DIRECTIVES

AngularJs-Directives

Directives are AngularJS way of extending HTML. As the AngularJS Official documentation says, they are basically markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell AngularJS's HTML compiler ($compile) to attach a specified behavior to that DOM element (e.g. via event listeners).

- Basic introduction on Directives
- Built-in Directives
- Creating a custom directive
  - Directive Definition Object
    - **template, templateUrl** : Reusing HTML snippets
    - **restrict** [Defines How the directive can be used in HTML]
    - **scope** [Isolating the scope of a Directive]
    - **link** [Defines directive's API for business logic]
    - **replace** [To replace the container directive is applied on]
    - **transclude, ng-transclude** [Used to preserve any content added as child of directive]
    - **Directive's Communication using require, link function & directive's controller**

Directive's typical usage

Directive are the preferred way to go if we have any of the below mentioned requirements.

- DOM manipulation
- Reusing HTML snippets
- Modifying the behavior of existing elements
- Integrating with third-party components

Type of Directives

AngularJS directives are created to be used as : attribute, element name, comment or CSS class. Recommended ways are as attribute & element. We will discuss this in deep details in Directive's restrict option further down in this tutorial.

Built-in Directives

AngularJS provides several built-in directives like ng-app, ng-model, ng-repeat, ng-show, ng-class etc. We have already seen quite a few of them in previous posts. Full list is available at AngularJS Built-in Directives.

Custom Directives

Quite often we will come to the point where built-in directives are simply not enough for our needs. In those cases, we can create custom directives. We will see how to create them in detail in this guide.

Creating a Custom Directive

Directives are created using AngularJS module's `directive function`.

```
angular.module('myApp', [])
.directive('itemWidget', [function() {
   return { //returns Directive Definition Object.
      // Directive definition goes here.
   };
}]);
```

First argument to the directive function is directive's name itself. Second parameter is a factory function which is used to setup & configure our directive. This function returns an object known as **Directive Definition Object**. We will see the content of this object in a moment.
Once declared as above, the directive can then be used in HTML. On every use of our directive in HTML, AngularJS will look into this definition object.

By default, directive can only be used as an attribute of existing element in HTML, for example <div item-widget>. But with the help of some extra configuration (using Directive's restrict option), we can make it to be used as independent element, like <item-widget>. We will come to that later in this tutorial.

A word on Directive Naming in HTML

AngularJS **normalizes** the name of a directive in HTML to match in JavaScript. It strips out certain prefix if they are present like [x-, data-] from directive names , and then converts (- , :, _) separated names to **camelCase** to match the name in javaScript. This means item-widget, Item:widget, data-item_widget or x-item-widget in HTML will all match to **itemWidget** in javaScript.

Directive Definition Object

Directive definition object contains several key:value pairs. We will discuss each one of them in detail in this guide, starting from template/templateUrl. Based on your particular requirement, you may decide to use several key:value pairs together.

template, templateUrl : Reusing HTML snippets

If the directive has any content that needs to be inserted when the directive is encountered, that can be done using **template** and **templateUrl** keys of the directive definition object. If the content is small, **template** key can be used, whereas if the content is large, **templateUrl** is preferable which points to the file containing actual content of directive. template & templateUrl are particularly handy for the scenario where you have some HTML that needs to be reused on multiple places in your application. Creating a directive with that content and then just simply using that directive is a nice way of HTML reuse.

Using templateUrl

```
<html>
    <head>
        <title>Directive Demo</title>
    </head>
    <body class="jumbotron container" ng-app="myApp">

        <div ng-controller="AppController as ctrl">
            <h3>List of Sale Items</h3>
            <div ng-repeat="item in ctrl.items">
                <div item-widget></div>
            </div>
        </div>

      <script>
            angular.module('myApp', [])
            .controller('AppController', [function() {
              var self = this;
              self.items = [
                                {name: 'Computer', price: 500, condition:'New',brand
: 'Lenovo', published:'01/11/2015'},
                                {name: 'Phone', price: 200, condition:'New',brand :
'Samsung', published:'02/11/2015'},
                                {name: 'Printer', price: 300, condition:'New',brand :
'Brother', published:'06/11/2015'},
                                {name: 'Dishwasher', price: 250, condition:'Second-
Hand',brand : 'WhirlPool', published:'01/12/2015'},
                                ];
            }])
            .directive('itemWidget', [function() {
                return{
                    templateUrl:'saleItem.html'
                }
            }]);
```

```
  </script>
  </body>
</html>
```

In above example, we have declared a simple directive with bare-minimum configuration. We are using only templateUrl key which points to a file [saleItem.html] containing content of directive. Then we have used this directive as an attribute of a div element within ng-repeat loop, passing items from Controller.

