## ****What are the basic steps to unit test an AngularJS filter?****

1. Inject the module that contains the filter.
2. Provide any mocks that the filter relies on.
3. Get an instance of the filter using $filter('yourFilterName').
4. Assert your expectations.

Dependency injection is a powerful software design pattern that Angular employs to compose responsibilities through an intrinsic interface. However, for those new to the process, it can be puzzling where you need to configure and mock these dependencies when creating your isolated unit tests. The open-source project “Angular Test Patterns” is a free resource that is focused on dispelling such confusion through high-quality examples.

This question is useful since it can give you a feel for how familiar the candidate is with automated testing (TDD (test-driven-development), BDD (behavior-driven-development like Jasmine), E2E (End to End)), as well as open up a conversation about approaches to code quality.

[Filters](https://docs.angularjs.org/api/ng/provider/$filterProvider) are functions which transform the data into a user readable format. They are important because they remove the formatting responsibility from the application logic, further simplifying the application logic.

myModule.filter('length', function() {

return function(text) {

return ('' + (text || '')).length;

}

});

describe('length filter', function() {

var $filter;

beforeEach(inject(function(\_$filter\_){

$filter = \_$filter\_;

}));

it('returns 0 when given null', function() {

var length = $filter('length');

expect(length(null)).toEqual(0);

});

it('returns the correct value when given a string of chars', function() {

var length = $filter('length');

expect(length('abc')).toEqual(3);

});

});

## ****What should be the maximum number of concurrent “watches”? Bonus: How would you keep an eye on that number?****

To reduce memory consumption and improve performance it is a good idea to limit the number of watches on a page to 2,000. A utility called ng-stats can help track your watch count and digest cycles.

*Jank* happens when your application cannot keep up with the screen refresh rate. To achieve 60 frames-per-second, you only have about 16 milliseconds for your code to execute. It is crucial that the scope digest cycles are as short as possible for your application to be responsive and smooth. Memory use and digest cycle performance are directly affected by the number of active watches. Therefore, it is best to keep the number of watches below 2,000. The open-source utility ng-stats gives developers insight into the number of watches Angular is managing, as well as the frequency and duration of digest cycles over time.

Caution: Be wary of relying on a “single magic metric” as the golden rule to follow. You must take the context of your application into account. The number of watches is simply a basic health signal. If you have many thousands of watches, or worse, if you see that number continue to grow as you interact with your page. Those are strong indications that you should look under the hood and review your code.

This question is valuable as it gives insight into how the candidate debugs runtime issues while creating a discussion about performance and optimization.

**ng-stats**

Little utility to show stats about your page's angular digest/watches. This library currently has a simple script to produce a chart (see below). It also creates a module called angularStats which has a directive called angular-stats which can be used to put angular stats on a specific place on the page that you specify.

**Installation**

1. Bookmarklet

Copy the code below and create a bookmarklet for ng-stats to use it on any angular website (so long as the debug info is enabled, if not, you'll need to run angular.reloadWithDebugInfo() first).

javascript: (function() {var a = document.createElement("script");a.src = "https://rawgit.com/kentcdodds/ng-stats/master/dist/ng-stats.js";a.onload=function(){window.showAngularStats()};document.head.appendChild(a)})();

If you just want the chart for development purposes, it's actually easiest to use as a [Chrome DevTools Snippet](https://developer.chrome.com/devtools/docs/authoring-development-workflow#snippets). Just copy/paste the dist/ng-stats.js file into a snippet.

However, it uses UMD, so you can also include it in your app if you want via:

$ npm|bower install ng-stats

or download dist/ng-stats.js and

<script src="path-to/ng-stats.js"></script>

or

var showAngularStats = require('path-to-ng-stats');

You now have a angularStats module and showAngularStats function you can call

1. Chart

**Usage**

Simply invoke showAngularStats( { options } ) and the chart will appear. It also returns an object with a few handy things depending on your options. One of these things is listeners which is an object that has two objects: digestLength and watchCount. You can add a custom listener that is called when the digest cycles happen (though for performance reasons when calculating the watchCount, the watchCount listeners are throttled). Here's an example of adding custom listeners:

var ngStats = showAngularStats();

ngStats.listeners.digestLength.nameOfYourListener = function(digestLength) {

console.log('Digest: ' + digestLength);

};

ngStats.listeners.watchCount.nameOfYourListener = function(watchCount) {

console.log('Watches: ' + watchCount);

};

**Options**

You can pass the function one (optional) argument. If you pass false it will turn off "autoload" and do nothing. You can also pass an object with other options:

position (object) - default: 'topleft'

Controls the position of the graphic. Possible values: Any combination of top, left, right, bottom.

digestTimeThreshold (number) - default: 16

The time (in milliseconds) where it goes from red to green.

autoload (string or boolean) - default: false

Uses the [Storage API](https://developer.mozilla.org/en-US/docs/Web/API/Storage) to store whether the graphic should be automatically loaded every time the page is reloaded. Pass in 'localStorage' for persistent loading or 'sessionStorage' to load ng-stats for only the current session.

Note, if you pass false as options, it will simply remove the stats window and exit: showAngularStats(false)

trackDigest (boolean) - default: false

showAngularStats returns an object. Setting this to true will add an array to that object called digest that holds all of the digest lengths.

trackWatches (boolean) - default: false

showAngularStats returns an object. Setting this to true will add an array to that object called watches that holds all of the watch counts as they change.

logDigest (boolean) - default: false

Setting this to true will cause ng-stats to log out the digest lengths to the console. It will be colored green or red based on the digestTimeThreshold.

logWatches (boolean) - default: false

Setting this to true will cause ng-stats to log out the watch count to the console as it changes.

htmlId (string) - default: null

Sets an HTML ID attribute to the rendered stats element.

rootScope (object) - default: undefined

Passes the $rootScope to ng-stats. This parameter is only required for Ionic support where the ng-scope and ng-isolate-scope classes are removed. The only way of using the ng-stats with Ionic is invoking showAngularStats( { options } ) in your code and passing the $rootScope manually.

Module

Simply declare it as a dependency angular.module('your-mod', ['angularStats']);

Then use the directive:

<div angular-stats watch-count=".watch-count" digest-length=".digest-length"

on-watch-count-update="onWatchCountUpdate(watchCount)"

on-digest-length-update="onDigestLengthUpdate(digestLength)">

Watch Count: <span class="watch-count"></span><br />

Digest Cycle Length: <span class="digest-length"></span>

</div>

angular-stats attributes

**angular-stats :** The directive itself. No value is expected

**watch-count :** Having this attribute will keep track of the watch count and update the text of a specified element. Possible values are:

1. Selector for a child element to update
2. no value - refers to the current element (updates the text of the current element)

**watch-count-root**

**angular-stats** defaults to keeping track of the watch count for the whole page, however if you want to keep track of a specific element (and its children), provide this with a element query selector. As a convenience, if this is provided then the watch-count-root will be set to the element itself. Also, if you want to scope the query selector to the element, add watch-count-of-child as an attribute (no value)

**on-watch-count-update**

Because of the performance implications of calculating the watch count, this is not called every digest but a maximum of once every 300ms. Still avoid invoking another digest here though. The name of the variable passed is watchCount (like you see in the example).

**digest-length**

This works similar to the watch-count attribute. It's presence will cause the directive to keep track of the digest-length and will update the text of a specified element (rounds to two decimal places). Possible values are:

1. Selector for a child element to update
2. no value - refers to the current element (updates the text of the current element)

**on-digest-length-update**

Pass an expression to evaluate with every digest length update. This gets called on every digest (so be sure you don't invoke another digest in this handler or you'll get an infinite loop of doom). The name of the variable passed is digestLength (as in the example).

**Roadmap**

* Add analysis to highlight areas on the page that have highest watch counts.
* Somehow find out which watches are taking the longest... Ideas on implementation are welcome...
* See what could be done with the new scoped digest coming in Angular version 1.3.
* Count the number of digests or provide some analytics for frequency?
* Create a Chrome Extension for the chart or integrate with [batarang](https://github.com/angular/batarang)?
* Other ideas?
* Other notes

**Performance impact**

This will not impact the speed of your application at all until you actually use it. It also will hopefully only negatively impact your app's performance minimally. This is intended to be used in development only for debugging purposes so it shouldn't matter much anyway. It should be noted that calculating the watch count can be pretty expensive, so it's throttled to be calculated a minimum of 300ms.

**Using in an iframe**

Thanks to [this brilliant PR](https://github.com/kentcdodds/ng-stats/pull/25) from [@jinyangzhen](https://github.com/jinyangzhen), you can run ng-stats in an iframe (like plunker!). See the PR for an example of how to accomplish this.



## ****How do you share data between controllers?****

Create an AngularJS service that will hold the data and inject it inside of the controllers.

Using a service is the cleanest, fastest and easiest way to test.  
However, there are couple of other ways to implement data sharing between controllers, like:  
– Using events  
– Using $parent, nextSibling, controllerAs, etc. to directly access the controllers  
– Using the $rootScope to add the data on (not a good practice)

## ****What is the difference between**** ng-show****/****ng-hide ****and**** ng-if ****directives?****

ng-show/ng-hide will always insert the DOM element, but will display/hide it based on the condition. ng-if will not insert the DOM element until the condition is not fulfilled.

ng-if is better when we needed the DOM to be loaded conditionally, as it will help load page bit faster compared to ng-show/ng-hide.

We only need to keep in mind what the difference between these directives is, so deciding which one to use totally depends on the task requirements.

## ****What is a digest cycle in AngularJS?****

In each digest cycle Angular compares the old and the new version of the scope model values. The digest cycle is triggered automatically. We can also use $apply() if we want to trigger the digest cycle manually.

**The Digest Loop and $apply**

Let’s take a peek at how Angular works underneath the hood. How do we get this magical data binding to work in only a few lines of code? It’s important that we understand how the $digest loop works and how to use the $apply() method.

In the normal browser flow, a browser executes callbacks that are registered with an event when that event occurs (e.g., clicking on a link).

Events are fired when the page loads, when an $http request comes back, when the mouse moves or a button is clicked, etc.

When an event is fired/triggered, JavaScript creates an event object and executes any functions listening for the specific events with this event object. This callback method then runs inside the JavaScript function, which returns to the browser, potentially updating the DOM.

No two events can run at the same time. The browser waits until one event handler finishes before the next handler is called.

In non-Angular JavaScript, we can attach a function callback to the click event on a div. Any time that a click event is found on an element, the function callback runs:

var div = document.getElementById("clickDiv");

div.addEventListener("click",

function(evt) {

console.log("evt", evt);

});

Open the Chrome developer tools, and copy and paste the previous code inside of any web page and click around the page.

Any time the browser detects a click, the browser calls the function registered with the addEventListener on the document.

When we mix Angular into the flow, it extends this normal browser flow to create an Angular context. The Angular context refers specifically to code that runs inside the Angular event loop, referred to as the $digest loop. To understand the Angular context, we need to look at exactly what goes on inside of it. There are two major components of the $digestloop:

* The $watch list
* The $evalAsync list

$watch List

Every time we track an event in the view, we are registering a callback function that we expect to be called when an event happens in the page. Recall our first example:

<!DOCTYPE html>

**<html** ng-app**>**

**<head>**

**<title>**Simple app**</title>**

**<script**

src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/angular.js"**>**

**</script>**

**</head>**

**<body>**

**<input** ng-model="name" type="text" placeholder="Your name"**>**

**<h1>**Hello {﻿{ name }﻿}**</h1>**

**</body>**

**</html>**

Any time a user updates the input field, {﻿{ name }﻿} changes in the UI. This change happens because we bind the input field in the UI to the $scope.name property. In order to update the view, Angular needs to track the change. It does so by adding a watch function to the $watch list.

Properties that are on the $scope object are only bound if they are used in the view. In the case above, we’ve added a single function to the $watch list.

Remember: For all UI elements that are bound to a $scope object, a $watch is added to the $watch list.

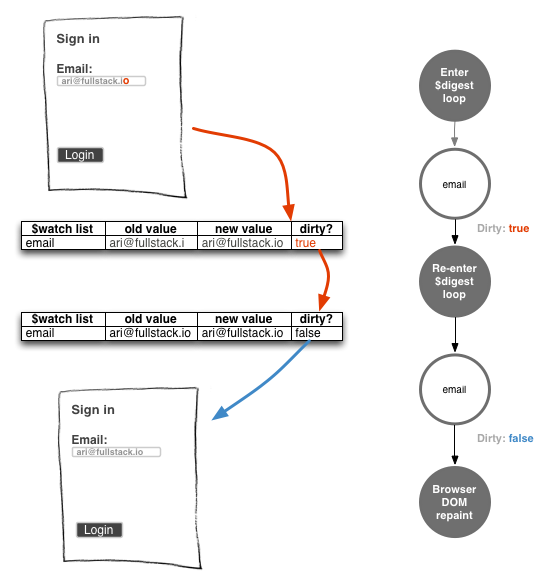
These $watch lists are resolved in the $digest loop through a process called dirty checking.

## Dirty Checking

Dirty checking is a simple process that boils down to a very basic concept: It checks whether a value has changed that hasn’t yet been synchronized across the app.

The dirty checking strategy is commonly used in plenty of different applications, beyond Angular. Game engines, database engines, and Object Relational Mappers (ORMs) are some examples of such systems.

Our Angular app keeps track of the values of the current watches (in the watch object, for those who are curious). Angular walks down the $watch list, and, if the updated value has not changed from the old value, it continues down the list. If the value has changed, the app records the new value and continues down the $watch list.



Once Angular has run through the entire $watch list, if any value changed, the app will fall back into the $watch loop until it detects that nothing has changed.

Why run the loop all over again? If we update a value in the $watch list that updates another value, Angular won’t detect the update unless we rerun the loop.

If the loop runs ten times or more, our Angular app throws an exception, and the app dies. If Angular doesn’t throw this exception, our app could launch into an infinite loop, with bad results.

In future versions of Angular, the framework will use the native browser specification Object.observe(), which will speed up the dirty checking process considerably.

## $watch

The $watch method on the $scope object sets up a dirty check on every call to $digest inside the Angular event loop. The Angular $digest loop always returns if it detects changes on the expression.

The $watch function itself takes two required arguments and a third optional one:

* watchExpression

The watchExpression can either be a property of a scope object or a function. It runs on **every** call to $digest in the $digest loop.

If the watchExpression is a string, Angular evaluates it in the context of the $scope. If it is a function, then Angular expects it to return the value that should be watched.

* listener/callback

The callback listener function is only called when the current value of the watchExpression and the previous value of the expression are not equal (except during initialization on the first run).

* objectEquality (optional)

The objectEquality parameter is a comparison boolean that tells Angular to check for strict equality.

The $watch function returns a deregistration function for the listener that we can call to cancel Angular’s watch on the value.

// ...

**var** unregisterWatch =

**$scope**.**$watch**('newUser.email',

**function**(newVal, oldVal) {

**if** (newVal === oldVal) **return**; // on init

});

// ...

// later, we can unregister this watcher

// by calling

unregisterWatch();

If we are done watching the newUser.email in this example, we can clean up our watcher by calling the deregistration function it returns.

For instance, let’s say we want to parse an input field value from a full name to split on spaces and find a simple first and last name. Given that our view looks like:

**<input** type="text" ng-model="full\_name" placeholder="Enter your full name" **/>**

We should **never** use $watch in a controller: It makes it difficult to test the controller. We’re making an allowance here for the sake of illustration, and we’ll move these watches into services later.

We want to set up a $watch listener on the full\_name property and detect any changes to the value. We also want to set the $watch function on the full\_name property.

angular.module("myApp")

.controller("MyController", ['$scope', **function**(**$scope**) {

**$scope**.**$watch**('full\_name', **function**(newVal, oldVal, scope) {

// the newVal of the full\_name will be available here

// while the oldVal is the old value of full\_name

});

}]);

In our example, we’re setting an AngularJS expression that tells our Angular app to “watch the full\_name property for any potential changes on it, and run the function if you detect any changes”.

The listener function is called once on initialization, so the first time around, the value of newVal and oldVal will be undefined (and will be equal). That being the case, it’s generally good to check inside the expression if we’re in the initialization phase or if there is an update to the previous value. We can easily accomplish this check inside the function, like so:

**$scope**.**$watch**('full\_name',

**function**(newVal, oldVal, scope) {

**if** (newVal === oldVal) {

// This will run only on the initialization of the watcher

} **else** {

// A change has occurred after initialization

}

});

The $scope.$watch() function sets up a watchExpression on the $scope for ‘full\_name’.

$apply() and $digest() are two core, and sometimes confusing, aspects of AngularJS. To understand how AngularJS works one needs to fully understand how $apply() and $digest() work. This article aims to explain what $apply() and $digest() really are, and how they can be useful in your day-to-day AngularJS programming.

## $apply and $digest Explored

AngularJS offers an incredibly awesome feature known as two way data binding which greatly simplifies our lives. Data binding means that when you change something in the view, the scope model automagically updates. Similarly, whenever the scope model changes, the view updates itself with the new value. How does does AngularJS do that? When you write an expression ({{aModel}}), behind the scenes Angular sets up a watcher on the scope model, which in turn updates the view whenever the model changes. This watcher is just like any watcher you set up in AngularJS:

$scope.$watch('aModel', function(newValue, oldValue) {

//update the DOM with newValue

});

The second argument passed to $watch() is known as a listener function, and is called whenever the value of aModel changes. It is easy for us to grasp that when the value of aModel changes this listener is called, updating the expression in HTML. But, there is still one big question! How does Angular figure out when to call this listener function? In other words, how does AngularJS know when aModel changes so it can call the corresponding listener? Does it run a function periodically to check whether the value of the scope model has changed? Well, this is where the $digest cycle steps in.

