Use this directive to **auto-bootstrap** an AngularJS application. The ngApp directive designates the **root element** of the application and is typically placed near the root element of the page - e.g. on the <body> or <html> tags.

There are a few things to keep in mind when using ngApp:

* only one AngularJS application can be auto-bootstrapped per HTML document. The first ngApp found in the document will be used to define the root element to auto-bootstrap as an application. To run multiple applications in an HTML document you must manually bootstrap them using [angular.bootstrap](https://docs.angularjs.org/api/ng/function/angular.bootstrap) instead.
* AngularJS applications cannot be nested within each other.

The ng-app directive tells AngularJS that this is the root element of the AngularJS application.

All AngularJS applications must have a root element.

You can only have one ng-app directive in your HTML document. If more than one ng-app directive appears, the first appearance will be used.

Syntax

<*element* ng-app="*modulename*">  
...  
*content inside the ng-app root element can contain AngularJS code*...  
</*element*>

https://www.guru99.com/angularjs-directive.html

1. **ng-app** – This is used to initialize an Angular.JS application. When this directive in placed in an HTML page, it basically tells Angular that this HTML page is an angular.js application.

The example below shows how to use the ng-app directive. In this example, we are simply going to show how to make a normal HTML application an angularJS application.

AngularJS directives is a way to extend the functionality of HTML.

The ng-init directive evaluates the given expression(s).

The ng-init directive can add some unnecessary logic into the scope, and you are recommended to do your evaluations in a controller instead, see the [ng-controller](https://www.w3schools.com/angular/ng_ng-controller.asp) directive.

Example:

<div ng-app="" ng-init="myText='Hello World!'">  
  
<h1>{{myText}}</h1>

http://tutorials.jenkov.com/angularjs/custom-directives.html

## Custom Directive Introduction

AngularJS directives are what controls the rendering of the HTML inside an AngularJS application. Examples of directives are the interpolation directive ( {{ }} ), the ng-repeat directive and ng-if directive.

It is possible to implement your own directives too. This is what AngularJS refers to as "teaching HTML new tricks". This text will show you how to do that.

## Directive Types

You can implement the following types of directives:

* Element directives
* Attribute directives
* CSS class directives
* Comment directives

Of these, AngularJS recommends that you try to use element and attribute directives, and leave the CSS class and comment directives (unless absolutely necessary).

The type of a directive determines how the directive is activated. An element directive is activated when AngularJS finds a matching HTML element in the HTML template. An attribute directive is activated when AngularJS finds a matching HTML element attribute. A CSS class directive is activated when AngularJS finds a matching CSS Class. And, a comment directive is activated when AngularJS finds a matching HTML comment.

## A Basic Directive

You register a directive with a module. Here is an example of how that looks:

myapp = angular.module("myapp", []);

myapp.directive('div', function() {

var directive = {};

directive.restrict = 'E'; /\* restrict this directive to elements \*/

directive.template = "My first directive: {{textToInsert}}";

return directive;

});

Notice the call to the directive() function on the module. When you call this function you can register a new directive. The first parameter to the directive() function call is the name of the directive to register. This name is what you use in your HTML templates when you want to activate the directive. In this example I have used the name 'div' which means that the directive is activated every time an HTML element named div is found in the HTML template.

The second parameter passed to the directive function is a factory function. This function should return a directive definition when invoked. AngularJS will invoke this function to obtain a JavaScript object which contains the definition of the directive. If you look inside the function in the above example you will see that it does indeed return a JavaScript object.

The JavaScript object returned from the factory function has two properties: A restrict field and a template field.

The restrict field is used to set if the directive should be activated by a matching HTML element, or an element attribute. By setting restrict to E you specify that only HTML elements named div should activate the directive. By setting restrict to A you specify that only HTML attributes named div should activate the directive. You can also use a value of AE which will match both HTML element names and attribute names.

The template field is an HTML template that will replace the content of the matched div element. It will work as if the content of the matched div element had not been there, and instead this HTML template had been located in the same place.

Imagine that your HTML page has this HTML:

<div ng-controller="MyController" >

<div>This div will be replaced</div>

</div>

Then the added directive would be activated when AngularJS finds the inner div element. Instead of this div element, this HTML will be inserted:

My first directive: {{textToInsert}}

As you can see, this HTML contains an interpolation directive ({{textToInsert}}). AngularJS will interpret this HTML again, so that the interpolation directive actually works. The value of the $scope.textToInsertproperty will be inserted at this point in the HTML.