Below shown is the directive's content[saleItem.html].

```
<div class="panel panel-default">
    <div class="panel-heading">
        Published at:<span ng-bind="item.published | date"></span>
    </div>
    <div class="panel-body">
            Name:<span ng-bind="item.name"></span>
            Condition:<span ng-bind="item.condition"></span>
            Price:<span ng-bind="item.price | currency"></span>
            Brand:<span ng-bind="item.brand"></span>
    </div>
</div>
```

This is a nice example of HTML reuse. Throw some CSS and you can now make your own amazon/e-bay like item list.

Using template

As mentioned above, if the directive's content is not large, you may prefer to inline the directive content using template key instead of templateUrl. Let's rewrite above example using template this time.

```
<html>

<head>
    <title>Directive Demo</title>
</head>

<body class="jumbotron container" ng-app="myApp">

    <div ng-controller="AppController as ctrl">
        <h3>List of Sale Items</h3>
        <div ng-repeat="item in ctrl.items">
            <div item-widget></div>
        </div>
    </div>
```

```
    <script>
        angular.module('myApp', [])
            .controller('AppController', [function () {
                var self = this;
                self.items = [
                    { name: 'Computer', price: 500, condition: 'New', brand:
'Lenovo', published: '01/11/2015' },
                    { name: 'Phone', price: 200, condition: 'New', brand:
'Samsung', published: '02/11/2015' },
                    { name: 'Printer', price: 300, condition: 'New', brand:
'Brother', published: '06/11/2015' },
                    { name: 'Dishwasher', price: 250, condition: 'Second-Hand',
brand: 'WhirlPool', published: '01/12/2015' },
                ];
            }])
            .directive('itemWidget', [function () {
                return {
                    template: '<div class="panel panel-default">' +
                        '<div class="panel-heading">' +
                        'Published at:<span ng-bind="item.published |
date"></span>' +
                        '</div>' +
                        '<div class="panel-body">' +
                        'Name:<span ng-bind="item.name"></span>' +
                        'Condition:<span ng-bind="item.condition"></span>' +
                        'Price:<span ng-bind="item.price | currency"></span>' +
                        'Brand:<span ng-bind="item.brand"></span>' +
                        '</div>' +
                        '</div>'
                }
            }]);
    </script>
</body>

</html>
```

As you can see, even if it does the job, in-lining large content with template key can easily become messy. First approach seems better.

That's it for template/templateUrl configuration. Let's move to next configuration option [`restrict`] of Directive Definition Object, which helps us to decide how the directive can be used in HTML.

AngularJS Custom-Directives restrict option guide

AngularJS Directive's **restrict** key defines how a directive can be used in HTML. In this post, we will see all possible configuration options using restrict.

In previous post, our directive was created to be used as an attribute of an existing element, like **<div item-widget>**which is the default behavior. But We can choose the way our directive should be used in HTML, using **restrict** key of Directive Definition Object. The possible values for restrict (and so the ways in which we can use our directive) are:

- **A** : Specifies that Directive will be used as an attribute, like **<div item-widget> **, as was done in last example. This is also the default behavior if restrict is not declared. Preferred if just modifying the behavior of existing elements.
- **E** : Specifies that Directive will be used as an Element. Like **<item-widget></item-widget>**. Preferred if creating new content.
- **C** : Specifies that Directive can be used as class name in existing HTML elements. Like ** **
- **M** : Specifies that Directive can be used as HTML comments. Like **<!– Using directive: item-widget–>**

These keys can be combined. Best practices suggest to use only A & E types.

Now if in above example, we were to use item-widget as an Element, we would just declare restrict key with following values:

```
.directive('itemWidget', [function () {
    return {
        templateUrl: 'saleItem.html',
        restrict: 'E'
    }
}]);
```

Do note that once you declare **restrict** key, you are fixing it. That means if you will try to use item-widget as attribute, AngularJS will throw an error. If we want to allow item-widget to use both as Element & Attribute, we can do so using following configuration:

```
.directive('itemWidget', [function () {
    return {
        templateUrl: 'saleItem.html',
        restrict: 'EA'
    }
}]);
```

Thanks to above configuration, now our directive can be used both as <item-widget></item-widget> & <div item-widget> . No additional changes.