It’s the $digest cycle where the watchers are fired. When a watcher is fired, AngularJS evaluates the scope model, and if it has changed then the corresponding listener function is called. So, our next question is when and how this $digest cycle starts.

The $digest cycle starts as a result of a call to $scope.$digest(). Assume that you change a scope model in a handler function through the ng-click directive. In that case AngularJS automatically triggers a $digest cycle by calling $digest(). When the $digest cycle starts, it fires each of the watchers. These watchers check if the current value of the scope model is different from last calculated value. If yes, then the corresponding listener function executes. As a result if you have any expressions in the view they will be updated. In addition to ng-click, there are several other built-in directives/services that let you change models (e.g. ng-model, $timeout, etc) and automatically trigger a $digest cycle.

So far, so good! But, there is a small gotcha. In the above cases, Angular doesn’t directly call $digest(). Instead, it calls $scope.$apply(), which in turn calls $rootScope.$digest(). As a result of this, a digest cycle starts at the $rootScope, and subsequently visits all the child scopes calling the watchers along the way.

Now, let’s assume you attach an ng-click directive to a button and pass a function name to it. When the button is clicked, AngularJS wraps the function call within $scope.$apply(). So, your function executes as usual, change models (if any), and a $digest cycle starts to ensure your changes are reflected in the view.

**Note**: $scope.$apply() automatically calls $rootScope.$digest(). The $apply() function comes in two flavors. The first one takes a function as an argument, evaluates it, and triggers a $digest cycle. The second version does not take any arguments and just starts a $digest cycle when called. We will see why the former one is the preferred approach shortly.

## When Do You Call $apply() Manually?

If AngularJS usually wraps our code in $apply() and starts a $digest cycle, then when do you need to do call $apply() manually? Actually, AngularJS makes one thing pretty clear. It will account for only those model changes which are done inside AngularJS’ context (i.e. the code that changes models is wrapped inside $apply()). Angular’s built-in directives already do this so that any model changes you make are reflected in the view. However, if you change any model outside of the Angular context, then you need to inform Angular of the changes by calling $apply() manually. It’s like telling Angular that you are changing some models and it should fire the watchers so that your changes propagate properly.

For example, if you use JavaScript’s setTimeout() function to update a scope model, Angular has no way of knowing what you might change. In this case it’s your responsibility to call $apply() manually, which triggers a $digest cycle. Similarly, if you have a directive that sets up a DOM event listener and changes some models inside the handler function, you need to call $apply() to ensure the changes take effect.

Let’s look at an example. Suppose you have a page, and once the page loads you want to display a message after a two second delay. Your implementation might look something like the JavaScript and HTML shown in the following listing.

<body ng-app="myApp">  
 <div ng-controller="MessageController">  
 Delayed Message: {{message}}  
 </div>   
 </body>

/\* What happens without an $apply() \*/  
   
 angular.module('myApp',[]).controller('MessageController', function($scope) {  
   
 $scope.getMessage = function() {  
 setTimeout(function() {  
 $scope.message = 'Fetched after 3 seconds';  
 console.log('message:'+$scope.message);  
 }, 2000);  
 }

By running the example, you will see that the delayed function runs after a two second interval, and updates the scope model message. Still, the view doesn’t update. The reason, as you may have guessed, is that we forgot to call $apply() manually. Therefore, we need to update our getMessage() function as shown below.

<body ng-app="myApp">  
 <div ng-controller="MessageController">  
 Delayed Message: {{message}}  
 </div>   
 </body>

/\* What happens with $apply \*/   
angular.module('myApp',[]).controller('MessageController', function($scope) {  
   
 $scope.getMessage = function() {  
 setTimeout(function() {  
 $scope.$apply(function() {  
 //wrapped this within $apply  
 $scope.message = 'Fetched after 3 seconds';   
 console.log('message:' + $scope.message);  
 });  
 }, 2000);  
 }  
   
 $scope.getMessage();  
   
 });

If you run this updated example, you can see the view update after two seconds. The only change is that we wrapped our code inside $scope.$apply() which automatically triggers $rootScope.$digest(). As a result the watchers are fired as usual and the view updates.

**Note**: By the way, you should use $timeout service whenever possible which is setTimeout() with automatic $apply() so that you don’t have to call $apply() manually.

Also, note that in the above code you could have done the model changes as usual and placed a call to $apply() (the no-arg version) in the end. Have a look at the following snippet:

$scope.getMessage = function() {

setTimeout(function() {

$scope.message = 'Fetched after two seconds';

console.log('message:' + $scope.message);

$scope.$apply(); //this triggers a $digest

}, 2000);

};

The above code uses the no-arg version of $apply() and works. Keep in mind that you should always use the version of $apply() that accepts a function argument. This is because when you pass a function to $apply(), the function call is wrapped inside a try...catch block, and any exceptions that occur will be passed to the $exceptionHandler service.

## How Many Times Does the $digest Loop Run?

When a $digest cycle runs, the watchers are executed to see if the scope models have changed. If they have, then the corresponding listener functions are called. This leads to an important question. What if a listener function itself changed a scope model? How would AngularJS account for that change?

The answer is that the $digest loop doesn’t run just once. At the end of the current loop, it starts all over again to check if any of the models have changed. This is basically dirty checking, and is done to account for any model changes that might have been done by listener functions. So, the $digest cycle keeps looping until there are no more model changes, or it hits the max loop count of 10. It’s always good to stay idempotent and try to minimize model changes inside the listener functions.

**Note**: At a minimum, $digest will run twice even if your listener functions don’t change any models. As discussed above, it runs once more to make sure the models are stable and there are no changes.

## Conclusion

I hope this article has clarified what $apply and $digest are all about. The most important thing to keep in mind is whether or not Angular **can** detect your changes. If it cannot, then you must call $apply() manually.

You're going to explore the Angular digest cycle, which is the process behind Angular data binding. By the end of this assignment, you'll understand what happens when you call $scope.$apply() to kick off a digest cycle. And understanding the digest cycle will allow you to write more complex directives.

**Scope, Binding, and $watch(ing)**

Scope is just a key-value pair storage system. You can refer to keys from a scope in your HTML templates in order to dynamically generate content in your views. In the simple example below, there's an ng-if directive in your template to conditionally display a paragraph if the scope variable enabled evaluates to true (note that for demonstration purposes all code has been put in a single HTML file here, which is *not*what you should do in your own code, and this is also working with .run() and $rootScope which you normally wouldn't do for something like this).

<!DOCTYPE html>

<html lang="en" ng-app="myApp">

<head>

<meta charset="UTF-8">

<title>Document</title>

</head>

<script src="https://code.angularjs.org/1.2.9/angular.min.js"></script>

<script>

angular.module('myApp', [])

.run(function($rootScope){

var $scope = $rootScope;

$scope.$watch('enabled', function(val) {

console.log('You are now: ' + (val ? 'enabled' : 'disabled'));

});

$scope.enabled = true;

$scope.enabled = false;

$scope.enabled = 1;

})

</script>

<body>

<p ng-if="enabled">I am here because I'm enabled</p>

<button ng-click="enabled=!enabled">Click Me!</button>

</body>

</html>

If you paste this code into a text editor, save it as an HTML file, and preview in your browser, you'll see that the behavior is in line with examples you've come across before. Initially, you see the "enabled" paragraph because $scope.enabled is initially set to true. Use the ng-click event on the button to toggle $scope.enabled's value. When $scope.enabled gets toggled to false, the paragraph text gets removed from the rendered view.

There's nothing surprising here in the behavior this code produces, but if you stop for a moment and ask *how* this behavior happens, the answer is not apparent. You know that the ng-if paragraph is effectively bound to the value of enabled, but when the value of enabled changes, how does the browser know that it needs to re-render the DOM to reflect the new scope value? How does the ng-if paragraph get to know about the change in the value of enabled?

The answer to this lies in Angular's internal *digest cycle* and what are called *watchers*. Let's address *watchers* first. You use $scope.watch() to register a watcher, and this method takes two parameters: the expression to watch, and the *listener function* to when there is a change to the watched expression.

In the ng-if example above, you're relying on watchers, even though you don't have to write this code yourself. Behind the scenes, ng-if registers a watcher on whatever expression it has been set to (in the case above, it's been set to whatever $scope.enabled evaluates to). Here's what a *very* simplified version of the source code for ng-if would look like:

.directive('ngIf', function() {

return function(scope, element, attrs) {

scope.$watch(attrs.ngIf, function expressionCallback(val) {

if(val) {

if(!element.parent()) {

//add element to page

}

} else if(element.parent()) {

//remove element from page

}

});

}

});

Stepping through this code you see that ng-if has a function that sets a $watch on the expression set in the ng-if attribute. If the value of that expression is *truthy*, and if the element with ng-if has no parent (e.g., it's not part of the DOM), it gets added to the page. If the value of the expression is *falsey*, the element gets removed from the page.

Okay, that explains what watchers are, but how does this fit into the *digest cycle*? Angular's digest cycle can be called on any scope object (be it rootScope or a child scope in a controller). When $scope.digest() gets called, Angular goes through all of the watchers that have been registered in the current scope and any child scopes. For each watcher, if the watched value has changed since the previous digest cycle, the watcher's listener function gets called. If the watched value has not changed, Angular simply moves on to the next watcher in the list. You may see this process referred to as *dirty checking*.

After Angular reaches the last watcher that has been registered in the scope, the digest cycle is not quite done. Angular runs through all of the watchers again to see if any changed values in the first journey through the list of watchers has caused any other watched expressions to change. Angular will continue running through the list of watchers up to 10 times. If it finds that no values have changed, the digest cycle is complete, and the DOM will be updated accordingly. If at the end of 10 trips through there are still changed values, Angular will throw an error.

Okay. So you now know what happens when $scope.digest() gets called, but how does $scope.digest() get called in the first place? The answer is: it depends. Sometimes you need to manually trigger the digest cycle by calling $scope.$apply() (the best practice) or $scope.digest() (which as you'll see, gets called by $scope.$apply()). Let's look at manual triggers for the digest cycle first.

**Manually Triggering the Digest Cycle**

You may have already encountered an example of $scope.$apply() when you implemented your own version of ng-click. Here's the code again for that directive:

directive('myClick', function() {

return function(scope, element, attrs) {

element.on('click', function() {

scope.$apply(function() {

//fire the onClick function

scope.$eval(attrs.myClick);

});

});

}

});

Recall that reason you needed to call $scope.$apply() here was to get Angular to run the callback function. Without $scope.$apply(), the click event would fire, but Angular would not register the $scope.eval() event. You were told that this was because when you want Angular to respond to a change in the world "outside" of Angular, you need to manually tell Angular about it.

Now you're in a position to understand what $scope.$apply() really does. The answer is that it runs the function you supply as a parameter, and then behind the scenes calls $scope.digest(), which triggers the digest cycle and the dirty checking described above. So, for instance, the following code

$scope.$apply(doSomething)

is nearly equivalent to

doSomething();

$scope.digest();

It's referred to as *nearly* here because the first approach is considered best practice. When you pass a function into $scope.$apply(), it automatically is wrapped in a try...catch block that gets passed on to an exception handler service. This means that if something goes wrong with the function call, it will bubble up through Angular. If you call your function outside of $scope.$apply() you won't get this error handling for free.

Let's have another look at the first code example you started out with in this reading to get these ideas to sink in. You're going to take the code you had before and register a watcher that logs to the console when the value of the *enabled* variable has changed:

<!DOCTYPE html>

<html lang="en" ng-app="myApp">

<head>

<meta charset="UTF-8">

<title>Document</title>

</head>

<script src="https://code.angularjs.org/1.2.9/angular.min.js"></script>

<script>

angular.module('myApp', [])

.run(function($rootScope){

var $scope = $rootScope;

$scope.$watch('enabled', function(val) {

console.log('You are now: ' + (val ? 'enabled' : 'disabled'));

});

$scope.enabled = true;

$scope.enabled = false;

$scope.enabled = 1;

})

</script>

<body>

<p ng-if="enabled">I am here because I'm enabled</p>

<button ng-click="enabled=!enabled">Click Me!</button>

</body>

</html>

If you save this code in an HTML file, then preview it in Chrome with the JavaScript console, you can see the results when the page loads. At first glance, you might expect to see the following in your console:

You are now: enabled

You are now: disabled

You are now: enabled

But in fact, what you actually find is a single "You are now: enabled". This is because in the code above, the digest cycle runs only once after the initial state has been set on $scope.enabled. The controller code simply sets, then overwrites, then overwrites the value of of enabled again. All three of these assignment statements get executed *before* the digest cycle gets triggered, which happens automatically after Angular has loaded any controllers.

To get the console output you're looking for, you can rewrite the controller code like this:

<!DOCTYPE html>

<html lang="en" ng-app="myApp">

<head>

<meta charset="UTF-8">

<title>Document</title>

</head>

<script src="https://code.angularjs.org/1.2.9/angular.min.js"></script>

<script>

angular.module('myApp', [])

.run(function($rootScope){

var $scope = $rootScope;

$scope.$watch('enabled', function(val) {

console.log('You are now: ' + (val ? 'enabled' : 'disabled'));

});

$scope.$apply(function() {

$scope.enabled = true;

});

$scope.$apply(function() {

$scope.enabled = false;

});

$scope.$apply(function() {

$scope.enabled = 1;

});

})

</script>

<body>

<p ng-if="enabled">I am here because I'm enabled</p>

<button ng-click="enabled=!enabled">Click Me!</button>

</body>

</html>

If you run this code, you'll see that you now get the expected output.

**Automatic Digest Triggers**

Now that you have an idea of how watchers and the digest cycle work, let's consider the cases where you *don't* need to manually trigger Angular's digest cycle. These cases are summarized in the table below:

| **Event** | **Why?** |
| --- | --- |
| An AJAX request completes | If an HTTP request is run via the $http service then Angular will trigger a digest cycle once a response is obtained. |
| A timer completes | When $timeout and/or $interval is used then Angular will trigger a digest cycle once the timeout is complete or each time the interval is run. |
| User interaction | When ngModel is used on a form input element then whenever the user changes the value, inputs new data or submits the form then a digest cycle will be run. |
| A link is clicked | When a link is clicked then a new route will be loaded. For this to happen Angular must place a click listener on the root of the browser body and then run a digest cycle each time a click operation happens on a link. |

What each of the events in this table have in common is that they are asynchronous. Since AngularJS has no idea when an asynchronous operation is complete (let alone started) then a digest cycle can't be run automatically. Thus to make things easier for the developer, special services such as $http and $timeout are provided to encapsulate the asynchronous operation to keep track of when it starts and ends.

Therefore, if you're using an Angular service that performs an asynchronous operation, chances are that you don't need to call $scope.$apply(). However, if you're listening on DOM events or waiting for an external event to complete itself then you'll need to run $scope.$apply() on your own.

**Watching !expressions**

In all of the examples you've encountered, you've had the watchers look for changes to an expression, represented by a string value. However, there are two other options you should know about.

First, you can watch a function:

$scope.$watch(function() { return someVal; }, function() {

//someCallback

});

This listener function here (someCallback) would be called any time function() { return someVal; } returns a value that is different than the previous one.

It is also possible to watch for changes to collections of objects. For instance, to listen for changes to an array, you would use $scope.$watchCollection():

var array = [1,2,3,4,5];

$scope.$watchCollection(function() { return array; }, function(updatedArray) {

//the array has been updated

console.log('updated array: ', updatedArray);

});

array.push(6);

array[0] = 100;

Take a look at a working plunkr [example](http://plnkr.co/edit/n2hGl1xcZ43XghPkMzyM?p=preview);

Finally, it's also possible to register a watcher that listens for changes to a group of expressions, using $scope.$watchGroup():

var values = ['firstName', 'lastName', 'email'];

$scope.$watchGroup(values, function(values) {

//the details for the user have been updated

});

The listener function here will be executed when any of the three expressions change.

SCOPES LIFE CYCLE

* Scopes provide APIs ([$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch)) to observe model mutations.
* Scopes provide APIs ([$apply](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply)) to propagate any model changes through the system into the view from outside of the "AngularJS realm" (controllers, services, AngularJS event handlers).
* Scopes can be nested to limit access to the properties of application components while providing access to shared model properties. Nested scopes are either "child scopes" or "isolate scopes". A "child scope" (prototypically) inherits properties from its parent scope. An "isolate scope" does not. See [isolated scopes](https://docs.angularjs.org/guide/directive#isolating-the-scope-of-a-directive) for more information.
* Scopes provide context against which [expressions](https://docs.angularjs.org/guide/expression) are evaluated. For example {{username}} expression is meaningless, unless it is evaluated against a specific scope which defines the username property.

Scope is the glue between application controller and the view. During the template [linking](https://docs.angularjs.org/guide/compiler) phase the [directives](https://docs.angularjs.org/api/ng/provider/$compileProvider#directive) set up [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) expressions on the scope. The $watch allows the directives to be notified of property changes, which allows the directive to render the updated value to the DOM.