## The template and templateUrl Properties

The easiest way to create your own directives is as shown in the example above. Your directive is intended to generate HTML, and you put that HTML inside the template attribute of the directive definition object. Here is the directive definition repeated from earlier, with the template string marked in bold:

myapp = angular.module("myapp", []);

myapp.directive('div', function() {

var directive = {};

directive.restrict = 'E'; /\* restrict this directive to elements \*/

**directive.template = "My first directive: {{textToInsert}}";**

return directive;

});

In case that HTML template grows big, it is gets hard to write and maintain the HTML inside a JavaScript string. You can then put the HTML into its own file and have AngularJS load it from that file. You do so by putting the URL of the HTML template file into the templateUrl property of the directive definition object. Here is an example:

myapp = angular.module("myapp", []);

myapp.directive('div', function() {

var directive = {};

directive.restrict = 'E'; /\* restrict this directive to elements \*/

**directive.templateUrl = "/myapp/html-templates/div-template.html";**

return directive;

});

AngularJS will now load the HTML template from the URL set in the templateUrl property.

Using the separate HTML template file and the templateUrl property is especially useful when you create more specialized directives, like a directives showing user info. Here is an example:

myapp = angular.module("myapp", []);

myapp.directive('userinfo', function() {

var directive = {};

directive.restrict = 'E'; /\* restrict this directive to elements \*/

directive.templateUrl = "/myapp/html-templates/userinfo-template.html";

return directive;

});

This example creates a directive that is activated whenever AngularJS finds a <userinfo> element. AngularJS loads the HTML template found at /myapp/html-templates/userinfo-template.html, and interprets that as if it had been located inside the parent HTML file from the beginning.

## Isolating the $scope From the Directive

In the example above the userinfo directive was bound hard to the $scope variable because the HTML template referenced the textToInsert property directly. Referencing $scope variables directly makes it hard to reuse the directive more than once within the same controller, since the $scope variables typically have the same values everywhere inside the same controller. For instance, if you wanted to have this HTML in your page:

<userinfo></userinfo>

<userinfo></userinfo>

Then the two <userinfo> elements would be replaced by the same HTML template, which is bound to the same $scope variable. The result would be that the two <userinfo> elements would be replaced by the exact same HTML code.

To be able to bind the two <userinfo> elements to different values in the $scope object, you need to bind the HTML template to an *isolate scope*.

An isolate scope is a separate scope object tied to the directive. Here is how you define it in the directive definition object:

myapp.directive('userinfo', function() {

var directive = {};

directive.restrict = 'E';

directive.template = "User : **{{user.firstName}}** **{{user.lastName}}**";

directive.scope = {

user : "=user"

}

return directive;

})

Notice how the HTML template has two interpolation directives bound to {{user.firstName}} and {{user.lastName}}. Notice the user. part. And notice the directive.scope property. The directive.scopeproperty is a JavaScript object which contains a property named user. The directive.scope property is the isolate scope object, and the HTML template is now bound to the directive.scope.user object (via the {{user.firstName}} and {{user.lastName}} interpolation directives).

The directive.scope.user property is set to "=user". That means, that the directive.scope.user property is bound to the property in the scope property (not in the isolate scope) with the name passed to the userattribute of the <userinfo> element. It sounds confusing, so look at this HTML example:

<userinfo user="jakob"></userinfo>

<userinfo user="john"></userinfo>

These two <userinfo> element contain a user attribute. The value of these attributes contain the names of properties in the $scope object which are to be referenced by the isolate scope object's userinfo property.

Here is a full example:

<userinfo user="jakob"></userinfo>

<userinfo user="john"></userinfo>

<script>

myapp.directive('userinfo', function() {

var directive = {};

directive.restrict = 'E';

directive.template = "User : <b>{{user.firstName}}</b> <b>{{user.lastName}}</b>";

directive.scope = {

user : "=user"

}

return directive;

});

myapp.controller("MyController", function($scope, $http) {

$scope.jakob = {};

$scope.jakob.firstName = "Jakob";

$scope.jakob.lastName = "Jenkov";

$scope.john = {};

$scope.john.firstName = "John";

$scope.john.lastName = "Doe";

});

</script>

## The compile() and link() Functions

If you need to do something more advanced inside your directive, something that you cannot do with an HTML template, you can use the compile() and link() functions instead.

The compile() and link() functions define how the directive is to modify the HTML that matched the directive.

The compile() function is called once for each occurrence of the directive in the HTML page. The compile() function can then do any one-time configuration needed of the element containing the directive.