Complete Code:

```
<html>
    <head>
        <title>Directive Demo</title>
    </head>
    <body class="jumbotron container" ng-app="myApp">
        <div ng-controller="AppController as ctrl">
            <h3>List of Sale Items</h3>
            <div ng-repeat="item in ctrl.items">
                <item-widget></item-widget>
            </div>
        </div>
      <script>
            angular.module('myApp', [])
            .controller('AppController', [function() {
              var self = this;
              self.items = [
                 {name: 'Computer', price: 500, condition:'New',brand : 'Lenovo',
published:'01/11/2015'},
                  {name: 'Phone', price: 200, condition:'New',brand : 'Samsung',
published:'02/11/2015'},
                   {name: 'Printer', price: 300, condition:'New',brand :
'Brother', published:'06/11/2015'},
                  {name: 'Dishwasher', price: 250, condition:'Second-Hand',brand
: 'WhirlPool', published:'01/12/2015'},
                            ];
            }])
            .directive('itemWidget', [function() {
                return{
                    templateUrl:'saleItem.html',
                    restrict: 'EA'
                }
            }]);
  </script>
      </body>
</html>
```

AngularJS Custom-Directives scope guide

AngularJS Directive's **scope** key provides us complete control over the scope of our directive element. Our goal while writing a directive should be to avoid polluting parent scope as much as possible. In this post we will discuss various scope related options/strategies including using parent scope, inheriting parent scope and creating an `Isolated scope`[isolating directive's inner scope from parent scope].

Isolated scope

Isolated Scope in particular is very interesting in the sense that by creating Isolated scope, directive's inner scope can be separated from outer (parent) scope. Directive's scope does not inherit anything from parent scope. This decoupling from parent scope is important as then directive becomes independent and can be used in different situations (reusable) without relying on specific properties/functions of parent. If directive needs some input from parent scope, that can be passed by mapping directive's inner-scope to outer-scope [with help of attributes e.g.]

```
<script src="angular.min.js"></script>
<script>
    var myApp = angular.module("myApp", []);

    myApp.directive("myDirective", function () {
        return {
            restrict: "E",
            scope: {
                //Its value (=itemInfo) tells AngularJS $compile to bind to the
item-info attribute [Normalized name as for directives] in HTML.
                item: "=iteminfo"
                //item:"=" this case you can use direct name like item in <my-
directive=item="bla"
            },
            template: "<span ng-bind='item'></span>"
        }
    });

    myApp.controller("myController", function ($scope) {
        $scope.name = "chandra";
        $scope.items = ["apple", "orange", "banana"];
        $scope.movies = ['Ice Age', 'Frozen'];
    })
</script>
```

```
<body ng-app="myApp">
    <div ng-controller="myController">
        <div ng-repeat="bla in items">
            <my-directive iteminfo="bla">
            </my-directive>
        </div>
    </div>
</body>
```

Scope provides much more...

Directive's scope key provides much more than mentioned above. In general, scope key can have following values:

- **false** [ex. scope : false or 'no scope key defined'] Specifies that **directive's scope is SAME AS parent scope**. There is no child scope. Every variables and functions from parent is available to directive and any modifications made by directive are reflected in parent. This is the default behavior.This option is not recommended.

- **true** [ex. scope : true] Specifies that directive's scope inherits the parent scope, but **creates a CHILD scope of it's own**. Every variables and functions from parent is available to directive, but any modifications made by directive are not reflected in parent. This option is recommended in the sense that you will not be polluting parent scope. But do note that the moment you made a modification from inside child scope on a primitive variable which was inherited from parent scope, that particular variable's inheritance will be totally disconnected from parent scope, and any further changes on that variable done inside parent scope will NOT be available in child scope.

- **Object** [ex. scope : {}] scope option can be passed an object. This triggers AngularJS to create an Isolated scope. With Isolated scope, directive does not inherit anything from parent. Any data that parent scope needs to share with this directive needs to be passed in using attributes. This is the best approach to create reusable & independent directives. The object passed in here, contains key that refers to directive's **attributes** in HTML, and the **types of values** that will be passed in to the directive. Essentially, We can specify three types of values that can be passed in, which AngularJS will directly put on the scope of the directive:
  - = : Specifies that value passed in directive's attribute in HTML is an Object. It is also known as bi-directional or two-way data binding. This will be bound to

directive scope and any changes done in attribute value in outer scope will be available in the directive.