Both controllers and directives have reference to the scope, but not to each other. This arrangement isolates the controller from the directive as well as from the DOM. This is an important point since it makes the controllers view agnostic, which greatly improves the testing story of the applications.

Each AngularJS application has exactly one [root scope](https://docs.angularjs.org/api/ng/service/$rootScope), but may have any number of child scopes.

The application can have multiple scopes, because [directives](https://docs.angularjs.org/guide/directive) can create new child scopes. When new scopes are created, they are added as children of their parent scope. This creates a tree structure which parallels the DOM where they're attached.

The section [Directives that Create Scopes](https://docs.angularjs.org/guide/scope#directives-that-create-scopes) has more info about which directives create scopes.

When AngularJS evaluates {{name}}, it first looks at the scope associated with the given element for the name property. If no such property is found, it searches the parent scope and so on until the root scope is reached. In JavaScript this behavior is known as prototypical inheritance, and child scopes prototypically inherit from their parents.

he normal flow of a browser receiving an event is that it executes a corresponding JavaScript callback. Once the callback completes the browser re-renders the DOM and returns to waiting for more events.

When the browser calls into JavaScript the code executes outside the AngularJS execution context, which means that AngularJS is unaware of model modifications. To properly process model modifications the execution has to enter the AngularJS execution context using the [$apply](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply) method. Only model modifications which execute inside the $apply method will be properly accounted for by AngularJS. For example if a directive listens on DOM events, such as [ng-click](https://docs.angularjs.org/api/ng/directive/ngClick) it must evaluate the expression inside the $applymethod.

After evaluating the expression, the $apply method performs a [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest). In the $digest phase the scope examines all of the $watchexpressions and compares them with the previous value. This dirty checking is done asynchronously. This means that assignment such as $scope.username="angular" will not immediately cause a $watch to be notified, instead the $watch notification is delayed until the $digest phase. This delay is desirable, since it coalesces multiple model updates into one $watch notification as well as guarantees that during the $watch notification no other $watches are running. If a $watch changes the value of the model, it will force additional$digest cycle.

1. **Creation**

The [root scope](https://docs.angularjs.org/api/ng/service/$rootScope) is created during the application bootstrap by the [$injector](https://docs.angularjs.org/api/auto/service/$injector). During template linking, some directives create new child scopes.

1. **Watcher registration**

During template linking, directives register [watches](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) on the scope. These watches will be used to propagate model values to the DOM.

1. **Model mutation**

For mutations to be properly observed, you should make them only within the [scope.$apply()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply). AngularJS APIs do this implicitly, so no extra $apply call is needed when doing synchronous work in controllers, or asynchronous work with [$http](https://docs.angularjs.org/api/ng/service/$http), [$timeout](https://docs.angularjs.org/api/ng/service/$timeout) or [$interval](https://docs.angularjs.org/api/ng/service/$interval) services.

1. **Mutation observation**

At the end of $apply, AngularJS performs a [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) cycle on the root scope, which then propagates throughout all child scopes. During the $digest cycle, all $watched expressions or functions are checked for model mutation and if a mutation is detected, the $watch listener is called.

1. **Scope destruction**

When child scopes are no longer needed, it is the responsibility of the child scope creator to destroy them via [scope.$destroy()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$destroy) API. This will stop propagation of $digest calls into the child scope and allow for memory used by the child scope models to be reclaimed by the garbage collector.

### **Scopes and Directives**

During the compilation phase, the [compiler](https://docs.angularjs.org/guide/compiler) matches [directives](https://docs.angularjs.org/api/ng/provider/$compileProvider#directive) against the DOM template. The directives usually fall into one of two categories:

* Observing [directives](https://docs.angularjs.org/api/ng/provider/$compileProvider#directive), such as double-curly expressions {{expression}}, register listeners using the [$watch()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) method. This type of directive needs to be notified whenever the expression changes so that it can update the view.
* Listener directives, such as [ng-click](https://docs.angularjs.org/api/ng/directive/ngClick), register a listener with the DOM. When the DOM listener fires, the directive executes the associated expression and updates the view using the [$apply()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply) method.

When an external event (such as a user action, timer or XHR) is received, the associated [expression](https://docs.angularjs.org/guide/expression) must be applied to the scope through the [$apply()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply) method so that all listeners are updated correctly.

### **Directives that Create Scopes**

In most cases, [directives](https://docs.angularjs.org/api/ng/provider/$compileProvider#directive) and scopes interact but do not create new instances of scope. However, some directives, such as [ng-controller](https://docs.angularjs.org/api/ng/directive/ngController) and [ng-repeat](https://docs.angularjs.org/api/ng/directive/ngRepeat), create new child scopes and attach the child scope to the corresponding DOM element.

A special type of scope is the isolate scope, which does not inherit prototypically from the parent scope. This type of scope is useful for component directives that should be isolated from their parent scope. See the [directives guide](https://docs.angularjs.org/guide/directive#isolating-the-scope-of-a-directive) for more information about isolate scopes in custom directives.

Note also that component directives, which are created with the [.component()](https://docs.angularjs.org/api/ng/type/angular.Module#component) helper always create an isolate scope.

### **Controllers and Scopes**

Scopes and controllers interact with each other in the following situations:

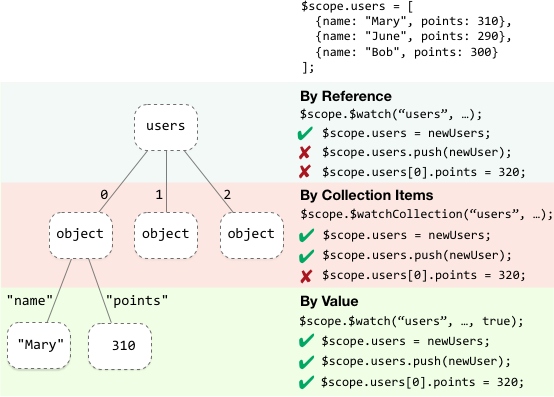
* Controllers use scopes to expose controller methods to templates (see [ng-controller](https://docs.angularjs.org/api/ng/directive/ngController)).
* Controllers define methods (behavior) that can mutate the model (properties on the scope).
* Controllers may register [watches](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) on the model. These watches execute immediately after the controller behavior executes.

See the [ng-controller](https://docs.angularjs.org/api/ng/directive/ngController) for more information.

### **Scope $watch Performance Considerations**

Dirty checking the scope for property changes is a common operation in AngularJS and for this reason the dirty checking function must be efficient. Care should be taken that the dirty checking function does not do any DOM access, as DOM access is orders of magnitude slower than property access on JavaScript object.

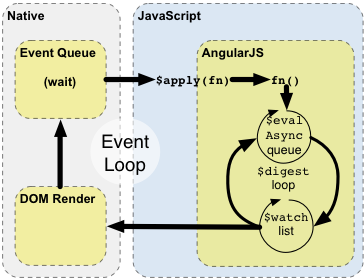
### **Scope $watch Depths**



Dirty checking can be done with three strategies: By reference, by collection contents, and by value. The strategies differ in the kinds of changes they detect, and in their performance characteristics.

* Watching by reference ([scope.$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch)(watchExpression, listener)) detects a change when the whole value returned by the watch expression switches to a new value. If the value is an array or an object, changes inside it are not detected. This is the most efficient strategy.
* Watching collection contents ([scope.$watchCollection](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watchCollection)(watchExpression, listener)) detects changes that occur inside an array or an object: When items are added, removed, or reordered. The detection is shallow - it does not reach into nested collections. Watching collection contents is more expensive than watching by reference, because copies of the collection contents need to be maintained. However, the strategy attempts to minimize the amount of copying required.
* Watching by value ([scope.$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) (watchExpression, listener, true)) detects any change in an arbitrarily nested data structure. It is the most powerful change detection strategy, but also the most expensive. A full traversal of the nested data structure is needed on each digest, and a full copy of it needs to be held in memory.

## Integration with the browser event loop



The diagram and the example below describe how AngularJS interacts with the browser's event loop.

1. The browser's event-loop waits for an event to arrive. An event is a user interaction, timer event, or network event (response from a server).
2. The event's callback gets executed. This enters the JavaScript context. The callback can modify the DOM structure.
3. Once the callback executes, the browser leaves the JavaScript context and re-renders the view based on DOM changes.

AngularJS modifies the normal JavaScript flow by providing its own event processing loop. This splits the JavaScript into classical and AngularJS execution context. Only operations which are applied in the AngularJS execution context will benefit from AngularJS data-binding, exception handling, property watching, etc... You can also use $apply() to enter the AngularJS execution context from JavaScript. Keep in mind that in most places (controllers, services) $apply has already been called for you by the directive which is handling the event. An explicit call to $apply is needed only when implementing custom event callbacks, or when working with third-party library callbacks.

1. Enter the AngularJS execution context by calling [scope](https://docs.angularjs.org/guide/scope).[$apply](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply)(stimulusFn), where stimulusFn is the work you wish to do in the AngularJS execution context.
2. AngularJS executes the stimulusFn(), which typically modifies application state.
3. AngularJS enters the [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) loop. The loop is made up of two smaller loops which process [$evalAsync](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$evalAsync) queue and the [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) list. The [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) loop keeps iterating until the model stabilizes, which means that the [$evalAsync](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$evalAsync) queue is empty and the [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) list does not detect any changes.
4. The [$evalAsync](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$evalAsync) queue is used to schedule work which needs to occur outside of current stack frame, but before the browser's view render. This is usually done with setTimeout(0), but the setTimeout(0) approach suffers from slowness and may cause view flickering since the browser renders the view after each event.
5. The [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) list is a set of expressions which may have changed since last iteration. If a change is detected then the $watchfunction is called which typically updates the DOM with the new value.
6. Once the AngularJS [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) loop finishes, the execution leaves the AngularJS and JavaScript context. This is followed by the browser re-rendering the DOM to reflect any changes.

Here is the explanation of how the Hello world example achieves the data-binding effect when the user enters text into the text field.

1. During the compilation phase:
   1. the [ng-model](https://docs.angularjs.org/api/ng/directive/ngModel) and [input](https://docs.angularjs.org/api/ng/directive/input) [directive](https://docs.angularjs.org/guide/directive) set up a keydown listener on the <input> control.
   2. the [interpolation](https://docs.angularjs.org/api/ng/service/$interpolate) sets up a [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) to be notified of name changes.
2. During the runtime phase:
   1. Pressing an 'X' key causes the browser to emit a keydown event on the input control.
   2. The [input](https://docs.angularjs.org/api/ng/directive/input) directive captures the change to the input's value and calls [$apply](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply)("name = 'X';") to update the application model inside the AngularJS execution context.
   3. AngularJS applies the name = 'X'; to the model.
   4. The [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) loop begins
   5. The [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) list detects a change on the name property and notifies the [interpolation](https://docs.angularjs.org/api/ng/service/$interpolate), which in turn updates the DOM.
   6. AngularJS exits the execution context, which in turn exits the keydown event and with it the JavaScript execution context.
   7. The browser re-renders the view with the updated text.

# $rootScope.Scope

1. **- type in module**[**ng**](https://docs.angularjs.org/api/ng)

A root scope can be retrieved using the [$rootScope](https://docs.angularjs.org/api/ng/service/$rootScope) key from the [$injector](https://docs.angularjs.org/api/auto/service/$injector). Child scopes are created using the [$new()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$new) method. (Most scopes are created automatically when compiled HTML template is executed.) See also the [Scopes guide](https://docs.angularjs.org/guide/scope) for an in-depth introduction and usage examples.

# Inheritance

A scope can inherit from a parent scope, as in this example:

var parent = $rootScope;

var child = parent.$new();

parent.salutation = "Hello";

expect(child.salutation).toEqual('Hello');

child.salutation = "Welcome";

expect(child.salutation).toEqual('Welcome');

expect(parent.salutation).toEqual('Hello');

When interacting with Scope in tests, additional helper methods are available on the instances of Scope type. See [ngMock Scope](https://docs.angularjs.org/api/ngMock/type/$rootScope.Scope) for additional details.

## Usage

$rootScope.Scope([providers], [instanceCache]);

### **Arguments**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| providers  (optional) | [**Object.<string, function()>=**](https://docs.angularjs.org/) | Map of service factory which need to be provided for the current scope. Defaults to [ng](https://docs.angularjs.org/api/ng). |
| instanceCache  (optional) | [**Object.<string, \*>=**](https://docs.angularjs.org/) | Provides pre-instantiated services which should append/override services provided by providers. This is handy when unit-testing and having the need to override a default service. |

### **Returns**

|  |  |
| --- | --- |
| [**Object**](https://docs.angularjs.org/) | Newly created scope. |

## Methods

* $new(isolate, parent);

Creates a new child [scope](https://docs.angularjs.org/api/ng/type/$rootScope.Scope).

The parent scope will propagate the [$digest()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) event. The scope can be removed from the scope hierarchy using [$destroy()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$destroy).

[$destroy()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$destroy) must be called on a scope when it is desired for the scope and its child scopes to be permanently detached from the parent and thus stop participating in model change detection and listener notification by invoking.

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| isolate | [**boolean**](https://docs.angularjs.org/) | If true, then the scope does not prototypically inherit from the parent scope. The scope is isolated, as it can not see parent scope properties. When creating widgets, it is useful for the widget to not accidentally read parent state. |
| parent  (optional) | [**Scope**](https://docs.angularjs.org/) | The [Scope](https://docs.angularjs.org/api/ng/type/$rootScope.Scope) that will be the $parent of the newly created scope. Defaults to this scope if not provided. This is used when creating a transclude scope to correctly place it in the scope hierarchy while maintaining the correct prototypical inheritance.  (default: this) |

#### **Returns**

|  |  |
| --- | --- |
| [**Object**](https://docs.angularjs.org/) | The newly created child scope. |

$watch(watchExpression, listener, [objectEquality]);

Registers a listener callback to be executed whenever the watchExpression changes.

* The watchExpression is called on every call to [$digest()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) and should return the value that will be watched. (watchExpressionshould not change its value when executed multiple times with the same input because it may be executed multiple times by [$digest()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest). That is, watchExpression should be [idempotent](http://en.wikipedia.org/wiki/Idempotence).)
* The listener is called only when the value from the current watchExpression and the previous call to watchExpression are not equal (with the exception of the initial run, see below). Inequality is determined according to reference inequality, [strict comparison](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators)via the !== Javascript operator, unless objectEquality == true (see next point)
* When objectEquality == true, inequality of the watchExpression is determined according to the [angular.equals](https://docs.angularjs.org/api/ng/function/angular.equals) function. To save the value of the object for later comparison, the [angular.copy](https://docs.angularjs.org/api/ng/function/angular.copy) function is used. This therefore means that watching complex objects will have adverse memory and performance implications.
* This should not be used to watch for changes in objects that are or contain [File](https://developer.mozilla.org/docs/Web/API/File) objects due to limitations with [angular.copy](https://docs.angularjs.org/api/ng/function/angular.copy).
* The watch listener may change the model, which may trigger other listeners to fire. This is achieved by rerunning the watchers until no changes are detected. The rerun iteration limit is 10 to prevent an infinite loop deadlock.

If you want to be notified whenever [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) is called, you can register a watchExpression function with no listener. (Be prepared for multiple calls to your watchExpression because it will execute multiple times in a single [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) cycle if a change is detected.)

After a watcher is registered with the scope, the listener fn is called asynchronously (via [$evalAsync](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$evalAsync)) to initialize the watcher. In rare cases, this is undesirable because the listener is called when the result of watchExpression didn't change. To detect this scenario within the listener fn, you can compare the newVal and oldVal. If these two values are identical (===) then the listener was called due to initialization.

# Example

* // let's assume that scope was dependency injected as the $rootScope
* var scope = $rootScope;
* scope.name = 'misko';
* scope.counter = 0;
* expect(scope.counter).toEqual(0);
* scope.$watch('name', function(newValue, oldValue) {
* scope.counter = scope.counter + 1;
* });
* expect(scope.counter).toEqual(0);
* scope.$digest();
* // the listener is always called during the first $digest loop after it was registered
* expect(scope.counter).toEqual(1);
* scope.$digest();
* // but now it will not be called unless the value changes
* expect(scope.counter).toEqual(1);
* scope.name = 'adam';
* scope.$digest();
* expect(scope.counter).toEqual(2);
* // Using a function as a watchExpression
* var food;
* scope.foodCounter = 0;
* expect(scope.foodCounter).toEqual(0);
* scope.$watch(
* // This function returns the value being watched. It is called for each turn of the $digest loop
* function() { return food; },
* // This is the change listener, called when the value returned from the above function changes
* function(newValue, oldValue) {
* if ( newValue !== oldValue ) {
* // Only increment the counter if the value changed
* scope.foodCounter = scope.foodCounter + 1;
* }
* }
* );
* // No digest has been run so the counter will be zero
* expect(scope.foodCounter).toEqual(0);
* // Run the digest but since food has not changed count will still be zero
* scope.$digest();
* expect(scope.foodCounter).toEqual(0);
* // Update food and run digest. Now the counter will increment
* food = 'cheeseburger';
* scope.$digest();

expect(scope.foodCounter).toEqual(1);

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| watchExpression | [**function()string**](https://docs.angularjs.org/) | Expression that is evaluated on each [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) cycle. A change in the return value triggers a call to the listener.   * + string: Evaluated as [expression](https://docs.angularjs.org/guide/expression)   + function(scope): called with current scope as a parameter. |
| listener | [**function(newVal, oldVal, scope)**](https://docs.angularjs.org/) | Callback called whenever the value of watchExpression changes.   * + newVal contains the current value of the watchExpression   + oldVal contains the previous value of the watchExpression   + scope refers to the current scope |
| objectEquality  (optional) | [**boolean**](https://docs.angularjs.org/) | Compare for object equality using [angular.equals](https://docs.angularjs.org/api/ng/function/angular.equals) instead of comparing for reference equality.  (default: false) |

#### **Returns**

|  |  |
| --- | --- |
| [**function()**](https://docs.angularjs.org/) | Returns a deregistration function for this listener. |

* $watchGroup(watchExpressions, listener);

A variant of [$watch()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) where it watches an array of watchExpressions. If any one expression in the collection changes the listener is executed.

