive and controller have their own scope. However, data can be passed to the directive scope in three possible ways.

1. Data can be passed as a string using the @ string literal
2. Data can be passed as an object using the = string literal
3. Data can be passed as a function the & string literal



Isolated scope is very important because it allows us to pass different data to the controller. To understand it better, let's assume that we have a controller as listed below:

```
MyApp.controller("ProductController", function ($scope) {  
    $scope.product1 = {  
        name: 'Phone',  
        price: '100',  
        stock: true  
    };  
    $scope.product2 = {  
        name: 'TV',  
        price: '1000',  
        stock: false  
    };  
    $scope.product3 = {  
        name: 'Laptop',  
        price: '800',  
        stock: false  
    };  
  
    $scope.ShowData = function () {  
        alert("Display Data");  
    }  
});
```

As you can see here, we have three different products and we want to pass it differently.

Pass data as a string

In the isolated scope, we can pass data as a string using the **@ string literal**. We can create a custom directive which will accept the string as an input parameter as shown in the listing below:

```
MyApp.directive('inventoryProduct', function () {  
    return {  
        restrict: 'E',
```

```

    scope: {
      name: '@',
      price: '@'
    },
    template: '
    {{name}} costs {{price}} $
    Change name
    '
  };
});

```

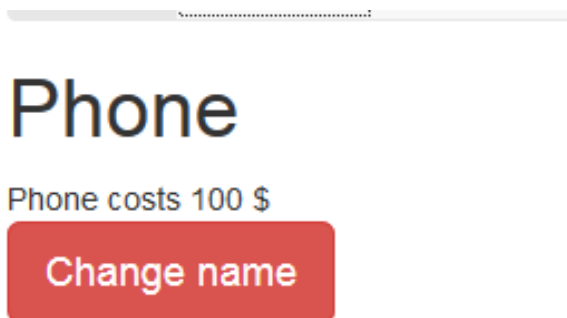
In the above listing, we are passing two string parameters using the `@` literal, meaning a string will be passed in the name and price variable. The directive can be used on the view as shown in the listing below, where you'll see we are passing a string value for name and the price in the directive.

```

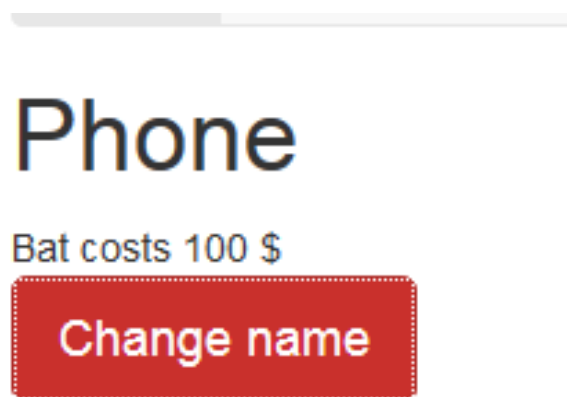
<div ng-controller="ProductController">
  <h1>{{product1.name}}</h1>
  <inventory-product name="{{product1.name}}" price="{{product1.price}}">inventory-product</div>

```

On running the application, we should be able to see name and the price of product1.



When we click on the Change name button, only the name of the directive will be changed and ProductController product1 object would not be affected due to the isolated scope.



Also keep in mind that when we pass data in isolated scope as a string, it gets passed in a one-way manner, so any change in the controller scope will be reflected in the directive. However, a change in the directive would not be reflected in the controller.

Pass data as an object

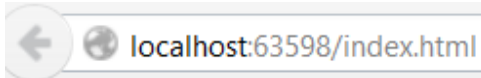
In isolated scope we can pass data as an object using the = **string literal**. We can create a custom directive which will accept object as input parameter as shown in the listing below:

```
MyApp.directive('inventoryProduct', function () {  
    return {  
        restrict: 'E',  
        scope: {  
            data: '='  
        },  
        template: '  
{{data.name}} costs {{data.price}} $  
Change name  
'  
    };  
});
```