- **@** : Specifies that value passed in directive's attribute in HTML is a string, which may contain AngularJS binding expressions ({{ }}). The calculated value(if binding expression is involved) will be assigned to the directive's scope and any changes in the value will also be available in the directive.

- **&** : Specifies that value passed in directive's attribute in HTML is a function in some controller. The directive can then trigger the function to fulfill it's job.

Digging scope:object configuration

Let's see the object scenario [**scope : {}**] in detail with all types of options with help of an example. [Open your browser console log to see the interaction in action.]

```
.directive('itemWidget', [function() {

            return{
                restrict: 'E',
                scope: {
                    item: '=',
                    promo: '@',
                    pickMe : '&onSelect'
                },
                templateUrl: 'saleItem3.html'
            }
        }]);
```

First key:value pair in scope is **item : '='**. That means in HTML, the attribute name is **item** and it refers to an object which is JSON [since it is defined as '='], passed from parent scope.

Second key:value pair in scope is **promo : '@'**. That means in HTML, the attribute name is **promo** and it refers to a String [since it is defined as '@'], can be passed from parent scope.

Third key:value pair in scope is **pickMe : '&onSelect'**. That means in HTML, the attribute name is **on-select** and it refers to a function [since it is defined as '&'] from a specific controller.

```
<script src="angular.min.js"></script>
<script>
    var myApp = angular.module("myApp", []);

    myApp.directive("myDirective", function () {
        return {
            restrict: "E",
            scope: {
                pickMe:'&onSelect',
                fname:'@first',
                item:'='
            },
            template: "<span ng-click='pickMe({selectedItem:item})'>{{item}}
{{fname}}</span>"
        }
    });

    myApp.controller("myController", function ($scope) {
        $scope.name = "chandra";
        $scope.items = ["apple", "orange", "banana"];
        $scope.movies = ['Ice Age', 'Frozen'];
        $scope.printName = function(name){
            console.log(name);
        }
    })
</script>

<body ng-app="myApp">
    <div ng-controller="myController">
        <div ng-repeat="bla in items">
            <my-directive item="bla" first="chandra" on-
select="printName(selectedItem)">
            </my-directive>
        </div>
    </div>
</body>
```

1. Here we are passing "bla" to "item" that will match with item in directive scope..and that will print in directive template
2. we are sending one string called first="chandra"...the first variable will map in directive and assigned to fname...and finally prints at template directive.
3. we are using on-select this will match in directive scope and assigned to pickMe...if we click on template span it will call pickMe and it contains printName(selectedItem) in ctrl.

AngularJS Custom-Directives link-function guide

AngularJS Directive's link key defines link function for the directive. Precisely, using link function, we can define directive's API & functions that can then be used by directive to perform some business logic. The link function is also responsible for registering DOM listeners as well as updating the DOM. It is executed after the template has been cloned.

link function gets executed for each instance of directive so that each instance gets it's own business-logic without affecting others. If we need to add functionality to our instance of the directive, we can add it to the scope of the element we're working with. AngularJS passes several arguments to the link function, which looks like following:

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

scope : It is the scope of the element the directive is applied on. Note that scope is not injected but passed in.

element : It is the DOM element in HTML the directive is applied on.

attrs : These are the list of attributes as string available on the element in HTML.

controller : It specifies the directive's required controller instance(s) or its own controller (if any). The exact value depends on the directive's require property. That controller itself is defined in directive function using `require` key.
transcludeFn : It is a transclude linking function pre-bound to the correct transclusion scope.
In this post, we will discuss the basic usages of link function. Advanced usage will be discussed in detail in post Directive's Controller & require option.

```
<script src="angular.min.js"></script>
<script>
    var myApp = angular.module("myApp", []);

    myApp.directive("myDirective", function () {
        return {
            restrict: "E",
            scope: {
                pickMe:'&onSelect',
                fname:'@first',
                item:'='
            },
```

```
            template: "<span ng-click='pickMe({selectedItem:item})'>{{item}}
{{fname}}</span>",
            link:function(scope,element,attrs){
                element.on("click",function(event){
                    element.html('thanks for click');
                    element.css({
                        color:"green"
                    })
                })
            }
        }
    });

    myApp.controller("myController", function ($scope) {
        $scope.name = "chandra";
        $scope.items = ["apple", "orange", "banana"];
        $scope.movies = ['Ice Age', 'Frozen'];
        $scope.printName = function(name){
            console.log(name);
        }
    })
</script>

<body ng-app="myApp">
    <div ng-controller="myController">
        <div ng-repeat="bla in items">
            <my-directive item="bla" first="chandra" on-
select="printName(selectedItem)">
            </my-directive>
        </div>
    </div>
</body>
```

This was a trivial example on using link-function. You can do a lot with link function knowing that each element get's it's business logic executed independent of others.