* + The items in the watchExpressions array are observed via the standard $watch operation. Their return values are examined for changes on every call to $digest.
  + The listener is called whenever any expression in the watchExpressions array changes.

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| watchExpressions | [**Array.<string|Function(scope)>**](https://docs.angularjs.org/) | Array of expressions that will be individually watched using [$watch()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) |
| listener | [**function(newValues, oldValues, scope)**](https://docs.angularjs.org/) | Callback called whenever the return value of any expression in watchExpressions changes The newValues array contains the current values of the watchExpressions, with the indexes matching those of watchExpression and the oldValues array contains the previous values of the watchExpressions, with the indexes matching those of watchExpression The scope refers to the current scope. |

#### **Returns**

|  |  |
| --- | --- |
| [**function()**](https://docs.angularjs.org/) | Returns a de-registration function for all listeners. |

* $watchCollection(obj, listener);

Shallow watches the properties of an object and fires whenever any of the properties change (for arrays, this implies watching the array items; for object maps, this implies watching the properties). If a change is detected, the listener callback is fired.

* + The obj collection is observed via standard $watch operation and is examined on every call to $digest() to see if any items have been added, removed, or moved.
  + The listener is called whenever anything within the obj has changed. Examples include adding, removing, and moving items belonging to an object or array.

# Example

$scope.names = ['igor', 'matias', 'misko', 'james'];

$scope.dataCount = 4;

$scope.$watchCollection('names', function(newNames, oldNames) {

$scope.dataCount = newNames.length;

});

expect($scope.dataCount).toEqual(4);

$scope.$digest();

//still at 4 ... no changes

expect($scope.dataCount).toEqual(4);

$scope.names.pop();

$scope.$digest();

//now there's been a change

expect($scope.dataCount).toEqual(3);

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| obj | [**stringfunction(scope)**](https://docs.angularjs.org/) | Evaluated as [expression](https://docs.angularjs.org/guide/expression). The expression value should evaluate to an object or an array which is observed on each [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) cycle. Any shallow change within the collection will trigger a call to the listener. |
| listener | [**function(newCollection, oldCollection, scope)**](https://docs.angularjs.org/) | a callback function called when a change is detected.   * + The newCollection object is the newly modified data obtained from the obj expression   + The oldCollection object is a copy of the former collection data. Due to performance considerations, theoldCollection value is computed only if the listener function declares two or more arguments.   + The scope argument refers to the current scope. |

#### **Returns**

|  |  |
| --- | --- |
| [**function()**](https://docs.angularjs.org/) | Returns a de-registration function for this listener. When the de-registration function is executed, the internal watch operation is terminated. |

* $digest();

Processes all of the [watchers](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) of the current scope and its children. Because a [watcher](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch)'s listener can change the model, the $digest()keeps calling the [watchers](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) until no more listeners are firing. This means that it is possible to get into an infinite loop. This function will throw 'Maximum iteration limit exceeded.' if the number of iterations exceeds 10.

Usually, you don't call $digest() directly in [controllers](https://docs.angularjs.org/api/ng/directive/ngController) or in [directives](https://docs.angularjs.org/api/ng/provider/$compileProvider#directive). Instead, you should call [$apply()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply) (typically from within a [directive](https://docs.angularjs.org/api/ng/provider/$compileProvider#directive)), which will force a $digest().

If you want to be notified whenever $digest() is called, you can register a watchExpression function with [$watch()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) with no listener.

In unit tests, you may need to call $digest() to simulate the scope life cycle.

# Example

var scope = ...;

scope.name = 'misko';

scope.counter = 0;

expect(scope.counter).toEqual(0);

scope.$watch('name', function(newValue, oldValue) {

scope.counter = scope.counter + 1;

});

expect(scope.counter).toEqual(0);

scope.$digest();

// the listener is always called during the first $digest loop after it was registered

expect(scope.counter).toEqual(1);

scope.$digest();

// but now it will not be called unless the value changes

expect(scope.counter).toEqual(1);

scope.name = 'adam';

scope.$digest();

expect(scope.counter).toEqual(2);

* $destroy();

Removes the current scope (and all of its children) from the parent scope. Removal implies that calls to [$digest()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) will no longer propagate to the current scope and its children. Removal also implies that the current scope is eligible for garbage collection.

The $destroy() is usually used by directives such as [ngRepeat](https://docs.angularjs.org/api/ng/directive/ngRepeat) for managing the unrolling of the loop.

Just before a scope is destroyed, a $destroy event is broadcasted on this scope. Application code can register a $destroy event handler that will give it a chance to perform any necessary cleanup.

Note that, in AngularJS, there is also a $destroy jQuery event, which can be used to clean up DOM bindings before an element is removed from the DOM.

* $eval([expression], [locals]);

Executes the expression on the current scope and returns the result. Any exceptions in the expression are propagated (uncaught). This is useful when evaluating AngularJS expressions.

# Example

var scope = ng.$rootScope.Scope();

scope.a = 1;

scope.b = 2;

expect(scope.$eval('a+b')).toEqual(3);

expect(scope.$eval(function(scope){ return scope.a + scope.b; })).toEqual(3);

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| expression  (optional) | [**stringfunction()**](https://docs.angularjs.org/) | An AngularJS expression to be executed.   * + string: execute using the rules as defined in [expression](https://docs.angularjs.org/guide/expression).   + function(scope): execute the function with the current scope parameter. |
| locals  (optional) | [**object**](https://docs.angularjs.org/) | Local variables object, useful for overriding values in scope. |

#### **Returns**

|  |  |
| --- | --- |
| [**\***](https://docs.angularjs.org/) | The result of evaluating the expression. |

* $evalAsync([expression], [locals]);

Executes the expression on the current scope at a later point in time.

The $evalAsync makes no guarantees as to when the expression will be executed, only that:

* + it will execute after the function that scheduled the evaluation (preferably before DOM rendering).
  + at least one [$digest cycle](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) will be performed after expression execution.

Any exceptions from the execution of the expression are forwarded to the [$exceptionHandler](https://docs.angularjs.org/api/ng/service/$exceptionHandler) service.

**Note:** if this function is called outside of a $digest cycle, a new $digest cycle will be scheduled. However, it is encouraged to always call code that changes the model from within an $apply call. That includes code evaluated via $evalAsync.

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| expression  (optional) | [**stringfunction()**](https://docs.angularjs.org/) | An AngularJS expression to be executed.   * + string: execute using the rules as defined in [expression](https://docs.angularjs.org/guide/expression).   + function(scope): execute the function with the current scope parameter. |
| locals  (optional) | [**object**](https://docs.angularjs.org/) | Local variables object, useful for overriding values in scope. |

* $apply([exp]);

$apply() is used to execute an expression in AngularJS from outside of the AngularJS framework. (For example from browser DOM events, setTimeout, XHR or third party libraries). Because we are calling into the AngularJS framework we need to perform proper scope life cycle of [exception handling](https://docs.angularjs.org/api/ng/service/$exceptionHandler), [executing watches](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest).

## Life cycle

# Pseudo-Code of $apply()

function $apply(expr) {

try {

return $eval(expr);

} catch (e) {

$exceptionHandler(e);

} finally {

$root.$digest();

}

}

Scope's $apply() method transitions through the following stages:

* + The [expression](https://docs.angularjs.org/guide/expression) is executed using the [$eval()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$eval) method.
  + Any exceptions from the execution of the expression are forwarded to the [$exceptionHandler](https://docs.angularjs.org/api/ng/service/$exceptionHandler) service.
  + The [watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) listeners are fired immediately after the expression was executed using the [$digest()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) method.

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| exp  (optional) | [**stringfunction()**](https://docs.angularjs.org/) | An AngularJS expression to be executed.   * + string: execute using the rules as defined in [expression](https://docs.angularjs.org/guide/expression).   + function(scope): execute the function with current scope parameter. |

#### **Returns**

|  |  |
| --- | --- |
| [**\***](https://docs.angularjs.org/) | The result of evaluating the expression. |

* $applyAsync([exp]);

Schedule the invocation of $apply to occur at a later time. The actual time difference varies across browsers, but is typically around ~10 milliseconds.

This can be used to queue up multiple expressions which need to be evaluated in the same digest.

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| exp  (optional) | [**stringfunction()**](https://docs.angularjs.org/) | An AngularJS expression to be executed.   * + string: execute using the rules as defined in [expression](https://docs.angularjs.org/guide/expression).   + function(scope): execute the function with current scope parameter. |

* $on(name, listener);

Listens on events of a given type. See [$emit](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$emit) for discussion of event life cycle.

The event listener function format is: function(event, args...). The event object passed into the listener has the following attributes:

* + targetScope - {Scope}: the scope on which the event was $emit-ed or $broadcast-ed.
  + currentScope - {Scope}: the scope that is currently handling the event. Once the event propagates through the scope hierarchy, this property is set to null.
  + name - {string}: name of the event.
  + stopPropagation - {function=}: calling stopPropagation function will cancel further event propagation (available only for events that were $emit-ed).
  + preventDefault - {function}: calling preventDefault sets defaultPrevented flag to true.
  + defaultPrevented - {boolean}: true if preventDefault was called.

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| name | [**string**](https://docs.angularjs.org/) | Event name to listen on. |
| listener | [**function(event, ...args)**](https://docs.angularjs.org/) | Function to call when the event is emitted. |

#### **Returns**

|  |  |
| --- | --- |
| [**function()**](https://docs.angularjs.org/) | Returns a deregistration function for this listener. |

* $emit(name, args);

Dispatches an event name upwards through the scope hierarchy notifying the registered [$rootScope.Scope](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$on) listeners.

The event life cycle starts at the scope on which $emit was called. All [listeners](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$on) listening for name event on this scope get notified. Afterwards, the event traverses upwards toward the root scope and calls all registered listeners along the way. The event will stop propagating if one of the listeners cancels it.

Any exception emitted from the [listeners](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$on) will be passed onto the [$exceptionHandler](https://docs.angularjs.org/api/ng/service/$exceptionHandler) service.

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| name | [**string**](https://docs.angularjs.org/) | Event name to emit. |
| args | [**\***](https://docs.angularjs.org/) | Optional one or more arguments which will be passed onto the event listeners. |

#### **Returns**

|  |  |
| --- | --- |
| [**Object**](https://docs.angularjs.org/) | Event object (see [$rootScope.Scope](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$on)). |

* $broadcast(name, args);

Dispatches an event name downwards to all child scopes (and their children) notifying the registered [$rootScope.Scope](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$on) listeners.

The event life cycle starts at the scope on which $broadcast was called. All [listeners](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$on) listening for name event on this scope get notified. Afterwards, the event propagates to all direct and indirect scopes of the current scope and calls all registered listeners along the way. The event cannot be canceled.

Any exception emitted from the [listeners](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$on) will be passed onto the [$exceptionHandler](https://docs.angularjs.org/api/ng/service/$exceptionHandler) service.

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| name | [**string**](https://docs.angularjs.org/) | Event name to broadcast. |
| args | [**\***](https://docs.angularjs.org/) | Optional one or more arguments which will be passed onto the event listeners. |

#### **Returns**

|  |  |
| --- | --- |
| [**Object**](https://docs.angularjs.org/) | Event object, see [$rootScope.Scope](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$on) |

## Events

### **$destroy**

Broadcasted when a scope and its children are being destroyed.

Note that, in AngularJS, there is also a $destroy jQuery event, which can be used to clean up DOM bindings before an element is removed from the DOM.

#### **Type:**

broadcast

#### **Target:**

scope being destroyed

**Where should we implement the DOM manipulation in AngularJS?**

In the directives. DOM Manipulations should not exist in controllers, services or anywhere else but in directives.

[Best Practice - Dom Manipulations](http://ng-learn.org/2014/01/Dom-Manipulations/)

Lets talk about best practices and roles in AngularJS. A controller is the middle man. Its main role is to talk to the Service to get the model and then make sure this model is available to our presentation layer(html). Even in large applications, the controller should be small, compact and dumb!

Dom Manipulations should not exist in controllers, services or anywhere else but in directives.

This came up today. We have two products with its own results list controller.

productAResultsCtrl.coffee and productBResultsCtrl.coffee were both presenting this code:

$(".search-details-form").hide()

$("#more").click ->

if $(".search-details-form").is(":hidden")

$(".search-details-form").slideDown()

else

$(".search-details-form").slideUp()

This is meant to hide a search bar with advanced options and if the user clicks on a button we display all the options with a classic jquery slide effect.

This works but

* It is not reusable
* It is not testable
* It include css hard coded selectors dependencies

So in order to resolve these problems, we should always:

* write a directive for dom manipulation
* the directive should not have dependencies on other html blocks or scope’s parents objects
* the directive should not have any hard coded css selector

We are going to write a directive to be applied in the div that should slide down or up based on an external variable.

To avoid dependency problems, we are going to pass this variable as a parameter. We dont care who changes this variable. We only care to react when the variable changes.

Our directive should look something like this:

<div data-my-slide="showDetails"> details content goes here</div>

We use ‘data-’ prefix to make sure html validates and we do not cause IE7, 8 browsers to go into quirks mode. Very Important!!!!

We use ‘my-’ prefix to determine this is a custom directive that belongs to our team.

Lets write the code for this. As always the test comes first:

describe("Unit testing jquery directive", function() {

var $compile, $scope, element;

$scope = element = $compile = void 0;

// Load the module, which contains the directive

beforeEach(module("jqueryDirectives"));

// Store references to $rootScope and $compile so they are available to all tests in this describe block

beforeEach(inject(function(\_$compile\_, \_$rootScope\_) {

// The injector unwraps the underscores (\_) from around the parameter names when matching

$scope = \_$rootScope\_;

return $compile = \_$compile\_;

}));

it("should slide Down a block", function() {

// Create html fragment

element = angular.element('<div class="form" data-my-slide="showForm">Text</div>');

// Set variable

$scope.showForm = true;

// Compile a piece of HTML containing the directive

$compile(element)($scope);

$scope.$digest();

// Set expectation

return expect(element.css('height')).toBe('1px');

});

it("should slide Up a block", function() {

// Create html fragment

element = angular.element('<div class="form" data-my-slide="showForm">Text</div>');

// Set variable

$scope.showForm = false;

// Compile a piece of HTML containing the directive

$compile(element)($scope);

$scope.$digest();

// Set expectation

return expect(element.css('height')).toBe('0px');

});

});

We have created two tests where after compiling the code, our directive applies slideDown or slideUp.

Based on jquery’s documentation the divs’ height gets primarily affected. We take this fact so as to evaluate the success of our directive.

Now lets write the directive

// Here we create a module to group these directives jquery related

var jqueryDirectives = angular.module("jqueryDirectives", []);

// Here we add a directive to the module. camelCase naming in this file (mySlide) and dash separated in html (my-Slide)

jqueryDirectives.directive("mySlide", [

function() {

return {

// This means the directive can be used as an attribute only. Example <div data-my-slide="variable"> </div>

restrict: "A",

// This is the functions that gets executed after Angular has compiled the html

link: function(scope, element, attrs) {

// We dont want to abuse on watch but here it is critical to determine if the parameter has changed.

scope.$watch(attrs.mySlide, function(newValue, oldValue) {

// This is our logic. If parameter is true slideDown otherwise slideUp.

// TODO: This should be transformed into css transition or angular animator if IE family supports it

if (newValue) {

return element.slideDown();

} else {

return element.slideUp();

}

});

}

};

}

]);

So now on both productAResults.html and productBResults.html we say

<div class="search-details-form" data-my-slide="showRedoSearchDetails">

And we remove the jquery dom maniuplations out of our controllers.

Now we have a reusable, independent, tested directive.

## ****Is it a good or bad practice to use AngularJS together with jQuery?****

It is definitely a bad practice. We need to stay away from jQuery and try to realize the solution with an AngularJS approach. jQuery takes a traditional imperative approach to manipulating the DOM, and in an imperative approach, it is up to the programmer to express the individual steps leading up to the desired outcome.

AngularJS, however, takes a declarative approach to DOM manipulation. Here, instead of worrying about all of the step by step details regarding how to do the desired outcome, we are just declaring what we want and AngularJS worries about the rest, taking care of everything for us.

**Detailed explanations**

AngularJS shouldn't be looked at as a replacement to jQuery, in my opinion. Though if you use AngularJS and you are using jQuery as well (which you are allowed to do), that's a clear sign that you are probably using AngularJS incorrectly. So in that sense, I can say that you will naturally use one or the other but I wouldn't call it a replacement.   
  
