AngularJS Comprehensive Directive API

There are many different options for a directive.

The difference resides in the return value of the factory function. You can either return a "Directive Definition Object" (see below) that defines the directive properties, or just the postLink function (all other properties will have the default values).

**Best Practice:** It's recommended to use the "directive definition object" form.

Here's an example directive declared with a Directive Definition Object:

var myModule = angular.module(...);

myModule.directive('directiveName', function factory(injectables) {

var directiveDefinitionObject = {

priority: 0,

template: '<div></div>', // or // function(tElement, tAttrs) { ... },

// or // templateUrl: 'directive.html', // or // function(tElement, tAttrs) { ... },

transclude: false,

restrict: 'A',

templateNamespace: 'html',

scope: false,

controller: function($scope, $element, $attrs, $transclude, otherInjectables) { ... },

controllerAs: 'stringIdentifier',

bindToController: false,

require: 'siblingDirectiveName', // or // ['^parentDirectiveName', '?optionalDirectiveName', '?^optionalParent'],

compile: function compile(tElement, tAttrs, transclude) {

return {

pre: function preLink(scope, iElement, iAttrs, controller) { ... },

post: function postLink(scope, iElement, iAttrs, controller) { ... }

}

// or // return function postLink( ... ) { ... }

},

// or

// link: {

// pre: function preLink(scope, iElement, iAttrs, controller) { ... },

// post: function postLink(scope, iElement, iAttrs, controller) { ... }

// }

// or // link: function postLink( ... ) { ... }

};

return directiveDefinitionObject;

});

**Note:** Any unspecified options will use the default value. You can see the default values below.

Therefore the above can be simplified as:

var myModule = angular.module(...);

myModule.directive('directiveName', function factory(injectables) {

var directiveDefinitionObject = {

link: function postLink(scope, iElement, iAttrs) { ... }

};

return directiveDefinitionObject;

// or // return function postLink(scope, iElement, iAttrs) { ... }

});

Directive Definition Object

The directive definition object provides instructions to the [compiler](https://docs.angularjs.org/api/ng/service/$compile). The attributes are:

multiElement

When this property is set to true, the HTML compiler will collect DOM nodes between nodes with the attributes directive-name-start anddirective-name-end, and group them together as the directive elements. It is recommended that this feature be used on directives which are not strictly behavioural (such as [ngClick](https://docs.angularjs.org/api/ng/directive/ngClick)), and which do not manipulate or replace child nodes (such as [ngInclude](https://docs.angularjs.org/api/ng/directive/ngInclude)).

priority

When there are multiple directives defined on a single DOM element, sometimes it is necessary to specify the order in which the directives are applied. The priority is used to sort the directives before their compile functions get called. Priority is defined as a number. Directives with greater numerical priority are compiled first. Pre-link functions are also run in priority order, but post-link functions are run in reverse order. The order of directives with the same priority is undefined. The default priority is 0.

terminal

If set to true then the current priority will be the last set of directives which will execute (any directives at the current priority will still execute as the order of execution on same priority is undefined). Note that expressions and other directives used in the directive's template will also be excluded from execution.

scope

The scope property can be true, an object or a falsy value:

* **falsy:** No scope will be created for the directive. The directive will use its parent's scope.
* **true:** A new child scope that prototypically inherits from its parent will be created for the directive's element. If multiple directives on the same element request a new scope, only one new scope is created. The new scope rule does not apply for the root of the template since the root of the template always gets a new scope.
* **{...} (an object hash):** A new "isolate" scope is created for the directive's element. The 'isolate' scope differs from normal scope in that it does not prototypically inherit from its parent scope. This is useful when creating reusable components, which should not accidentally read or modify data in the parent scope.