Advanced Usage

Advanced usage of link function involves the other parameters including controller & transcludeFn. transcludeFn is used for more exotic situations while controller is a medium for inter-directive communication. Post Directives controllers & require option discusses them in detail.

That's it for link option. Let's move to next configuration option [replace] of Directive Definition Object, which can be used to replace the container element altogether by directive's actual content HTML.

AngularJS Custom-Directives replace option guide

AngularJS Directive's **replace** option can be used to replace the container element itself by directive content. By default, the directive content inserted as the child of the element directive is applied on. But using **replace**, that container element altogether can be replaced by directive's actual content HTML.

By default replace:false. means directive <my-directive> will display in developer tools

```
myApp.directive("myDirective", function () {
        return {
            restrict: "E",
            replace:true,
            scope: {
                pickMe:'&onSelect',
                fname:'@first',
                item:'='
            },
            template: "<span ng-click='pickMe({selectedItem:item})'>{{item}}
{{fname}}</span>",
            link:function(scope,element,attrs){
                element.on("click",function(event){
                    element.html('thanks for click');
                    element.css({
                        color:"green"
                    })
                })
            }
        }
    });
```

In above we declared replace:true so <my-directive> statement completely replaced by the directive content

AngularJS Custom-Directives transclude, ngTransclude guide

AngularJS Directive's **transclude & ng-transclude** can be used to create directive that wraps other elements. By-default, AngularJS Directive element replaces any element declared as a child element of directive in HTML, by directive's own content. But sometimes you way want to preserve the child element and let the directive act as more of a wrapper rather than replacement. This can be done thanks to **transclude & ng-transclude.**

Let's understand it with help of few examples. In all our previous examples, you have noticed that there is no child-element added under a directive element. Now if you try to wrap an element by a directive, something like shown below:

```
<my-directive item="bla" first="chandra" on-select="printName(selectedItem)">
                example for directive
            </my-directive>
```

on execution, you will see that child element of this widget [that poor dummy message] is found nowhere. It gets kicked out by directive's own content [template]. There might however be cases when we want to preserve the child element of a directive, OR in other words, wrap the existing content by the directive's own content.

AngularJS provides `transclude` option and `ng-transclude` directive to help with this issue. It's a two step process:

- 1. Including transclude:true in directive definition.
- 2. Include ng-transclude directive as an element in directive's own content. ng-transclude acts like a placeholder.

How it works?

- Whenever AngularJS encounters a directive in HTML, and if that directive definition has transclude set to true, AngularJS will make a copy of content the directive is applied on, and will store it somewhere so that it does not get lost when angular replaces it with directive's own template.
- AngularJS needs to know where to put the content it copied in previous step. Since we want to make that content as a child of directive, why not create a placeholder in directive's content itself, which will then be replaced by content copied above. That's exactly angularJS does. It provides **ng-transclude** directive that can be used as a placeholder somewhere in directive's own content, that then be replaced by element's content stored in previous step.

```
myApp.directive("myDirective", function () {
        return {
            restrict: "E",
            replace:true,
            transclude:true,
            scope: {
                pickMe:'&onSelect',
                fname:'@first',
                item:'='
            },
            template: "<div><span ng-click='pickMe({selectedItem:item})'>{{item}}
{{fname}}</span><span ng-transclude></span></div>",
            link:function(scope,element,attrs){
                element.on("click",function(event){
                    element.html('thanks for click');
                    element.css({
                        color:"green"
                    })
                })
            }
        }
    });
```

AngularJS Custom-Directives controllers, require option guide

AngularJS Custom Directive's can have controllers. Controllers in Directive's are used for inter-directive communication. This post discusses Directive's controller, require option and controller argument in directive's link function. Lets get started.