With that said, before we look at when AngularJS should be used, it might be important to state why I think they are naturally mutually exclusive.  
  
**Declarative vs Imperative Approach to DOM Manipulation**jQuery takes a traditional imperative approach to manipulating the DOM. In an imperative approach, it is up to the programmer to express the individual steps leading up to the desired outcome. What do I mean by this? So if we want an action to occur when a user types say 150 characters into an input, in jQuery we would say, "every time the user hits a key, check how many characters are in the input, if it exceeds 150 characters, do the action." Every step is addressed along the way.  
  
AngularJS however takes a declarative approach to DOM manipulation. Here instead of worrying about all of the step by step details regarding how to do the desired outcome, AngularJS abstracts that and allows you to just say what you want done, in this case, "AngularJS, when the state of the input is at 150 characters, do this." We are just declaring what we want and AngularJS worries about the rest, taking care of everything for us.  
  
It might seem like I'm just splitting hairs here, but it's really an important distinction. AngularJS wants you basing your actions around the data models you create. It's how the entire framework works and how your applications will be structured. To simply begin writing side scripts in jQuery where you are plucking out elements and setting up side event listeners just goes against the AngularJS approach in my opinion.  
  
The AngularJS approach does have the one downside that most abstracted layers have, though it is simpler, it gives you less direct control. Often times the control is not needed and when it is there are things like Angular-UI and ngAnimate that often can solve the problem you're having in a more "Angular way", but in the case that it is required, AngularJS actually comes prepackaged with jqLite, which is a subset of jQuery. This was done intentionally to eliminate another dependency, which makes sense given that AngularJS is an attempt to be an all encompassing front-end framework. So this gives you even more reason not to use jQuery with AngularJS.   
  
With all of this said, if you still feel the need to use jQuery, all you need to do is include the jQuery script tag in your page before the AngularJS script tag. AngularJS is smart enough to know that you did this and it will switch to using jQuery for DOM manipulation. This alone shows you that they aren't inherent replacements of each other, but as per the above, they naturally don't go together in many ways.  
  
Okay so that was long winded... but now onto the question, why should you use AngularJS in the first place?  
  
**When to Use AngularJS... That is the Question.**The intention of jQuery is to be a library to simplify DOM manipulation. The intention of AngularJS is to be a full framework that provides structure and addresses all aspects of modern web applications that contain heavy front-ends, including DOM manipulation. So when do you use what?  
  
My opinion is that it depends on the type of application you are building. If you are building a very heavy front end, where most of the heavy lifting is going to be done in the front and all the back end is really going to be is a REST API that you can make calls to... than I would say a full framework like AngularJS (or Backbone.js or Ember.js or React.js) might be a good idea as it will help you with way more than just DOM manipulation.  
  
However if you are creating a good old fashioned site, with a very light front end where most of the work is being done in the back end, a large front end framework might be of much lesser use to you. You could still use it, but if the front end isn't doing a whole lot and is pretty small, it seems like it might not be worth it in my opinion.

1. **Don't design your page, and then change it with**[**DOM**](http://en.wikipedia.org/wiki/Document_Object_Model)**manipulations**

In jQuery, you design a page, and then you make it dynamic. This is because jQuery was designed for augmentation and has grown incredibly from that simple premise.  
  
But in AngularJS, you must start from the ground up with your architecture in mind. Instead of starting by thinking "I have this piece of the DOM and I want to make it do X", you have to start with what you want to accomplish, then go about designing your application, and then finally go about designing your view.

1. **Don't augment jQuery with AngularJS**

Similarly, don't start with the idea that jQuery does X, Y, and Z, so I'll just add AngularJS on top of that for models and controllers. This is *really* tempting when you're just starting out, which is why I always recommend that new AngularJS developers don't use jQuery at all, at least until they get used to doing things the "Angular Way".  
  
I've seen many developers here and on the mailing list create these elaborate solutions with jQuery plugins of 150 or 200 lines of code that they then glue into AngularJS with a collection of callbacks and $applys that are confusing and convoluted; but they eventually get it working! The problem is that in**most** cases that jQuery plugin could be rewritten in AngularJS in a fraction of the code, where suddenly everything becomes comprehensible and straightforward.  
  
The bottom line is this: when solutioning, first "think in AngularJS"; if you can't think of a solution, ask the community; if after all of that there is no easy solution, *then* feel free to reach for the jQuery. But don't let jQuery become a crutch or you'll never master AngularJS.

1. **Always think in terms of architecture**

First know that [single-page applications](http://en.wikipedia.org/wiki/Single-page_application) are applications. They're not webpages. So we need to think like a server-side developer in addition to thinking like a client-side developer. We have to think about how to divide our application into individual, extensible, testable components.  
  
So then how do you do that? How do you "think in AngularJS"? Here are some general principles, contrasted with jQuery.  
  
The view is the "official record"  
  
In jQuery, we programmatically change the view. We could have a dropdown menu defined as a ullike so:

<ul class="main-menu">

    <li class="active">

        <a href="#/home">Home</a>

    </li>

    <li>

        <a href="#/menu1">Menu 1</a>

        <ul>

            <li><a href="#/sm1">Submenu 1</a></li>

            <li><a href="#/sm2">Submenu 2</a></li>

            <li><a href="#/sm3">Submenu 3</a></li>

        </ul>

    </li>

    <li>

        <a href="#/home">Menu 2</a>

    </li>

</ul>

In jQuery, in our application logic, we would activate it with something like:

$('.main-menu').dropdownMenu();

When we just look at the view, it's not immediately obvious that there is any functionality here. For small applications, that's fine. But for non-trivial applications, things quickly get confusing and hard to maintain.  
In AngularJS, though, the view is the official record of view-based functionality. Our ul declaration would look like this instead:

<ul class="main-menu" dropdown-menu> ...</ul>

These two do the same thing, but in the AngularJS version anyone looking at the template knows what's supposed to happen. Whenever a new member of the development team comes on board, she can look at this and then know that there is a directive called dropdownMenu operating on it; she doesn't need to intuit the right answer or sift through any code. The view told us what was supposed to happen. Much cleaner.  
  
Developers new to AngularJS often ask a question like: how do I find all links of a specific kind and add a directive onto them. The developer is always flabbergasted when we reply: you don't. But the reason you don't do that is that this is like half-jQuery, half-AngularJS, and no good. The problem here is that the developer is trying to "do jQuery" in the context of AngularJS. That's never going to work well. The view is the official record. Outside of a directive (more on this below), you never, ever, neverchange the DOM. And directives are applied in the view, so intent is clear.  
  
Remember: don't design, and then mark up. You must architect, and then design.  
  
Data binding : This is by far one of the most awesome features of AngularJS and cuts out a lot of the need to do the kinds of DOM manipulations I mentioned in the previous section. AngularJS will automatically update your view so you don't have to! In jQuery, we respond to events and then update content. Something like:

$.ajax({  url: '/myEndpoint.json',  success: function ( data, status ) {

    $('ul#log').append('<li>Data Received!</li>');

  }

});

For a view that looks like this:  
  
**<ul** class="messages" id="log"**></ul>**  
  
Apart from mixing concerns, we also have the same problems of signifying intent that I mentioned before. But more importantly, we had to manually reference and update a DOM node. And if we want to delete a log entry, we have to code against the DOM for that too. How do we test the logic apart from the DOM? And what if we want to change the presentation?  
  
This a little messy and a trifle frail. But in AngularJS, we can do this:  
  
$http( '/myEndpoint.json' ).then( function ( response ) {

    $scope.log.push( { msg: 'Data Received!' } );

});

And our view can look like this:  
  
<ul class="messages">

    <li ng-repeat="entry in log">{{ entry.msg }}</li>

</ul>

But for that matter, our view could look like this:  
  
<div class="messages">

    <div class="alert" ng-repeat="entry in log">

        {{ entry.msg }}

     </div>

</div>

And now instead of using an unordered list, we're using Bootstrap alert boxes. And we never had to change the controller code! But more importantly, no matter *where* or *how* the log gets updated, the view will change too. Automatically. Neat!  
  
Though I didn't show it here, the data binding is two-way. So those log messages could also be editable in the view just by doing this: **<input** ng-model="entry.msg" **/>**. And there was much rejoicing.  
  
**Distinct model layer**  
  
In jQuery, the DOM is kind of like the model. But in AngularJS, we have a separate model layer that we can manage in any way we want, completely independently from the view. This helps for the above data binding, maintains [separation of concerns](http://en.wikipedia.org/wiki/Separation_of_concerns), and introduces far greater testability. Other answers mentioned this point, so I'll just leave it at that.  
  
**Separation of concerns**  
  
And all of the above tie into this over-arching theme: keep your concerns separate. Your view acts as the official record of what is supposed to happen (for the most part); your model represents your data; you have a service layer to perform reusable tasks; you do DOM manipulation and augment your view with directives; and you glue it all together with controllers. This was also mentioned in other answers, and the only thing I would add pertains to testability, which I discuss in another section below.  
  
**Dependency injection**  
  
To help us out with separation of concerns is [dependency injection](http://en.wikipedia.org/wiki/Dependency_injection) (DI). If you come from a server-side language (from [Java](http://en.wikipedia.org/wiki/Java_%28programming_language%29) to [PHP](http://en.wikipedia.org/wiki/PHP)) you're probably familiar with this concept already, but if you're a client-side guy coming from jQuery, this concept can seem anything from silly to superfluous to hipster. But it's not. :-)  
  
From a broad perspective, DI means that you can declare components very freely and then from any other component, just ask for an instance of it and it will be granted. You don't have to know about loading order, or file locations, or anything like that. The power may not immediately be visible, but I'll provide just one (common) example: testing.  
  
Let's say in our application, we require a service that implements server-side storage through a [REST](http://en.wikipedia.org/wiki/Representational_State_Transfer) API and, depending on application state, local storage as well. When running tests on our controllers, we don't want to have to communicate with the server - we're testing the *controller*, after all. We can just add a mock service of the same name as our original component, and the injector will ensure that our controller gets the fake one automatically - our controller doesn't and needn't know the difference.  
  
Speaking of testing...

## 4. Test-driven development - *always*

This is really part of section 3 on architecture, but it's so important that I'm putting it as its own top-level section.  
Out of all of the many jQuery plugins you've seen, used, or written, how many of them had an accompanying test suite? Not very many because jQuery isn't very amenable to that. But AngularJS is.  
  
In jQuery, the only way to test is often to create the component independently with a sample/demo page against which our tests can perform DOM manipulation. So then we have to develop a component separately and *then* integrate it into our application. How inconvenient! So much of the time, when developing with jQuery, we opt for iterative instead of test-driven development. And who could blame us?  
  
But because we have separation of concerns, we can do test-driven development iteratively in AngularJS! For example, let's say we want a super-simple directive to indicate in our menu what our current route is. We can declare what we want in the view of our application:  
  
**<a** href="/hello" when-active**>**Hello**</a>**  
  
Okay, now we can write a test for the non-existent when-active directive:  
  
it( 'should add "active" when the route changes', inject(function() {

    var elm = $compile( '<a href="/hello" when-active>Hello</a>' )( $scope );

    $location.path('/not-matching');

    expect( elm.hasClass('active') ).toBeFalsey();

    $location.path( '/hello' );

    expect( elm.hasClass('active') ).toBeTruthy();

}));

And when we run our test, we can confirm that it fails. Only now should we create our directive:  
  
.directive( 'whenActive', function ( $location ) {

    return {

        scope: true,

        link: function ( scope, element, attrs ) {

            scope.$on( '$routeChangeSuccess', function () {

                if ( $location.path() == element.attr( 'href' ) ) {

                    element.addClass( 'active' );

                }

                else {

                    element.removeClass( 'active' );

                }

            });

        }

    };

});

Our test now passes *and* our menu performs as requested. Our development is *both* iterative *and*test-driven. Wicked-cool.

## 5. Conceptually, directives are *not* packaged jQuery

You'll often hear "only do DOM manipulation in a directive". **This is a necessity.** Treat it with due deference!  
  
But let's dive a little deeper...  
  
Some directives just decorate what's already in the view (think ngClass) and therefore sometimes do DOM manipulation straight away and then are basically done. But if a directive is like a "widget" and has a template, it should *also* respect separation of concerns. That is, the template *too* should remain largely independent from its implementation in the link and controller functions.  
  
AngularJS comes with an entire set of tools to make this very easy; with ngClass we can dynamically update the class; ngModel allows two-way data binding; ngShow and ngHide programmatically show or hide an element; and many more - including the ones we write ourselves. In other words, we can do all kinds of awesomeness *without* DOM manipulation. The less DOM manipulation, the easier directives are to test, the easier they are to style, the easier they are to change in the future, and the more re-usable and distributable they are.  
  
I see lots of developers new to AngularJS using directives as the place to throw a bunch of jQuery. In other words, they think "since I can't do DOM manipulation in the controller, I'll take that code put it in a directive". While that certainly is much better, it's often *still wrong*.  
  
Think of the logger we programmed in section 3. Even if we put that in a directive, we *still* want to do it the "Angular Way". It *still* doesn't take any DOM manipulation! There are lots of times when DOM manipulation is necessary, but it's a *lot* rarer than you think! Before doing DOM manipulation *anywhere* in your application, ask yourself if you really need to. There might be a better way.  
  
Here's a quick example that shows the pattern I see most frequently. We want a toggle-able button. (Note: this example is a little contrived and a skosh verbose to represent more complicated cases that are solved in exactly the same way.)  
  
.directive( 'myDirective', function () {

    return {

        template: '<a class="btn">Toggle me!</a>',

        link: function ( scope, element, attrs ) {

            var on = false;

            $(element).click( function () {

                on = !on;

                $(element).toggleClass('active', on);

            });

        }

    };

});

There are a few things wrong with this:

1. First, jQuery was never necessary. There's nothing we did here that needed jQuery at all!
2. Second, even if we already have jQuery on our page, there's no reason to use it here; we can simply use angular.element and our component will still work when dropped into a project that doesn't have jQuery.
3. Third, even assuming jQuery *was* required for this directive to work, jqLite (angular.element) will *always* use jQuery if it was loaded! So we needn't use the $- we can just use angular.element.
4. Fourth, closely related to the third, is that jqLite elements needn't be wrapped in $- the element that is passed to the link function would *already be* a jQuery element!
5. And fifth, which we've mentioned in previous sections, why are we mixing template stuff into our logic?

This directive can be rewritten (even for very complicated cases!) much more simply like so:  
  
.directive( 'myDirective', function () {

    return {

        scope: true,

        template: '<a class="btn" ng-class="{active: on}" ng-click="toggle()">Toggle me!</a>',

        link: function ( scope, element, attrs ) {

            scope.on = false;

            scope.toggle = function () {

                scope.on = !scope.on;

            };

        }

    };

});

Again, the template stuff is in the template, so you (or your users) can easily swap it out for one that meets any style necessary, and the **logic** never had to be touched. Reusability - boom!  
  
And there are still all those other benefits, like testing - it's easy! No matter what's in the template, the directive's internal API is never touched, so refactoring is easy. You can change the template as much as you want without touching the directive. And no matter what you change, your tests still pass.  
  
w00t!  
  
So if directives aren't just collections of jQuery-like functions, what are they? Directives are actually **extensions of HTML**. If HTML doesn't do something you need it to do, you write a directive to do it for you, and then use it just as if it was part of HTML.  
  
Put another way, if AngularJS doesn't do something out of the box, think how the team would accomplish it to fit right in with ngClick, ngClass, et al.

## Summary

Don't even use jQuery. Don't even include it. It will hold you back. And when you come to a problem that you think you know how to solve in jQuery already, before you reach for the $, try to think about how to do it within the confines the AngularJS. If you don't know, ask! 19 times out of 20, the best way to do it doesn't need jQuery and to try to solve it with jQuery results in more work for you.

## ****If you were to migrate from Angular 1.4 to Angular 1.5, what is the main thing that would need refactoring?****

Changing .directive to .component to adapt to the new Angular 1.5 components

## **How would you specify that a scope variable should have one-time binding only?**

By using “::” in front of it. This allows the check if the candidate is aware of the available variable bindings in AngularJS.

## ****What is the difference between one-way binding and two-way binding?****

– One way binding implies that the scope variable in the html will be set to the first value its model is bound to (i.e. assigned to)  
– Two way binding implies that the scope variable will change it’s value everytime its model is assigned to a different value

## ****Explain how**** $scope.$apply() ****works****

$scope.$apply re-evaluates all the declared ng-models and applies the change to any that have been altered (i.e. assigned to a new value)  
Explanation: $scope.$apply() is one of the core angular functions that should never be used explicitly, it forces the angular engine to run on all the watched variables and all external variables and apply the changes on their values

## ****What directive would you use to hide elements from the HTML DOM by removing them from that DOM not changing their styling?****

The ngIf Directive, when applied to an element, will remove that element from the DOM if it’s condition is false.

## **What makes the** angular.copy() **method so powerful?**

It creates a deep copy of the variable.

A deep copy of a variable means it doesn’t point to the same memory reference as that variable. Usually assigning one variable to another creates a “shallow copy”, which makes the two variables point to the same memory reference. Therefore if we change one, the other changes as well

angular.copy:Creates a deep copy of source, which should be an object or an array.

* If no destination is supplied, a copy of the object or array is created.
* If a destination is provided, all of its elements (for arrays) or properties (for objects) are deleted and then all elements/properties from the source are copied to it.
* If source is not an object or array (inc. null and undefined), source is returned.
* If source is identical to destination an exception will be thrown.