The compile() function finishes by returning the link() function. The link() function is called every time the element is to be bound to data in the $scope object.

As mentioned, you add the compile() function to the directive definition object, and the compile() function has to return the link() function when executed. Here is how that looks:

<script>

myapp = angular.module("myapp", []);

myapp.directive('userinfo', function() {

var directive = {};

directive.restrict = 'E'; /\* restrict this directive to elements \*/

directive.compile = function(element, attributes) {

// do one-time configuration of element.

var linkFunction = function($scope, element, atttributes) {

// bind element to data in $scope

}

return linkFunction;

}

return directive;

});

</script>

The compile() function takes two parameters: The element and attributes parameters.

The element parameter is a jqLite wrapped DOM element. AngularJS contains a lite version of jQuery to help you do DOM manipulation, so the element's DOM manipulation methods are the same as you know from jQuery.

The attributes parameter is a JavaScript object containing properties for all the attributes of the DOM element. Thus, to access an attribute named type you would write attributes.type.

The link() function takes three parameters: The $scope parameter, the element parameter and the attributes parameter. The element and attributes parameter is the same as passed to the compile()function. The $scope parameter is the normal scope object, or an isolate scope in case you have specified one in the directive definition object.

The compile() and link() function names are actually confusing. They are inspired by compiler terms. I can see the resemblance, but a compiler parses an input once, and creates an output. A directive configures an HTML element and then updates that HTML subsequently whenever the $scope object changes.

A better name for the compile() function would have been something like create(), init() or configure(). Something that signals that this function is only called once.

A better name for the link() function would have been something like bind() or render(), which signals that this function is called whenever the directive needs to bind data to it, or to re-render it.

Here is a full example that shows a directive that uses both a compile() and link() function:

<div ng-controller="MyController" >

<userinfo >This will be replaced</userinfo>

</div>

<script>

myapp = angular.module("myapp", []);

myapp.directive('userinfo', function() {

var directive = {};

directive.restrict = 'E'; /\* restrict this directive to elements \*/

directive.compile = function(element, attributes) {

element.css("border", "1px solid #cccccc");

var linkFunction = function($scope, element, attributes) {

element.html("This is the new content: " + $scope.firstName);

element.css("background-color", "#ffff00");

}

return linkFunction;

}

return directive;

})

myapp.controller("MyController", function($scope, $http) {

$scope.cssClass = "notificationDiv";

$scope.firstName = "Jakob";

$scope.doClick = function() {

console.log("doClick() called");

}

});

</script>

The compile() function sets a border on the HTML element. This is only executed once because the compile() function is only executed once.

The link() function replaces the content of the HTML element, and sets the background color to yellow.

There is no particular reason why the border was set in the compile() function, and the background color in the link() function. Both could have been set in the compile() function, or both in the link() function. If set in the compile() function they would only have been set once (which is often what you want). If set in the link() function they would be set every time the HTML element is bound to data in the $scope object. This might be useful if you needed to set the border and background color differently depending on data in the $scope object.

### Setting Only a link() Function

Sometimes you do not need the compile() step for your directive. You only need th link() function. In that case you can set the link() function directly on the directive definition object. Here is the example from before, with only a link function:

<div ng-controller="MyController" >

<userinfo >This will be replaced</userinfo>

</div>

<script>

myapp = angular.module("myapp", []);

myapp.directive('userinfo', function() {

var directive = {};

directive.restrict = 'E'; /\* restrict this directive to elements \*/

directive.link = function($scope, element, attributes) {

element.html("This is the new content: " + $scope.firstName);

element.css("background-color", "#ffff00");

}

return directive;

})

myapp.controller("MyController", function($scope, $http) {

$scope.cssClass = "notificationDiv";

$scope.firstName = "Jakob";

$scope.doClick = function() {

console.log("doClick() called");

}

});

</script>

Notice how the link() function does the same as the link() function returned in the previous example.

## Directives Which Wraps Elements Via Transclusion

The examples we have seen so far all set the content of the element matching the directive themselves, either via JavaScript code or an HTML template. But what if you wanted a directive to wrap elements inserted into the directive body by the developer? For instance:

<mytransclude>This is a transcluded directive {{firstName}}</mytransclude>

The directive is marked by the <mytransclude> element. But the content inside it is set by the developer. Thus, this part of the HTML should not be replaced by the directive's HTML template. We actually want that part of the HTML to be processed by AngularJS. This processing is called "transclusion".