In the above listing we are passing one object parameter using the = literal. Here the object will be passed in the data variable. Directive can be used on the view as shown in the listing below. As we see we are passing an object value for a data variable in the directive.

```
<div ng-controller="ProductController">  
    <h1>{{product1.name}}</h1>  
    <inventory-product data="product1">inventory-product</inventory-product>  
    <h1>{{product2.name}}</h1>  
    <inventory-product data="product2">inventory-product</inventory-product>  
    <h1>{{product3.name}}</h1>  
    <inventory-product data="product3">inventory-product</inventory-product>  
</div>
```

In the above listing we are using the directives three times and passing three distinct objects as input. On running, we'll see the output as shown below:

localhost:63598/index.html

Phone costs 100 \$

Change name

TV costs 1000 \$

Change name

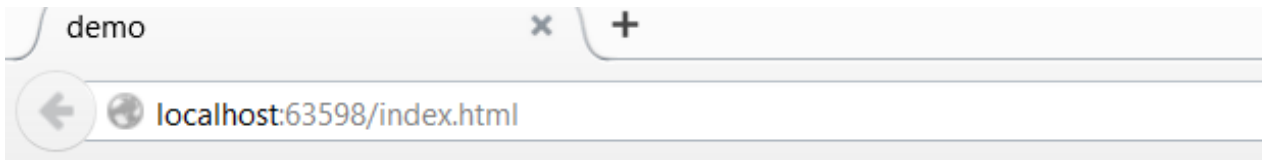
Laptop costs 800 \$

Change name

Passing the object in the isolated scope works in two way binding mode, meaning any changes in the directive would be reflected to the enclosed controller. Let us say we pass product1 twice as shown in the listing below:

```
<div ng-controller="ProductController">
  <inventory-product data="product1">inventory-product<
  <inventory-product data="product1">inventory-product<
</div>
```

When we run the application, the same object will be passed to both instances of the directive.



Phone costs 100 \$

Change name

Phone costs 100 \$

Change name

Clicking any of the **Change name** buttons will change the name of both the instance of the directives because same object is passed, and passing an object works in two-way mode. Any change in the directive would be reflected to the enclosing controller and vice versa.



IG costs 100 \$

Change name

IG costs 100 \$

Change name

Calling an external function

We can call an external function in the enclosed directive using the literal variable &. As we see with the ProductController, there is a ShowData() function. This function can be called in a custom directive by

modifying the directive as shown below:

```
MyApp.directive('inventoryProduct', function () {  
    return {  
        restrict: 'E',  
        scope: {  
            data: '&',  
        },  
        template: '  

```

{{data.name}} costs {{data.price}} \$

Change name

```
',  
    };  
});
```

Here the directive can be used on the view as shown here:

```
<div ng-controller="ProductController">  
    <inventory-product data="ShowData()">inventory-product<  
</div>
```

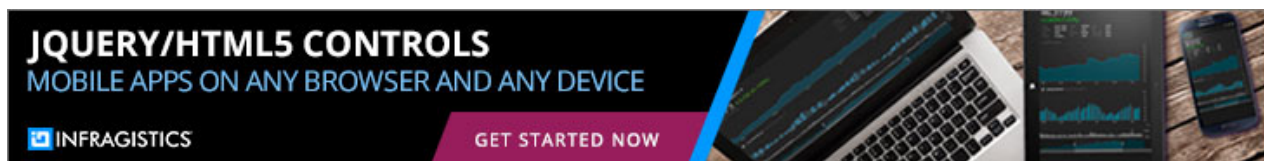
Conclusion

In this post, we started with a basic understanding of directives and then moved to scopes, where we learned that:

1. Shared scope: directive and controllers share the scope and the data. We cannot pass data explicitly to the directive.
2. Inherited scope: directive inherits the scope of the controller. We cannot pass data explicitly to the directive.
3. Isolated scope: directive and controllers don't share the data and scope. We can pass data either as a string or an object explicitly to the directive.

I hope you find this post useful. Thanks for reading!

Looking for an advanced set of HTML5 & JavaScript UI controls and components? [Download IgniteUI now](#) and see what it can do for you!



- [Tweet](#)

-  1

Tags / [news](#), [javascript](#), [AngularJS](#), [scopes in AngularJS custom directives](#), [directives](#)