Only enumerable properties are taken into account. Non-enumerable properties (both on source and on destination) will be ignored.

## Usage

angular.copy(source, [destination]);

### **Arguments**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| source | [**\***](https://docs.angularjs.org/) | The source that will be used to make a copy. Can be any type, including primitives, null, and undefined. |
| destination  (optional) | [**ObjectArray**](https://docs.angularjs.org/) | Destination into which the source is copied. If provided, must be of the same type as source. |

### **Returns**

|  |  |
| --- | --- |
| [**\***](https://docs.angularjs.org/) | The copy or updated destination, if destination was specified. |

## Examples

[**index.html**](https://docs.angularjs.org/)

<div ng-controller="ExampleController">

<form novalidate class="simple-form">

<label>Name: <input type="text" ng-model="user.name" /></label><br />

<label>Age: <input type="number" ng-model="user.age" /></label><br />

Gender: <label><input type="radio" ng-model="user.gender" value="male" />male</label>

<label><input type="radio" ng-model="user.gender" value="female" />female</label><br />

<button ng-click="reset()">RESET</button>

<button ng-click="update(user)">SAVE</button>

</form>

<pre>form = {{user | json}}</pre>

<pre>master = {{master | json}}</pre>

</div>

-----script code-------

// Module: copyExample

angular.

module('copyExample', []).

controller('ExampleController', ['$scope', function($scope) {

$scope.master = {};

$scope.reset = function() {

// Example with 1 argument

$scope.user = angular.copy($scope.master);

};

$scope.update = function(user) {

// Example with 2 arguments

angular.copy(user, $scope.master);

};

$scope.reset();

}]);

## ****How would you make an Angular service return a promise? Write a code snippet as an example**** to add promise functionality to a service, we inject the “$q” dependency in the service, and then use it like so:

angular.factory('testService', function($q){

return {

getName: function(){

var deferred = $q.defer();

//API call here that returns data

testAPI.getName().then(function(name){

deferred.resolve(name)

})

return deferred.promise;

}

}

})

The $q library is a helper provider that implements promises and deferred objects to enable asynchronous functionality

**$q :** A service that helps you run functions asynchronously, and use their return values (or exceptions) when they are done processing.

This is a [Promises/A+](https://promisesaplus.com/)-compliant implementation of promises/deferred objects inspired by [Kris Kowal's Q](https://github.com/kriskowal/q).

$q can be used in two fashions --- one which is more similar to Kris Kowal's Q or jQuery's Deferred implementations, and the other which resembles ES6 (ES2015) promises to some degree.

**$q constructor :** The streamlined ES6 style promise is essentially just using $q as a constructor which takes a resolver function as the first argument. This is similar to the native Promise implementation from ES6, see [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise).

While the constructor-style use is supported, not all of the supporting methods from ES6 promises are available yet.

It can be used like so:

// for the purpose of this example let's assume that variables `$q` and `okToGreet`

// are available in the current lexical scope (they could have been injected or passed in).

function asyncGreet(name) {

// perform some asynchronous operation, resolve or reject the promise when appropriate.

return $q(function(resolve, reject) {

setTimeout(function() {

if (okToGreet(name)) {

resolve('Hello, ' + name + '!');

} else {

reject('Greeting ' + name + ' is not allowed.');

}

}, 1000);

});

}

var promise = asyncGreet('Robin Hood');

promise.then(function(greeting) {

alert('Success: ' + greeting);

}, function(reason) {

alert('Failed: ' + reason);

});

Note: progress/notify callbacks are not currently supported via the ES6-style interface.

Note: unlike ES6 behavior, an exception thrown in the constructor function will NOT implicitly reject the promise.

However, the more traditional CommonJS-style usage is still available, and documented below.

[The CommonJS Promise proposal](http://wiki.commonjs.org/wiki/Promises) describes a promise as an interface for interacting with an object that represents the result of an action that is performed asynchronously, and may or may not be finished at any given point in time.

From the perspective of dealing with error handling, deferred and promise APIs are to asynchronous programming what try, catch and throw keywords are to synchronous programming.

// for the purpose of this example let's assume that variables `$q` and `okToGreet`

// are available in the current lexical scope (they could have been injected or passed in).

function asyncGreet(name) {

var deferred = $q.defer();

setTimeout(function() {

deferred.notify('About to greet ' + name + '.');

if (okToGreet(name)) {

deferred.resolve('Hello, ' + name + '!');

} else {

deferred.reject('Greeting ' + name + ' is not allowed.');

}

}, 1000);

return deferred.promise;

}

var promise = asyncGreet('Robin Hood');

promise.then(function(greeting) {

alert('Success: ' + greeting);

}, function(reason) {

alert('Failed: ' + reason);

}, function(update) {

alert('Got notification: ' + update);

});

At first it might not be obvious why this extra complexity is worth the trouble. The payoff comes in the way of guarantees that promise and deferred APIs make, see <https://github.com/kriskowal/uncommonjs/blob/master/promises/specification.md>.

Additionally the promise api allows for composition that is very hard to do with the traditional callback ([CPS](http://en.wikipedia.org/wiki/Continuation-passing_style)) approach. For more on this please see the [Q documentation](https://github.com/kriskowal/q) especially the section on serial or parallel joining of promises.

**The Deferred API**

A new instance of deferred is constructed by calling $q.defer().

The purpose of the deferred object is to expose the associated Promise instance as well as APIs that can be used for signaling the successful or unsuccessful completion, as well as the status of the task.

**Methods**

* resolve(value) – resolves the derived promise with the value. If the value is a rejection constructed via $q.reject, the promise will be rejected instead.
* reject(reason) – rejects the derived promise with the reason. This is equivalent to resolving it with a rejection constructed via $q.reject.
* notify(value) - provides updates on the status of the promise's execution. This may be called multiple times before the promise is either resolved or rejected.

**Properties**

* promise – {Promise} – promise object associated with this deferred.

**The Promise API**

A new promise instance is created when a deferred instance is created and can be retrieved by calling deferred.promise.

The purpose of the promise object is to allow for interested parties to get access to the result of the deferred task when it completes.

**Methods**

* then(successCallback, [errorCallback], [notifyCallback]) – regardless of when the promise was or will be resolved or rejected, then calls one of the success or error callbacks asynchronously as soon as the result is available. The callbacks are called with a single argument: the result or rejection reason. Additionally, the notify callback may be called zero or more times to provide a progress indication, before the promise is resolved or rejected.

This method returns a new promise which is resolved or rejected via the return value of the successCallback, errorCallback(unless that value is a promise, in which case it is resolved with the value which is resolved in that promise using [promise chaining](http://www.html5rocks.com/en/tutorials/es6/promises/#toc-promises-queues)). It also notifies via the return value of the notifyCallback method. The promise cannot be resolved or rejected from the notifyCallback method. The errorCallback and notifyCallback arguments are optional.

* catch(errorCallback) – shorthand for promise.then(null, errorCallback)
* finally(callback, notifyCallback) – allows you to observe either the fulfillment or rejection of a promise, but to do so without modifying the final value. This is useful to release resources or do some clean-up that needs to be done whether the promise was rejected or resolved. See the [full specification](https://github.com/kriskowal/q/wiki/API-Reference#promisefinallycallback) for more information.

**Chaining promises**

Because calling the then method of a promise returns a new derived promise, it is easily possible to create a chain of promises:

promiseB = promiseA.then(function(result) {

return result + 1;

});

// promiseB will be resolved immediately after promiseA is resolved and its value

// will be the result of promiseA incremented by 1

It is possible to create chains of any length and since a promise can be resolved with another promise (which will defer its resolution further), it is possible to pause/defer resolution of the promises at any point in the chain. This makes it possible to implement powerful APIs like $http's response interceptors.

**Differences between Kris Kowal's Q and $q**

There are two main differences:

* $q is integrated with the [$rootScope.Scope](https://docs.angularjs.org/api/ng/type/$rootScope.Scope) Scope model observation mechanism in AngularJS, which means faster propagation of resolution or rejection into your models and avoiding unnecessary browser repaints, which would result in flickering UI.
* Q has many more features than $q, but that comes at a cost of bytes. $q is tiny, but contains all the important functionality needed for common async tasks.

**Testing**

it('should simulate promise', inject(function($q, $rootScope) {

var deferred = $q.defer();

var promise = deferred.promise;

var resolvedValue;

promise.then(function(value) { resolvedValue = value; });

expect(resolvedValue).toBeUndefined();

// Simulate resolving of promise

deferred.resolve(123);

// Note that the 'then' function does not get called synchronously.

// This is because we want the promise API to always be async, whether or not

// it got called synchronously or asynchronously.

expect(resolvedValue).toBeUndefined();

// Propagate promise resolution to 'then' functions using $apply().

$rootScope.$apply();

expect(resolvedValue).toEqual(123);

}));

**Dependencies:** [$rootScope](https://docs.angularjs.org/api/ng/service/$rootScope)

**Usage:** $q(resolver);

Arguments

| Param | | | Type | | Details |
| --- | --- | --- | --- | --- | --- |
| resolver | | [**function(function, function)**](https://docs.angularjs.org/) | | Function which is responsible for resolving or rejecting the newly created promise. The first parameter is a function which resolves the promise, the second parameter is a function which rejects the promise. | |
| Returns: | | The newly created promise. | | | | |

**Methods**

* **defer();** Creates a Deferred object which represents a task which will finish in the future.

**Returns:** Returns a new instance of deferred.

* **reject(reason);**

Creates a promise that is resolved as rejected with the specified reason. This api should be used to forward rejection in a chain of promises. If you are dealing with the last promise in a promise chain, you don't need to worry about it.

When comparing deferreds/promises to the familiar behavior of try/catch/throw, think of reject as the throw keyword in JavaScript. This also means that if you "catch" an error via a promise error callback and you want to forward the error to the promise derived from the current promise, you have to "rethrow" the error by returning a rejection constructed via reject.

promiseB = promiseA.then(function(result) {

// success: do something and resolve promiseB

// with the old or a new result

return result;

}, function(reason) {

// error: handle the error if possible and

// resolve promiseB with newPromiseOrValue,

// otherwise forward the rejection to promiseB

if (canHandle(reason)) {

// handle the error and recover

return newPromiseOrValue;

}

return $q.reject(reason);

});

**Parameters**

| **Param** | | **Type** | | **Details** |
| --- | --- | --- | --- | --- |
| reason | [**\***](https://docs.angularjs.org/) | | Constant, message, exception or an object representing the rejection reason. | |

**Returns:** Returns a promise that was already resolved as rejected with the reason.

* **when(value, [successCallback], [errorCallback], [progressCallback]);**

Wraps an object that might be a value or a (3rd party) then-able promise into a $q promise. This is useful when you are dealing with an object that might or might not be a promise, or if the promise comes from a source that can't be trusted.

**Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| **value** | [\*](https://docs.angularjs.org/) | **Value or a promise** |
| **successCallback**  ***(optional)*** | [Function=](https://docs.angularjs.org/) |  |
| **errorCallback**  ***(optional)*** | [Function=](https://docs.angularjs.org/) |  |
| **progressCallback**  ***(optional)*** | [Function=](https://docs.angularjs.org/) |  |

**Returns**

|  |  |
| --- | --- |
| [Promise](https://docs.angularjs.org/) | Returns a promise of the passed value or promise |

* **resolve(value, [successCallback], [errorCallback], [progressCallback]);**

Alias of [when](https://docs.angularjs.org/api/ng/service/$q#when) to maintain naming consistency with ES6.

**Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| **value** | [\*](https://docs.angularjs.org/) | **Value or a promise** |
| **successCallback**  ***(optional)*** | [Function=](https://docs.angularjs.org/) |  |
| **errorCallback**  ***(optional)*** | [Function=](https://docs.angularjs.org/) |  |
| **progressCallback**  ***(optional)*** | [Function=](https://docs.angularjs.org/) |  |

**Returns**

|  |  |
| --- | --- |
| [Promise](https://docs.angularjs.org/) | **Returns a promise of the passed value or promise** |

* **all(promises);**

Combines multiple promises into a single promise that is resolved when all of the input promises are resolved.

**Parameters**

| **Param** | | **Type** | | **Details** |
| --- | --- | --- | --- | --- |
| **promises** | [Array.<Promise>Object.<Promise>](https://docs.angularjs.org/) | | **An array or hash of promises.** | |

**Returns**

|  |  |
| --- | --- |
| [Promise](https://docs.angularjs.org/) | Returns a single promise that will be resolved with an array/hash of values, each value corresponding to the promise at the same index/key in the promises array/hash. If any of the promises is resolved with a rejection, this resulting promise will be rejected with the same rejection value. |

* **race(promises);**

Returns a promise that resolves or rejects as soon as one of those promises resolves or rejects, with the value or reason from that promise.

**Parameters**

| **Param** | | **Type** | | **Details** |
| --- | --- | --- | --- | --- |
| **promises** | [Array.<Promise>Object.<Promise>](https://docs.angularjs.org/) | | **An array or hash of promises.** | |

**Returns**

|  |  |
| --- | --- |
| [Promise](https://docs.angularjs.org/) | a promise that resolves or rejects as soon as one of the promises resolves or rejects, with the value or reason from that promise. |

## ****What is the role of services in AngularJS and name any services made available by default?****

– AngularJS Services are objects that provide separation of concerns to an AngularJS app.  
– AngularJS Services can be created using a factory method or a service method.  
– Services are singleton components. All components of the application (into which the service is injected) will work with single instance of the service.  
– An AngularJS service allows developing of business logic without depending on the View logic which will work with it.

Few of the inbuilt services in AngularJS are:  
– the $http service: The $http service is a core Angular service that facilitates communication with the remote HTTP servers via the browser’s XMLHttpRequest object or via JSONP  
– the $log service: Simple service for logging. Default implementation safely writes the message into the browser’s console  
– the $anchorScroll: it scrolls to the element related to the specified hash or (if omitted) to the current value of $location.hash()  
**Why should one know about AngularJS Services**, you may ask. Well, understanding the purpose of AngularJS Services helps bring modularity to AngularJS code.  
Services are the best may to evolve reusable API within and AngularJS app

**Overview:**

* AngularJS Services help create reusable components.
* A Service can be created either using the service() method or the factory() method.
* A typical service can be injected into another service or into an AngularJS Controller.

## ****When creating a directive, it can be used in several different ways in the view. Which ways for using a directive do you know? How do you define the way your directive will be used?****

When you create a directive, it can be used as an attribute, element or class name. To define which way to use, you need to set the restrict option in your directive declaration.

The restrict option is typically set to:

‘A’ – only matches attribute name  
‘E’ – only matches element name  
‘C’ – only matches class name

These restrictions can all be combined as needed:

‘AEC’ – matches either attribute or element or class name

## ****When should you use an attribute versus an element?****

Use an element when you are creating a component that is in control of the template. Use an attribute when you are decorating an existing element with new functionality.

This topic is important so developers can understand the several ways a directive can be used inside a view and when to use each way.

**$compile:-** Compiles an HTML string or DOM into a template and produces a template function, which can then be used to link [scope](https://docs.angularjs.org/api/ng/type/$rootScope.Scope) and the template together.

The compilation is a process of walking the DOM tree and matching DOM elements to [directives](https://docs.angularjs.org/api/ng/provider/$compileProvider#directive).

**Note:** This document is an in-depth reference of all directive options. For a gentle introduction to directives with examples of common use cases, see the [directive guide](https://docs.angularjs.org/guide/directive).