In order to make AngularJS process the HTML inside a directive, you have to set the transclude property of the directive definition object to true. You will also have to tell AngularJS what part of the directive's HTML template that is to contain the transcluded HTML. You do so by inserting the ng-transclude attribute (a directive, really) into the HTML element in the HTML template where you want the transcluded HTML inserted.

Here is an AngularJS directive that shows how to use transclusion:

<mytransclude>This is a transcluded directive {{firstName}}</mytransclude>

<script>

myapp = angular.module("myapp", []);

myapp.directive('mytransclude', function() {

var directive = {};

directive.restrict = 'E'; /\* restrict this directive to elements \*/

**directive.transclude = true;**

directive.template = "<div class='myTransclude' **ng-transclude**></div>";

return directive;

});

myapp.controller("MyController", function($scope, $http) {

$scope.firstName = "Jakob";

});

</script>

Notice the HTML inside the <mytransclude> element. This HTML code contains the interpolation directive {{firstName}}. We want AngularJS to process this HTML for us so that interpolation directive is executed. To achieve that I have set the transclude property to true on the directive definition object. I have also inserted an ng-transclude attribute into the HTML template. This attribute tells AngularJS what element to insert the transcluded HTML into.

Difference between factory and service ?

How to pass data between controllers ?

There are many ways you can share the data between controllers

1. using services
2. using $state.go services
3. using stateparams
4. using rootscope

Explanation of each method:

1. I am not going to explain as its already explained by someone
2. using $state.go
3. $state.go('book.name', {Name: 'XYZ'});
4. // then get parameter out of URL

$state.params.Name;

1. $stateparam works in a similar way to $state.go, you pass it as object from sender controller and collect in receiver controller using stateparam
2. using $rootscope

(a) sending data from child to parent controller

$scope.Save(Obj,function(data) {

$scope.$emit('savedata',data);

//pass the data as the second parameter

});

$scope.$on('savedata',function(event,data) {

//receive the data as second parameter

});

(b) sending data from parent to child controller

$scope.SaveDB(Obj,function(data){

$scope.$broadcast('savedata',data);

});

$scope.SaveDB(Obj,function(data){`enter code here`

$rootScope.$broadcast('saveCallback',data);

});

1. We can share data between two or more controllers using service or factory.

**Example:** First, I am creating a product service.

myApp.service('productService', function() {

var product = [], addProduct, getProductList, removeProduct;

addProduct = function(obj) {

product.push(obj);

};

getProductList = function(){

return product;

};

removeProduct = function(id {

product.splice(id, 1);

};

return {

addProduct: addProduct,

removeProduct: removeProduct,

getProductList: getProductList

};

});

Service will inject as a Dependency injection into controllers.

Now i will show. How to share product service in two controllers.

In ListController, we can add or remove product to cart from product list:

myApp.controller('ListController', function($scope, productService) {

$scope.addToCart = function(obj) {

productService.addProduct(obj);

};

$scope.removeToCart = function(id) {

productService.removeProduct(id);

};

});

In CartController, here we can get all product which we added to cart and also we can remove product from cart:

myApp.controller('CartController', function($scope, productService) {

$scope.productList = productService.getProductList();

$scope.removeToCart = function(id){

productService.removeProduct(id);

};

});

In place of service we can also use factory.

How to share data between controllers in AngularJS

In my AngularJS classes, I often get asked, “How do I share data between the controllers in AngularJS?” On the Internet, there are many solutions suggested. However, I prefer to share data using the Shared Data Service method, and that’s what we’re going to explore in this post.

To start with, let’s suppose that we want to share a Product object between the controllers. Here I have created an AngularJS service named SharedDataService as shown in the snippet below:

myApp.service('SharedDataService', function () {

var Product = {

name: '',

price: ''

};

return Product;

});

Next let’s go ahead and create two different controllers using SharedDataService. In the controllers, we are using the service created in the previous step. Controllers can be created as shown below:

var myApp = angular.module("myApp", ['CalService']);

myApp.controller("DataController1", ['$scope', 'SharedDataService',

function ($scope, SharedDataService) {

$scope.Product = SharedDataService;

}]);

myApp.controller("DataController2", ['$scope', 'SharedDataService',

function ($scope, SharedDataService) {

$scope.Product = SharedDataService;

}]);

On the view we can simply use the controllers as shown in the listing below:

<div ng-controller="DataController1">

<h2>In Controller 1h2>

<input type="text" ng-model="Product.name" /> <br/>

<input type="text" ng-model="Product.price" />

<h3>Product {{Product.name}} costs {{Product.price}} h3>

div>

<hr/>

<div ng-controller="DataController2">

<h2>In Controller 2h2>

<h3>Product {{Product.name}} costs {{Product.price}} h3>

div>

Now we can share the data between the controllers. As you can see, the name and price of the product is being set in the DataController1, and we are fetching the data in the DataController2.