The 'isolate' scope object hash defines a set of local scope properties derived from attributes on the directive's element. These local properties are useful for aliasing values for templates. The keys in the object hash map to the name of the property on the isolate scope; the values define how the property is bound to the parent scope, via matching attributes on the directive's element:

* @ or @attr - bind a local scope property to the value of DOM attribute. The result is always a string since DOM attributes are strings. If no attr name is specified then the attribute name is assumed to be the same as the local name. Given<widget my-attr="hello {{name}}"> and widget definition of scope: { localName:'@myAttr' }, then widget scope propertylocalName will reflect the interpolated value of hello {{name}}. As the name attribute changes so will the localName property on the widget scope. The name is read from the parent scope (not component scope).
* = or =attr - set up bi-directional binding between a local scope property and the parent scope property of name defined via the value of the attr attribute. If no attr name is specified then the attribute name is assumed to be the same as the local name. Given <widget my-attr="parentModel"> and widget definition of scope: { localModel:'=myAttr' }, then widget scope propertylocalModel will reflect the value of parentModel on the parent scope. Any changes to parentModel will be reflected inlocalModel and any changes in localModel will reflect in parentModel. If the parent scope property doesn't exist, it will throw a NON\_ASSIGNABLE\_MODEL\_EXPRESSION exception. You can avoid this behavior using =? or =?attr in order to flag the property as optional. If you want to shallow watch for changes (i.e. $watchCollection instead of $watch) you can use =\* or =\*attr(=\*? or =\*?attr if the property is optional).
* & or &attr - provides a way to execute an expression in the context of the parent scope. If no attr name is specified then the attribute name is assumed to be the same as the local name. Given <widget my-attr="count = count + value"> and widget definition of scope: { localFn:'&myAttr' }, then isolate scope property localFn will point to a function wrapper for thecount = count + value expression. Often it's desirable to pass data from the isolated scope via an expression to the parent scope, this can be done by passing a map of local variable names and values into the expression wrapper fn. For example, if the expression is increment(amount) then we can specify the amount value by calling the localFn as localFn({amount: 22}).

In general it's possible to apply more than one directive to one element, but there might be limitations depending on the type of scope required by the directives. The following points will help explain these limitations. For simplicity only two directives are taken into account, but it is also applicable for several directives:

* **no scope** + **no scope** => Two directives which don't require their own scope will use their parent's scope
* **child scope** + **no scope** => Both directives will share one single child scope
* **child scope** + **child scope** => Both directives will share one single child scope
* **isolated scope** + **no scope** => The isolated directive will use it's own created isolated scope. The other directive will use its parent's scope
* **isolated scope** + **child scope** => **Won't work!** Only one scope can be related to one element. Therefore these directives cannot be applied to the same element.
* **isolated scope** + **isolated scope** => **Won't work!** Only one scope can be related to one element. Therefore these directives cannot be applied to the same element.

bindToController

When an isolate scope is used for a component (see above), and controllerAs is used, bindToController: true will allow a component to have its properties bound to the controller, rather than to scope. When the controller is instantiated, the initial values of the isolate scope bindings are already available.

controller

Controller constructor function. The controller is instantiated before the pre-linking phase and can be accessed by other directives (seerequire attribute). This allows the directives to communicate with each other and augment each other's behavior. The controller is injectable (and supports bracket notation) with the following locals:

* $scope - Current scope associated with the element
* $element - Current element
* $attrs - Current attributes object for the element
* $transclude - A transclude linking function pre-bound to the correct transclusion scope:function([scope], cloneLinkingFn, futureParentElement).
  + scope: optional argument to override the scope.
  + cloneLinkingFn: optional argument to create clones of the original transcluded content.
  + futureParentElement:
    - defines the parent to which the cloneLinkingFn will add the cloned elements.
    - default: $element.parent() resp. $element for transclude:'element' resp. transclude:true.
    - only needed for transcludes that are allowed to contain non html elements (e.g. SVG elements) and when thecloneLinkinFn is passed, as those elements need to created and cloned in a special way when they are defined outside their usual containers (e.g. like <svg>).
    - See also the directive.templateNamespace property.

require

Require another directive and inject its controller as the fourth argument to the linking function. The require takes a string name (or array of strings) of the directive(s) to pass in. If an array is used, the injected argument will be an array in corresponding order. If no such directive can be found, or if the directive does not have a controller, then an error is raised (unless no link function is specified, in which case error checking is skipped). The name can be prefixed with:

* (no prefix) - Locate the required controller on the current element. Throw an error if not found.
* ? - Attempt to locate the required controller or pass null to the link fn if not found.
* ^ - Locate the required controller by searching the element and its parents. Throw an error if not found.
* ^^ - Locate the required controller by searching the element's parents. Throw an error if not found.
* ?^ - Attempt to locate the required controller by searching the element and its parents or pass null to the link fn if not found.
* ?^^ - Attempt to locate the required controller by searching the element's parents, or pass null to the link fn if not found.

controllerAs

Identifier name for a reference to the controller in the directive's scope. This allows the controller to be referenced from the directive template. This is especially useful when a directive is used as component, i.e. with an isolate scope. It's also possible to use it in a directive without an isolate / new scope, but you need to be aware that the controllerAs reference might overwrite a property that already exists on the parent scope.

restrict

String of subset of EACM which restricts the directive to a specific directive declaration style. If omitted, the defaults (elements and attributes) are used.

* E - Element name (default): <my-directive></my-directive>
* A - Attribute (default): <div my-directive="exp"></div>
* C - Class: <div class="my-directive: exp;"></div>
* M - Comment: <!-- directive: my-directive exp -->

templateNamespace

String representing the document type used by the markup in the template. AngularJS needs this information as those elements need to be created and cloned in a special way when they are defined outside their usual containers like <svg> and <math>.

* html - All root nodes in the template are HTML. Root nodes may also be top-level elements such as <svg> or <math>.
* svg - The root nodes in the template are SVG elements (excluding <math>).
* math - The root nodes in the template are MathML elements (excluding <svg>).

If no templateNamespace is specified, then the namespace is considered to be html.

template

HTML markup that may:

* Replace the contents of the directive's element (default).
* Replace the directive's element itself (if replace is true - DEPRECATED).
* Wrap the contents of the directive's element (if transclude is true).

Value may be:

* A string. For example <div red-on-hover>{{delete\_str}}</div>.
* A function which takes two arguments tElement and tAttrs (described in the compile function api below) and returns a string value.

templateUrl

This is similar to template but the template is loaded from the specified URL, asynchronously.

Because template loading is asynchronous the compiler will suspend compilation of directives on that element for later when the template has been resolved. In the meantime it will continue to compile and link sibling and parent elements as though this element had not contained any directives.

The compiler does not suspend the entire compilation to wait for templates to be loaded because this would result in the whole app "stalling" until all templates are loaded asynchronously - even in the case when only one deeply nested directive has templateUrl.

Template loading is asynchronous even if the template has been preloaded into the [$templateCache](https://docs.angularjs.org/api/ng/service/$templateCache)

You can specify templateUrl as a string representing the URL or as a function which takes two arguments tElement and tAttrs(described in the compile function api below) and returns a string value representing the url. In either case, the template URL is passed through [$sce.getTrustedResourceUrl](https://docs.angularjs.org/api/ng/service/$sce#getTrustedResourceUrl).

replace ([*DEPRECATED*!], will be removed in next major release - i.e. v2.0)

specify what the template should replace. Defaults to false.

* true - the template will replace the directive's element.
* false - the template will replace the contents of the directive's element.

The replacement process migrates all of the attributes / classes from the old element to the new one. See the [Directives Guide](https://docs.angularjs.org/guide/directive#template-expanding-directive) for an example.

There are very few scenarios where element replacement is required for the application function, the main one being reusable custom components that are used within SVG contexts (because SVG doesn't work with custom elements in the DOM tree).

transclude

Extract the contents of the element where the directive appears and make it available to the directive. The contents are compiled and provided to the directive as a **transclusion function**. See the [Transclusion](https://docs.angularjs.org/api/ng/service/$compile" \l "transclusion) section below.