**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)](https://docs.angularjs.org/api/ng/service/$compile#directive-definition-object) 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](https://docs.angularjs.org/api/ng/service/$compile#-priority-): 0,

[template](https://docs.angularjs.org/api/ng/service/$compile#-template-): '<div></div>', // or // function(tElement, tAttrs) { ... },

// or

// [templateUrl](https://docs.angularjs.org/api/ng/service/$compile#-templateurl-): 'directive.html', // or // function(tElement, tAttrs) { ... },

[transclude](https://docs.angularjs.org/api/ng/service/$compile#-transclude-): false,

[restrict](https://docs.angularjs.org/api/ng/service/$compile#-restrict-): 'A',

[templateNamespace](https://docs.angularjs.org/api/ng/service/$compile#-templatenamespace-): 'html',

[scope](https://docs.angularjs.org/api/ng/service/$compile#-scope-): false,

[controller](https://docs.angularjs.org/api/ng/service/$compile#-controller-): function($scope, $element, $attrs, $transclude, otherInjectables) { ... },

[controllerAs](https://docs.angularjs.org/api/ng/service/$compile#-controlleras-): 'stringIdentifier',

[bindToController](https://docs.angularjs.org/api/ng/service/$compile#-bindtocontroller-): false,

[require](https://docs.angularjs.org/api/ng/service/$compile#-require-): 'siblingDirectiveName', // or // ['^parentDirectiveName', '?optionalDirectiveName', '?^optionalParent'],

[multiElement](https://docs.angularjs.org/api/ng/service/$compile#-multielement-): false,

[compile](https://docs.angularjs.org/api/ng/service/$compile#-compile-): function compile(tElement, tAttrs, transclude) {

return {

[pre](https://docs.angularjs.org/api/ng/service/$compile#pre-linking-function): function preLink(scope, iElement, iAttrs, controller) { ... },

[post](https://docs.angularjs.org/api/ng/service/$compile#post-linking-function): function postLink(scope, iElement, iAttrs, controller) { ... }

}

// or

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

},

// or

// [link](https://docs.angularjs.org/api/ng/service/$compile#-link-): {

// [pre](https://docs.angularjs.org/api/ng/service/$compile#pre-linking-function): function preLink(scope, iElement, iAttrs, controller) { ... },

// [post](https://docs.angularjs.org/api/ng/service/$compile#post-linking-function): function postLink(scope, iElement, iAttrs, controller) { ... }

// }

// or

// [link](https://docs.angularjs.org/api/ng/service/$compile#-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) { ... }

});

**Life-cycle hooks**

Directive controllers can provide the following methods that are called by AngularJS at points in the life-cycle of the directive:

* $onInit() - Called on each controller after all the controllers on an element have been constructed and had their bindings initialized (and before the pre & post linking functions for the directives on this element). This is a good place to put initialization code for your controller.
* $onChanges(changesObj) - Called whenever one-way (<) or interpolation (@) bindings are updated. The changesObj is a hash whose keys are the names of the bound properties that have changed, and the values are an object of the form { currentValue, previousValue, isFirstChange() }. Use this hook to trigger updates within a component such as cloning the bound value to prevent accidental mutation of the outer value. Note that this will also be called when your bindings are initialized.
* $doCheck() - Called on each turn of the digest cycle. Provides an opportunity to detect and act on changes. Any actions that you wish to take in response to the changes that you detect must be invoked from this hook; implementing this has no effect on when $onChanges is called. For example, this hook could be useful if you wish to perform a deep equality check, or to check a Date object, changes to which would not be detected by AngularJS's change detector and thus not trigger $onChanges. This hook is invoked with no arguments; if detecting changes, you must store the previous value(s) for comparison to the current values.
* $onDestroy() - Called on a controller when its containing scope is destroyed. Use this hook for releasing external resources, watches and event handlers. Note that components have their $onDestroy() hooks called in the same order as the $scope.$broadcast events are triggered, which is top down. This means that parent components will have their $onDestroy()hook called before child components.
* $postLink() - Called after this controller's element and its children have been linked. Similar to the post-link function this hook can be used to set up DOM event handlers and do direct DOM manipulation. Note that child elements that contain templateUrldirectives 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.

**Comparison with life-cycle hooks in the new Angular**

The new Angular also uses life-cycle hooks for its components. While the AngularJS life-cycle hooks are similar there are some differences that you should be aware of, especially when it comes to moving your code from AngularJS to Angular:

* AngularJS hooks are prefixed with $, such as $onInit. Angular hooks are prefixed with ng, such as ngOnInit.
* AngularJS hooks can be defined on the controller prototype or added to the controller inside its constructor. In Angular you can only define hooks on the prototype of the Component class.
* Due to the differences in change-detection, you may get many more calls to $doCheck in AngularJS than you would to ngDoCheckin Angular.
* Changes to the model inside $doCheck will trigger new turns of the digest loop, which will cause the changes to be propagated throughout the application. Angular does not allow the ngDoCheck hook to trigger a change outside of the component. It will either throw an error or do nothing depending upon the state of enableProdMode().

**Life-cycle hook examples**

This example shows how you can check for mutations to a Date object even though the identity of the object has not changed.

angular.module('do-check-module', [])

.component('app', {

template:

'Month: <input ng-model="$ctrl.month" ng-change="$ctrl.updateDate()">' +

'Date: {{ $ctrl.date }}' +

'<test date="$ctrl.date"></test>',

controller: function() {

this.date = new Date();

this.month = this.date.getMonth();

this.updateDate = function() {

this.date.setMonth(this.month);

};

}

})

.component('test', {

bindings: { date: '<' },

template:

'<pre>{{ $ctrl.log | json }}</pre>',

controller: function() {

var previousValue;

this.log = [];

this.$doCheck = function() {

var currentValue = this.date && this.date.valueOf();

if (previousValue !== currentValue) {

this.log.push('doCheck: date mutated: ' + this.date);

previousValue = currentValue;

}

};

}

});

In Index.cshtml

<app></app>

Month: Date: "2017-02-15T13:37:04.709Z"

[

"doCheck: date mutated: Wed Feb 15 2017 19:07:04 GMT+0530 (India Standard Time)"

]

This example show how you might use $doCheck to trigger changes in your component's inputs even if the actual identity of the component doesn't change. (Be aware that cloning and deep equality checks on large arrays or objects can have a negative impact on your application performance)

angular.module('do-check-module', [])

.component('test', {

bindings: { items: '<' },

template:

'<pre>{{ $ctrl.log | json }}</pre>',

controller: function() {

this.log = [];

this.$doCheck = function() {

if (this.items\_ref !== this.items) {

this.log.push('doCheck: items changed');

this.items\_ref = this.items;

}

if (!angular.equals(this.items\_clone, this.items)) {

this.log.push('doCheck: items mutated');

this.items\_clone = angular.copy(this.items);

}

};

}

});

In index.cshtml

<div ng-init="items = []">

<button ng-click="items.push(items.length)">Add Item</button>

<button ng-click="items = []">Reset Items</button>

<pre>{{ items }}</pre>

<test items="items"></test>

</div>

Add Item Reset Items

[]

[

"doCheck: items changed",

"doCheck: items mutated"

]

**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 (default is false), the HTML compiler will collect DOM nodes between nodes with the attributes directive-name-start and directive-name-end, and group them together as the directive elements. It is recommended that this feature be used on directives which are not strictly behavioral (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 false, true, or an object:

* **false (default):** 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.
* **{...} (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 <my-component my-attr="hello {{name}}"> and the isolate scope definition scope: { localName:'@myAttr' }, the directive's scope property localName will reflect the interpolated value of hello {{name}}. As the name attribute changes so will the localName property on the directive's scope. The name is read from the parent scope (not the directive's scope).
* = or =attr - set up a bidirectional binding between a local scope property and an expression passed via the attribute attr. The expression is evaluated 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 <my-component my-attr="parentModel"> and the isolate scope definition scope: { localModel: '=myAttr' }, the property localModel on the directive's scope will reflect the value of parentModel on the parent scope. Changes to parentModel will be reflected in localModel and vice versa. Optional attributes should be marked as such with a question mark: =? or =?attr. If the binding expression is non-assignable, or if the attribute isn't optional and doesn't exist, an exception ([$compile:nonassign](https://docs.angularjs.org/error/$compile/nonassign)) will be thrown upon discovering changes to the local value, since it will be impossible to sync them back to the parent scope. By default, the [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) method is used for tracking changes, and the equality check is based on object identity. However, if an object literal or an array literal is passed as the binding expression, the equality check is done by value (using the [angular.equals](https://docs.angularjs.org/api/ng/function/angular.equals) function). It's also possible to watch the evaluated value shallowly with [$watchCollection](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watchCollection): use =\* or =\*attr (=\*? or =\*?attr if the attribute is optional).
* < or <attr - set up a one-way (one-directional) binding between a local scope property and an expression passed via the attribute attr. The expression is evaluated 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. You can also make the binding optional by adding ?: <? or <?attr.

For example, given <my-component my-attr="parentModel"> and directive definition of scope: { localModel:'<myAttr' }, then the isolated scope property localModel will reflect the value of parentModel on the parent scope. Any changes to parentModelwill be reflected in localModel, but changes in localModel will not reflect in parentModel. There are however two caveats:

one-way binding does not copy the value from the parent to the isolate scope, it simply sets the same value. That means if your bound value is an object, changes to its properties in the isolated scope will be reflected in the parent scope (because both reference the same object).

one-way binding watches changes to the **identity** of the parent value. That means the [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) on the parent value only fires if the reference to the value has changed. In most cases, this should not be of concern, but can be important to know if you one-way bind to an object, and then replace that object in the isolated scope. If you now change a property of the object in your parent scope, the change will not be propagated to the isolated scope, because the identity of the object on the parent scope has not changed. Instead you must assign a new object.

One-way binding is useful if you do not plan to propagate changes to your isolated scope bindings back to the parent. However, it does not make this completely impossible.

* & 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 <my-component my-attr="count = count + value"> and the isolate scope definition scope: { localFn:'&myAttr' }, the isolate scope property localFn will point to a function wrapper for the count = 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**

This property is used to bind scope properties directly to the controller. It can be either true or an object hash with the same format as the scope property.

When an isolate scope is used for a directive (see above), bindToController: true will allow a component to have its properties bound to the controller, rather than to scope.

After the controller is instantiated, the initial values of the isolate scope bindings will be bound to the controller properties. You can access these bindings once they have been initialized by providing a controller method called $onInit, which is called after all the controllers on an element have been constructed and had their bindings initialized.

**Deprecation warning:** although bindings for non-ES6 class controllers are currently bound to this before the controller constructor is called, this use is now deprecated. Please place initialization code that relies upon bindings inside a $onInit method on the controller, instead.

It is also possible to set bindToController to an object hash with the same format as the scope property. This will set up the scope bindings to the controller directly. Note that scope can still be used to define which kind of scope is created. By default, no scope is created. Use scope: {} to create an isolate scope (useful for component directives).

If both bindToController and scope are defined and have object hashes, bindToController overrides scope.

**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, slotName):
  + scope: (optional) override the scope.
  + cloneLinkingFn: (optional) argument to create clones of the original transcluded content.
  + futureParentElement (optional):
    - 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 the cloneLinkingFn 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.
  + slotName: (optional) the name of the slot to transclude. If falsy (e.g. null, undefined or '') then the default transclusion is provided. The $transclude function also has a method on it, $transclude.isSlotFilled(slotName), which returns trueif the specified slot contains content (i.e. one or more DOM nodes).

**require**

Require another directive and inject its controller as the fourth argument to the linking function. The require property can be a string, an array or an object:

* a **string** containing the name of the directive to pass to the linking function
* an **array** containing the names of directives to pass to the linking function. The argument passed to the linking function will be an array of controllers in the same order as the names in the require property
* an **object** whose property values are the names of the directives to pass to the linking function. The argument passed to the linking function will also be an object with matching keys, whose values will hold the corresponding controllers.

If the require property is an object and bindToController is truthy, then the required controllers are bound to the controller using the keys of the require property. This binding occurs after all the controllers have been constructed but before $onInit is called. If the name of the required controller is the same as the local name (the key), the name can be omitted. For example, {parentDir: '^^'} is equivalent to {parentDir: '^^parentDir'}. See the [$compileProvider](https://docs.angularjs.org/api/ng/provider/$compileProvider#component) helper for an example of how this can be used. If no such required directive(s) can be found, or if the directive does not have a controller, then an error is raised (unless no link function is specified and the required controllers are not being bound to the directive controller, 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#transclusion) section below.

**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 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 template ortemplateUrl 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 $transcludeparameter of directive controllers, see [the controller section for details](https://docs.angularjs.org/api/ng/service/$compile#-controller-).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).

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

* 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.
* {...} **(an object hash):** - map elements of the content onto transclusion "slots" in the template.

**Mult-slot transclusion** is declared by providing an object for the transclude property.

This object is a map where the keys are the name of the slot to fill and the value is an element selector used to match the HTML to the slot. The element selector should be in normalized form (e.g. myElement) and will match the standard element variants (e.g. my-element, my:element, data-my-element, etc).

For further information check out the guide on [Matching Directives](https://docs.angularjs.org/guide/directive#matching-directives)

If the element selector is prefixed with a ? then that slot is optional.

For example, the transclude object { slotA: '?myCustomElement' } maps <my-custom-element> elements to the slotA slot, which can be accessed via the $transclude function or via the [ngTransclude](https://docs.angularjs.org/api/ng/directive/ngTransclude) directive.

Slots that are not marked as optional (?) will trigger a compile time error if there are no matching elements in the transclude content. If you wish to know if an optional slot was filled with content, then you can call $transclude.isSlotFilled(slotName) on the transclude function passed to the directive's link function and injectable into the directive's controller.

**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, which the clone will be linked to.

**Best Practice**: Always provide a cloneFn (clone attach function) when you call a transclude 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 transcluded 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) {

// get the attribute value

console.log(attrs.ngModel);

// change the attribute

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

// observe changes to interpolated attribute

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

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

});

}

## ****How do you reset a**** $timeout****,**** $interval()****, and disable a**** $watch()****?****

To reset a timeout and/or $interval, assign the result of the function to a variable and then call the .cancel() function.

var customTimeout = $timeout(function () {

// arbitrary code

}, 55);

$timeout.cancel(customTimeout);

to disable $watch(), we call its deregistration function. $watch() then returns a deregistration function that we store to a variable and that will be called for cleanup

var deregisterWatchFn = $scope.$on(‘$destroy’, function () {

// we invoke that deregistration function, to disable the watch

deregisterWatchFn();

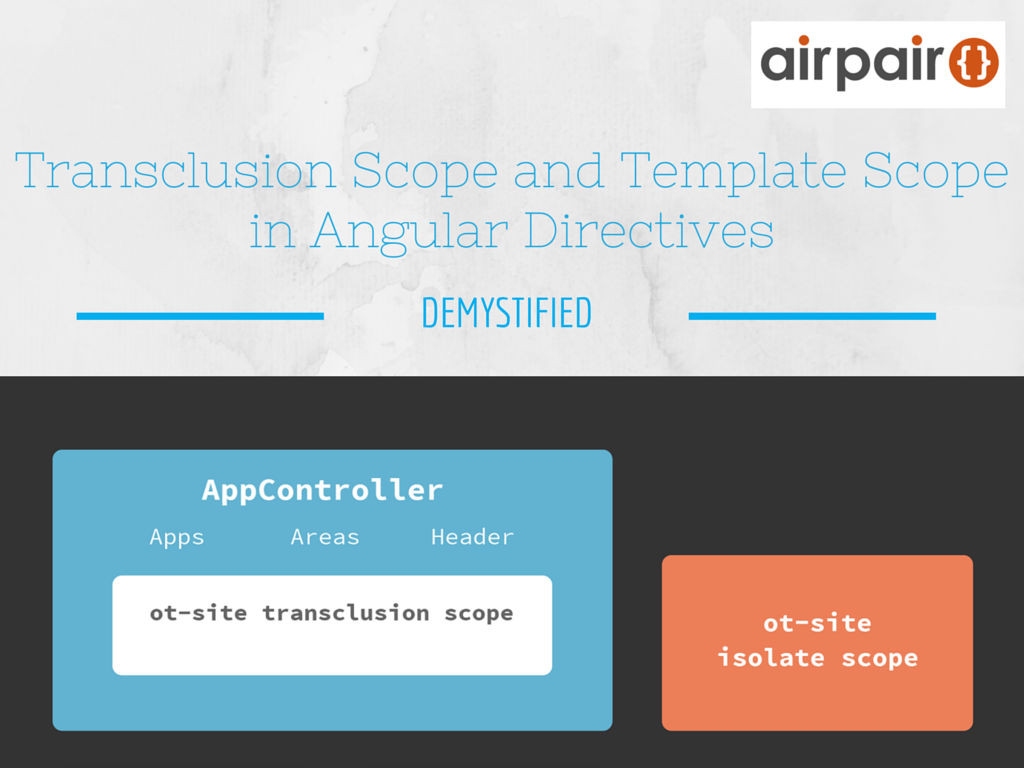
});

## ****Explain what is a**** $scope ****in AngularJS****

Scope is an object that refers to the application model. It is an execution context for expressions. Scopes are arranged in hierarchical structure which mimic the DOM structure of the application. Scopes can watch expressions and propagate events. Scopes are objects that refer to the model. They act as glue between controller and view.

This question is important as it will judge a persons knowledge about a $scope object, and it is one of the most important concepts in AngularJS. Scope acts like a bridge between view and model.

**Transclusion and Template Scope in Angular Directives Demystified**



Synopsis

Understanding scope is a vital part of writing robust Angular directives. It's one of those concepts that seems simple at first, but turns out to have some nuances that can make or break your app -- especially when transclusion comes into the mix or you begin to build directives on top of one another.

This post will delve into some of these scope-related nuances by explaining the scope hierarchy of an example application. It assumes some prior knowledge of Angular directives and the Directive Definition Object (DDO), so if you're brand new at directives, I'd suggest taking a look at the documentation first[[1](https://www.airpair.com/#r1)].

Let's start with some basics.

## What is directive scope?

Typically, when developers refer to a directive's scope, they mean the scope bound to that directive's template during the linking phase. This scope -- configurable through the scope property in the DDO -- is the execution context that Angular uses to look up any expressions defined in the template (such as {{ }} bindings).

There are three types of scope that you can configure for your directive's template:

1. shared scope (scope: false)
2. child scope (scope: true)
3. isolate scope (scope: {})

For most directives, the template's scope is all you need to consider. However, it's important to note that directives that transclude[[2](https://www.airpair.com/#r2)] have a second scope in addition to the template scope: a transclusion scope that follows different rules. But more on that in a moment.

Given that brief introduction, let's dive into an example to dig into these concepts.

## Scope example app

Let's say we are building an application with the help of two components:

1. a site-layout component called ot-site
2. a list component called ot-list



ot-site provides the UI scaffold for our application: the static header, logo, and footer that will appear on every page. However, like any layout component, it has to allow the directive user to pass in some arbitrary, dynamic content for the body section, because that will inevitably change from page-to-page. Given these two requirements, ot-site is a classic use case for transclusion. Through ot-site, we can review how transclusion scope works\*.

ot-list is a simple list component that takes a set of data and turns it into a styled ul element with selection logic. We can use ot-list to demonstrate the three different options for passing data through into the template scope and how they affect the directive.