Do you have any better options that you use to share data? Or perhaps you have a complex scenario which may be not be solved by the above approach. If so, let me know! Tell me about it in the comments below.

What is Singleton object

Assume u having an object but it is injected in many places, but it only uses object reference, In AngularJS all services are singletons. service, providers, factory, constants all are a singleton.

Eg: If you wanna access a data from one controller to another controller singleton service is used.

It's singleton, there is only one object, but is injected into many places. (objects are passed by reference to a method)

Example :

App.js

var app = angular.module('plunker', []);

// Singleton

app.service('ChoirBoyService', function () {

this.number = Math.random(10);

this.sing = function () {

return this.number;

};

});

function HelloCtrl($scope, ChoirBoyService) {

choirBoy = ChoirBoyService;

$scope.song = ChoirBoyService.sing();

}

function GoodbyeCtrl($scope, ChoirBoyService) {

choirBoy = ChoirBoyService;

$scope.song = ChoirBoyService.sing();

}

Index.html

<!DOCTYPE html>

<html ng-app="plunker">

<head>

<meta charset="utf-8" />

<title>AngularJS Plunker</title>

<link data-require="bootstrap-css@\*" data-semver="3.0.2" rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.2/css/bootstrap.min.css" />

<script>

document.write('<base href="' + document.location + '" />');

</script>

<link rel="stylesheet" href="style.css" />

<script data-require="angular.js@1.0.x" src="http://code.angularjs.org/1.2.1/angular.js" data-semver="1.2.1"></script>

<script src="app.js"></script>

</head>

<body>

<div ng-controller="HelloCtrl">

<p>{{song}}</p>

</div>

<br />

<div ng-controller="GoodbyeCtrl">

<p>{{song}}</p>

</div>

</body>

</html>

Output :

0.7677268186672896

0.7677268186672896

What is callback

callback is not a keyword, its just a name of parameter that is passed into the function, you can call it whatever you want (callback or cb is pretty common).

I'll try to explain it on example of super simplistic custom build callback function:

function useAsCallback(string){

console.log("callback is being executed with passed parameter: " + string)

}

function main(param, callback){

callback(param)

}

main(123456, useAsCallback)

if you run this, it would print: callback is being executed with passed parameter: 123456

another problem is... what callback functions are questioned?

you can have a function, which accepts function as cb function alias:

function test(param1, params2, callback) {

// do something

callback();

}

But if your function is a promise function like:

function test() {

var q = $q.defer();

// do something

if (success) {

q.resolve();

} else {

q.reject();

}

return q.promise;

}

you can set callbacks like that:

test().then(function () {

// success

}, function () {

// error

});

A **Callback** is a function which we call inside other function. Normally callback runs after the parent function completes its operation. Let’s say the function is sending AJAX query, we can use callback function in a way so, when the AJAX request gets response, we can use the response in the function to do some other operation.

Pseudo element

<!DOCTYPE html>

<html>

<head>

<style>

p::first-line {

color: #ff0000;

font-variant: small-caps;

}

</style>

</head>

<body>

<p>You can use the ::first-line pseudo-element to add a special effect to the first line of a text. Some more text. And even more, and more, and more, and more, and more, and more, and more, and more, and more, and more, and more, and more.</p>

</body>

</html>

First line red color

O/P : You can use the ::first-line pseudo-element to add a special effect to the first line of a text. Some more text. And even more, and more, and more, and more, and more, and more, and more, and more, and more, and more, and more, and more.

* [::after](https://developer.mozilla.org/en-US/docs/Web/CSS/::after)
* [::before](https://developer.mozilla.org/en-US/docs/Web/CSS/::before)
* [::first-letter](https://developer.mozilla.org/en-US/docs/Web/CSS/::first-letter)

What are Pseudo-classes?

A pseudo-class is used to define a special state of an element.

For example, it can be used to:

* Style an element when a user mouses over it
* Style visited and unvisited links differently
* Style an element when it gets focus

/\* unvisited link \*/  
a:link {  
    color: #FF0000;  
}  
  
/\* visited link \*/  
a:visited {  
    color: #00FF00;  
}  
  
/\* mouse over link \*/  
a:hover {  
    color: #FF00FF;  
}  
  
/\* selected link \*/  
a:active {  
    color: #0000FF;  
}