Difference between Directive's link function & Directive's controller

In previous posts, we have discussed `link function`. link function are specific to directive instance and can be used to define **directive's behavior & business logic**. `Controller in directive's` on the other hand are used for **Directive's inter-communication**. That means, one directive on an element wants to communicate with another directive [on the same element or on parent hierarchy]. This includes sharing state or variables, or even functions.

```
<script src="angular.min.js"></script>
<script>
    var myApp = angular.module("myApp", []);

    myApp.directive("myDirective", function () {
        return {
            restrict: "E",
            replace:true,
            transclude:true,
            require:"^parentDirective",
            scope: {
                pickMe:'&onSelect',
                fname:'@first',
                item:'='
            },
            template: "<div><span ng-click='pickMe({selectedItem:item})'>{{item}}
{{fname}}</span><span ng-transclude></span></div>",
            link:function(scope,element,attrs,itemCtrl){
                element.on("click",function(event){
                    element.html('thanks for click');
                    element.css({
                        color:"green"
                    });
                    itemCtrl.addParentItem(scope.item);//Add item to Shopping
Cart.
                        scope.$apply();
                })
            }
```

```
        }
    });

    myApp.directive("parentDirective",function(){
        return{
            restrict:"E",
            replace:true,
            transclude:true,
            template:"<div><span ng-transclude></span><span>parent
config{{parentItems.length}}</span></div>",
            scope:true,
            controller:['$scope',function($scope){
                var self = this;
                $scope.parentItems = [];
                self.addParentItem = function(item){
                    $scope.parentItems.push(item);
                }
            }]
        }
    });

    myApp.controller("myController", function ($scope) {
        $scope.name = "chandra";
        $scope.items = ["apple", "orange", "banana"];
        $scope.movies = ['Ice Age', 'Frozen'];
        $scope.printName = function(name){
            console.log(name);
        }
    })
</script>

<body ng-app="myApp">
    <div ng-controller="myController">
        <parent-directive>
            <div ng-repeat="bla in items">
                <my-directive item="bla" first="chandra" on-
select="printName(selectedItem)">
                    example for directive
                </my-directive>
            </div>
    </parent-directive>
    </div>
</body>
```

1. require option

First of all, notice the `require` option in directive definition. It says to AngularJS that in order to fulfill it's job, itemWidget requires another directives to be present in HTML. In other words, require option specifies the dependencies. A directive can be dependent on more than one directives.

- **require : 'shoppingWidget'** : Specifies that there is only one dependency.
- **require : ['shoppingWidget','ngModel']** : Specifies that there are two dependencies. For multiple dependencies, we use array syntax.

If required directive [on which our directive depends on] found successfully by AngularJS , that directive's controller will be available as the 4th argument in our directive's link function. In case our directive is dependent on more that one directive's, then link function gets an array of controllers as 4th argument. We will come to link function in a moment.

require option sign:
Sign preceding required directive-name provides extra meaning to AngularJS, it is explained below in detail:

- **require: 'shoppingWidget'** : It specifies that a directive shoppingWidget must be present on the element the current directive is applied on. If it is not found, AngularJS will throw an exception.
- **require: '^shoppingWidget'** : It specifies that a directive shoppingWidget must be present on the parent hierarchy [not necessarily immediate parent] of the element the current directive is applied on. If it is not found, AngularJS will throw an exception.
- **require: '?shoppingWidget'** : It specifies that a directive shoppingWidget is an optional dependency. If it is not found on the same element, no exception will be thrown. But then AngularJS will pass null as 4th argument to the link function. Don't forget to null-check before using the controller in this case.
- **require: '?^shoppingWidget'** : It specifies that a directive shoppingWidget is an optional dependency. If it is not found on the parent hierarchy of the current element this directive is applied on, no exception will be thrown. But then AngularJS will pass null as 4th argument to the link function. Don't forget to null-check before using the controller in this case.

2. Controller argument in link function

```
link : function(scope, element, attrs, cartCtrl){

}
```

3. Directive's Controller

There are several interesting things going on here.

- Firstly, we have defined our scope using `scope : true`. It defines a new Scope for this directive so that local variables from this directive don't override anything in the parent scope (whichever it may be). It's a good practice.
- Next, we have defined a controller, using same array syntax we saw in previous posts. In this case, it is injected with $scope. Thanks to scope:true, $scope will refer to directive's own scope, and not polluting parent scope. Anything defined on $scope can directly be accessed in directive's template/HTML.
- We have defined a function **addItemToCart** in controller's instance using **this**. That means this function can be accessed by other components if they have access to this controller. Note that this function can not be used in directive's template/HTML as this is not defined on $scope of directive.
- Finally, we simply push the item passed, to cart array defined on $scope, which can then be available in directive`s HTML/template.

Interesting Notes:

- In case you do not have require key defined in your directive, but your link function contains controller as 4th argument, in that case the controller will refer to your directive's own controller, or undefined if there is no controller defined in your directive.
- You CAN require your own directive. In that case, the 4th argument will refer to your directive's own controller or undefined