There are two kinds of transclusion depending upon whether you want to transclude just the contents of the directive's element or the entire element:

* true - transclude the content (i.e. the child nodes) of the directive's element.
* 'element' - transclude the whole of the directive's element including any directives on this element that defined at a lower priority than this directive. When used, the template property is ignored.

compile

function compile(tElement, tAttrs, transclude) { ... }

The compile function deals with transforming the template DOM. Since most directives do not do template transformation, it is not used often. The compile function takes the following arguments:

* tElement - template element - The element where the directive has been declared. It is safe to do template transformation on the element and child elements only.
* tAttrs - template attributes - Normalized list of attributes declared on this element shared between all directive compile functions.
* transclude - [*DEPRECATED*!] A transclude linking function: function(scope, cloneLinkingFn)

**Note:** The template instance and the link instance may be different objects if the template has been cloned. For this reason it is **not**safe to do anything other than DOM transformations that apply to all cloned DOM nodes within the compile function. Specifically, DOM listener registration should be done in a linking function rather than in a compile function.

**Note:** The compile function cannot handle directives that recursively use themselves in their own templates or compile functions. Compiling these directives results in an infinite loop and a stack overflow errors. This can be avoided by manually using $compile in the postLink function to imperatively compile a directive's template instead of relying on automatic template compilation via templateor templateUrl declaration or manual compilation inside the compile function.

**Note:** The transclude function that is passed to the compile function is deprecated, as it e.g. does not know about the right outer scope. Please use the transclude function that is passed to the link function instead.

A compile function can have a return value which can be either a function or an object.

* returning a (post-link) function - is equivalent to registering the linking function via the link property of the config object when the compile function is empty.
* returning an object with function(s) registered via pre and post properties - allows you to control when a linking function should be called during the linking phase. See info about pre-linking and post-linking functions below.

link

This property is used only if the compile property is not defined.

function link(scope, iElement, iAttrs, controller, transcludeFn) { ... }

The link function is responsible for registering DOM listeners as well as updating the DOM. It is executed after the template has been cloned. This is where most of the directive logic will be put.

* scope - [Scope](https://docs.angularjs.org/api/ng/type/$rootScope.Scope) - The scope to be used by the directive for registering [watches](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch).
* iElement - instance element - The element where the directive is to be used. It is safe to manipulate the children of the element only in postLink function since the children have already been linked.
* iAttrs - instance attributes - Normalized list of attributes declared on this element shared between all directive linking functions.
* controller - the directive's required controller instance(s) - Instances are shared among all directives, which allows the directives to use the controllers as a communication channel. The exact value depends on the directive's require property:
  + no controller(s) required: the directive's own controller, or undefined if it doesn't have one
  + string: the controller instance
  + array: array of controller instances

If a required controller cannot be found, and it is optional, the instance is null, otherwise the [Missing Required Controller](https://docs.angularjs.org/error/$compile/ctreq) error is thrown.

Note that you can also require the directive's own controller - it will be made available like any other controller.

* transcludeFn - A transclude linking function pre-bound to the correct transclusion scope. This is the same as the $transcludeparameter of directive controllers, see there for details. function([scope], cloneLinkingFn, futureParentElement).

Pre-linking function

Executed before the child elements are linked. Not safe to do DOM transformation since the compiler linking function will fail to locate the correct elements for linking.

Post-linking function

Executed after the child elements are linked.

Note that child elements that contain templateUrl directives will not have been compiled and linked since they are waiting for their template to load asynchronously and their own compilation and linking has been suspended until that occurs.

It is safe to do DOM transformation in the post-linking function on elements that are not waiting for their async templates to be resolved.

Transclusion

Transclusion is the process of extracting a collection of DOM elements from one part of the DOM and copying them to another part of the DOM, while maintaining their connection to the original AngularJS scope from where they were taken.

Transclusion is used (often with [ngTransclude](https://docs.angularjs.org/api/ng/directive/ngTransclude)) to insert the original contents of a directive's element into a specified place in the template of the directive. The benefit of transclusion, over simply moving the DOM elements manually, is that the transcluded content has access to the properties on the scope from which it was taken, even if the directive has isolated scope. See the [Directives Guide](https://docs.angularjs.org/guide/directive#creating-a-directive-that-wraps-other-elements).

This makes it possible for the widget to have private state for its template, while the transcluded content has access to its originating scope.

**Note:** When testing an element transclude directive you must not place the directive at the root of the DOM fragment that is being compiled. See [Testing Transclusion Directives](https://docs.angularjs.org/guide/unit-testing#testing-transclusion-directives).

Transclusion Functions

When a directive requests transclusion, the compiler extracts its contents and provides a **transclusion function** to the directive's linkfunction and controller. This transclusion function is a special **linking function** that will return the compiled contents linked to a new transclusion scope.

If you are just using [ngTransclude](https://docs.angularjs.org/api/ng/directive/ngTransclude) then you don't need to worry about this function, since ngTransclude will deal with it for us.

If you want to manually control the insertion and removal of the transcluded content in your directive then you must use this transclude function. When you call a transclude function it returns a a jqLite/JQuery object that contains the compiled DOM, which is linked to the correct transclusion scope.

When you call a transclusion function you can pass in a **clone attach function**. This function accepts two parameters,function(clone, scope) { ... }, where the clone is a fresh compiled copy of your transcluded content and the scope is the newly created transclusion scope, to which the clone is bound.

**Best Practice**: Always provide a cloneFn (clone attach function) when you call a translude function since you then get a fresh clone of the original DOM and also have access to the new transclusion scope.

It is normal practice to attach your transcluded content (clone) to the DOM inside your **clone attach function**:

var transcludedContent, transclusionScope;

$transclude(function(clone, scope) {

element.append(clone);

transcludedContent = clone;

transclusionScope = scope;

});

Later, if you want to remove the transcluded content from your DOM then you should also destroy the associated transclusion scope:

transcludedContent.remove();

transclusionScope.$destroy();

**Best Practice**: if you intend to add and remove transcluded content manually in your directive (by calling the transclude function to get the DOM and calling element.remove() to remove it), then you are also responsible for calling $destroy on the transclusion scope.

The built-in DOM manipulation directives, such as [ngIf](https://docs.angularjs.org/api/ng/directive/ngIf), [ngSwitch](https://docs.angularjs.org/api/ng/directive/ngSwitch) and [ngRepeat](https://docs.angularjs.org/api/ng/directive/ngRepeat) automatically destroy their transluded clones as necessary so you do not need to worry about this if you are simply using [ngTransclude](https://docs.angularjs.org/api/ng/directive/ngTransclude) to inject the transclusion into your directive.

Transclusion Scopes

When you call a transclude function it returns a DOM fragment that is pre-bound to a **transclusion scope**. This scope is special, in that it is a child of the directive's scope (and so gets destroyed when the directive's scope gets destroyed) but it inherits the properties of the scope from which it was taken.

For example consider a directive that uses transclusion and isolated scope. The DOM hierarchy might look like this:

<div ng-app>

<div isolate>

<div transclusion>

</div>

</div>

</div>

The $parent scope hierarchy will look like this:

- $rootScope

- isolate

- transclusion

but the scopes will inherit prototypically from different scopes to their $parent.

- $rootScope

- transclusion

- isolate

Attributes

The [Attributes](https://docs.angularjs.org/api/ng/type/$compile.directive.Attributes) object - passed as a parameter in the link() or compile() functions. It has a variety of uses.

accessing *Normalized attribute names:* Directives like 'ngBind' can be expressed in many ways: 'ng:bind', data-ng-bind, or 'x-ng-bind'. the attributes object allows for normalized access to the attributes.

* *Directive inter-communication:* All directives share the same instance of the attributes object which allows the directives to use the attributes object as inter directive communication.
* *Supports interpolation:* Interpolation attributes are assigned to the attribute object allowing other directives to read the interpolated value.
* *Observing interpolated attributes:* Use $observe to observe the value changes of attributes that contain interpolation (e.g.src="{{bar}}"). Not only is this very efficient but it's also the only way to easily get the actual value because during the linking phase the interpolation hasn't been evaluated yet and so the value is at this time set to undefined.

function linkingFn(scope, elm, attrs, ctrl) {

console.log(attrs.ngModel);

attrs.$set('ngModel', 'new value');

// observe changes to interpolated attribute

attrs.$observe('ngModel', function(value) {

console.log('ngModel has changed value to ' + value);

});

}