\*This layout component and transclusion in general is discussed in more detail in a [*previous blog post*](https://www.airpair.com/angularjs/posts/creating-container-components-part-2-angular-1-directives). Here, the implementation of *ot-site* is simplified, as our focus is on scope.

## Transclusion scope

We'll start withot-site. Before jumping into its scope hierarchy, let's take a moment to review how it is structured.

### Brief intro to transclusion

As mentioned before, we'd like to pass arbitrary content into the body of our scaffold, and the best way to do this is through transclusion. If we make ot-site a transcluding directive, any HTML passed between its opening and closing tags will be transcluded into the template. So if we want ot-list to appear in the body of the layout scaffold, our application markup might look like this:

**index.html**

<ot-site>

<ot-list></ot-list>

</ot-site>

To hook up the transclusion on the JS side, our directive definition just needs to:

* add a template with its default scaffold (header, logo, etc)
* set transclude to true (to let Angular know to save the content between the directive tags)
* and add the ng-transclude directive to the element where the content should go.

**otSite.js**

angular.module("ot-components")

.directive("otSite", function() {

return {

transclude: true,

scope: {},

template: [

'<div class="ot-site">',

'<div class="ot-site--head">',

'<span class="ot-site--logo"></span>',

'<h1 ng-bind="title"></h1>',

'</div>',

'<div class="ot-site--menu"></div>',

'<div class="ot-site--body" ng-transclude></div>',

'<div class="ot-site--foot">',

'&copy; 2015 OpenTable, Inc.',

'</div>',

'</div>'

].join('')

};

});

So given that implementation, where does transclusion scope come in?

### Template scope vs. transclusion scope

As a transcluding directive, ot-site essentially has two templates:

1. its own internal template (header/logo/footer defined in otSite.js)
2. the custom HTML passed in by the directive user (ot-list tags in index.html)

Because the templates are eventually combined in the DOM when the custom HTML is appended to the directive template, it's intuitive to assume that the two pieces will share the same scope once they are brought together. But in fact, they are completely separate scopes that follow different rules.

1. The internal template's scope is always controlled by whatever you set in the scope property of the DDO, as described earlier.
2. However, the scope of the custom HTML (a.k.a. the transclusion scope) is unaffected by how you've configured the scope property. **It will always be a child scope of whatever outer context the directive was placed in.** \*\*

\*\*If you don't use the built-in transclusion functionality and transclude manually (by using the low-level transclude function), you can technically pass in whichever scope you'd like to be linked to the custom template. However, this is NOT recommended because it typically breaks bindings.

### Wait, what's a child scope?

Let's take a step back for a moment.

Scopes in Angular are organized into a hierarchy. When you bootstrap your application with ng-app, exactly one rootScope is created to form the top of that hierarchy. As the root of the scope tree, it's the scope from which any other scopes created in your application descend.

rootScope

child child

child child

So when a new scope is created (for example, by a directive like ng-controller or maybe one of your own), it becomes a child scope of the root scope or one of its descendants.

Child scopes in the hierarchy inherit prototypically from their parent scopes, all the way up to the rootScope. When a lookup fails for a property on the child scope, next it will check the parent's scope for that property, and so on up the chain.

This hierarchy broadly mimics the DOM structure of the app. Which scope is bound to a particular HTML tag does depend on where the tag falls in the DOM (with some notable exceptions in directives with isolate scope). So if a tag is within a div that contains an ng-controller (which creates its own child scope), that tag will be within that controller scope's sphere of influence.

### The current scope hierarchy

Let's zoom out and take a look at where the ot-site tag has been placed in index.html, so we can start to understand the scope hierarchy.

**index.html**

<html ng-app="ot-components">

<head>...</head>

<body ng-controller="AppController">

<ot-site>

<ot-list></ot-list>

</ot-site>

</body>

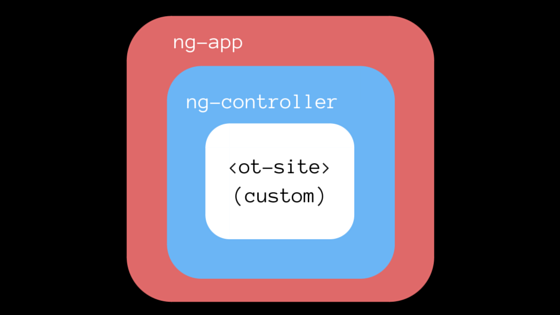
</html>

From the broader index.html, we can see that there are two higher-level scopes defined in this application so far:

1. **the rootScope** - which stems from whichever element ng-app is on (<html>).
2. **the AppController scope** - which stems from whichever element has the matching ng-controller tag (<body>). This is a child scope of the $rootScope.

So where does the scope of ot-site's custom template -- the transclusion scope -- fall in this hierarchy? As mentioned, the transclusion scope is always a child scope of its outer context. Because <ot-site> has been placed within the <body> tags, the AppController scope is its outer context. Thus, the transclusion scope for ot-site will be a child of the AppController scope.

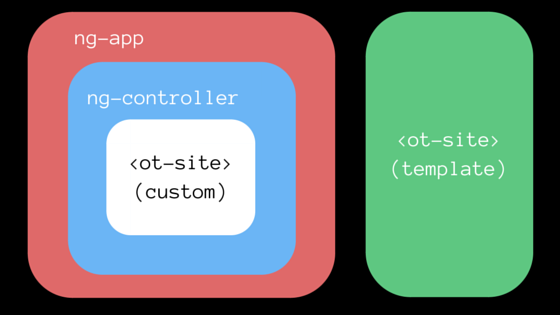
**scope hierarchy**



As a child of AppController, the custom template is perfectly set up to inherit any bindings it needs from the broader application. This makes sense for transclusion, because if you had to pass in each model to the directive explicitly, it wouldn't truly support arbitrary content. The directive itself would have to anticipate every potential toggle or piece of data, which has its limits.

So where does the actual ot-site template (the divs that represent the header and logo, etc) fall in that tree?

You may have noticed that we set an isolate scope for ot-siteearlier (scope: {}), so unlike the transclusion scope, the scope for the template does **not** inherit prototypically from anything:



As such, it's removed from the prototype chain. While the custom template can reach up to access bindings from the controller, the isolated template is protected from any leaking to or from the application (more on this later).

Now that we understand ot-site, let's take a look at ot-list.

## Template scope

The template for ot-list is fairly straightforward. Basically, we're just iterating over a list of items with an ng-repeat and setting up a selection callback:

**ot-list.html**

<ul>

<li ng-repeat="item in items" ng-bind="item"

ng-class="{'ot-selected': item === selected}"

ng-click="selectItem(item)">

</li>

</ul>

From the template, you can see that we really need two pieces of information to generate our ul:

1. The data set (items)
2. The initial selection for the item (selected)

Let's assume those properties are coming from our controller scope, through the areas object:

**app.js**

angular.module("my-app")

.controller("AppController", ($scope) => {

$scope.areas = {

list: [

"Floorplan",

"Combinations",

"Schedule",

"Publish"

],

current: "Floorplan"

};

});

We can pass the properties into the directive using HTML attributes, so our application markup might look like this:

**index.html**

<ot-site>

<ot-list items="{{ areas }}" selected="{{ areas.current }}"></ot-list>

</ot-site>

So our ot-list implementation will have to "catch" these properties from the controller and put them on our template scope. Remember that there are three ways to accomplish this: by setting a shared scope, a child scope, or an isolate scope.

### Shared scope

If we don't set the scope property, we can pass the data through by taking advantage of the attrs argument of the link function. We can transfer each item from attrs to the scope one-by-one:

**otList.js**

angular.module("ot-components")

.directive("otList", function() {

return {

templateUrl: "ot-list.html"

link: function (scope, elem, attrs) {

scope.items = JSON.parse(attrs.items);

scope.selected = attrs.selected;

scope.selectItem = function(item) {

scope.selected = item;

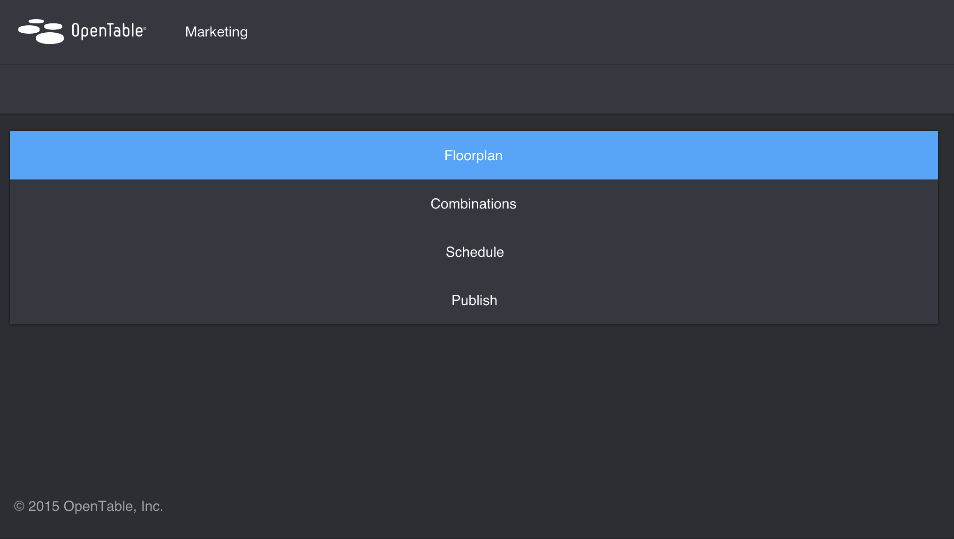
};

}

};

})

If we test that code, it will actually appear to work fine:



[Play with the code demo here](http://codepen.io/kara/pen/jPQeXw)

However, when you don't set the scope property at all, as we've done, the value is false by default. This means that your directive creates **no** new scope of its own. It shares the scope of whatever its outside context happens to be. This means the scope of the directive is completely vulnerable to its outside environment - and vice versa.

To drive that point home, let's see what happens when we add a second list to the application, drawing from a second data source in the controller, apps.

**app.js**

angular.module("ot-components")

.controller("AppController", ($scope) => {

$scope.areas = {...};

$scope.apps = {

list: [

"Marketing",

"Planning",

"Reservations",

"Settings"

],

current: "Marketing"

};

**index.html**

<ot-site>

<ot-list

items="{{ areas.list }}"

selected="{{ areas.current }}">

</ot-list>

<ot-list

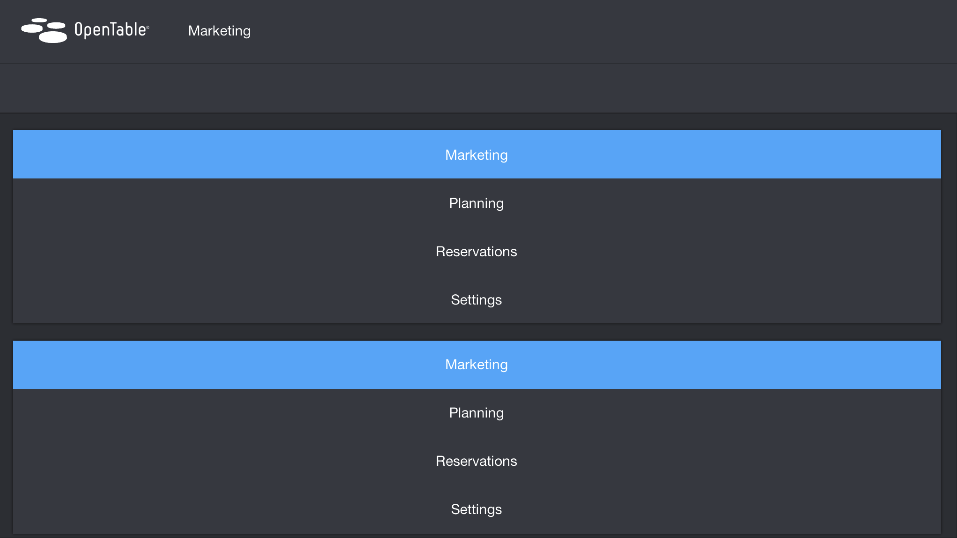
items="{{ apps.list }}"

selected="{{ apps.current }}">

</ot-list>

</ot-site>

If we look at the output for the two lists...

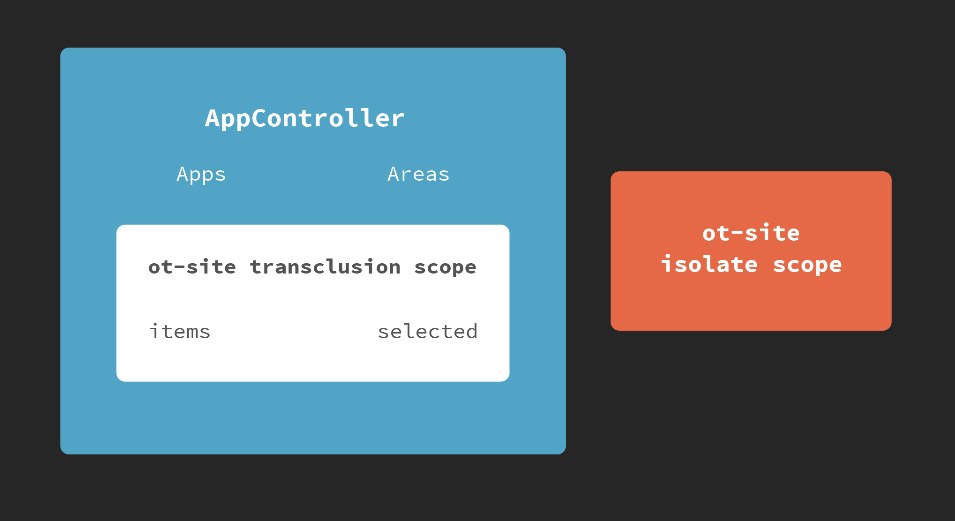


...something is obviously off. Check out [the code demo](http://codepen.io/kara/pen/MwzPZr) and click around.

We set up two different lists of data on the controller, one for each list instance, so we’d expect each list to show its own set of items. But both of the lists are displaying the same data.

And if we click on either of lists to select something, both of the lists show the new selection. They're glued together. We would want each list to be selectable independently of other lists... so what’s going on?

As foreshadowed, shared scope is the culprit here. As the list directives aren’t defining their own scopes, you’ll remember that both of their templates are bound to whatever outer scope they were placed in. Since we have transcluded the lists into the site scaffold, they are sharing the ot-site transclusion scope.



Design credit: Simon Attley

Since we can only have one items property and one selected property per scope, this means that the two list instances are sharing these properties. The first instance sets an items property and a selected property, then the second instance immediately overwrites them. That’s why the lists are the same, and the selections are coordinated. We need to have a setup where the instances aren’t sharing variables and overwriting each other.

This setup also has another problem - even with one instance of ot-list. What would happen if we dropped ot-list in an outer context that already had an items or selected variable. In that case, the second instance of ot-list wouldn’t just overwrite the variables of the first instance. **It would also break whatever was using those variables in the broader application**. You’re giving the directive the ability to pollute its outer environment and potentially create odd problems down the line. Shared scope can be pretty risky.

### Child scope

We can improve the situation by simply setting the scope property to true.

**otList.js**

angular.module("ot-components")

.directive("otList", function() {

return {

scope: true,

templateUrl: "ot-list.html"

link: function (scope, elem, attrs) {

scope.items = JSON.parse(attrs.items);

scope.selected = attrs.selected;

scope.selectItem = function(item) {

scope.selected = item;

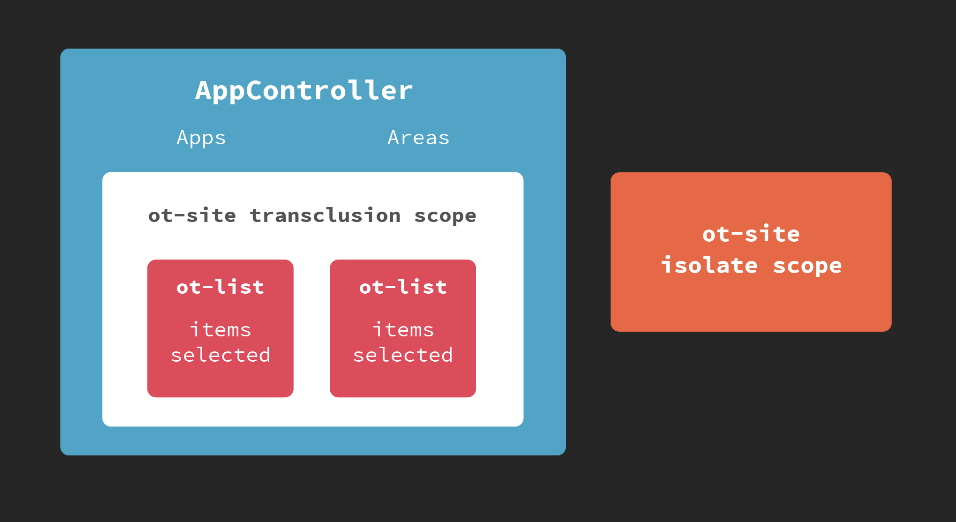
};

}

};

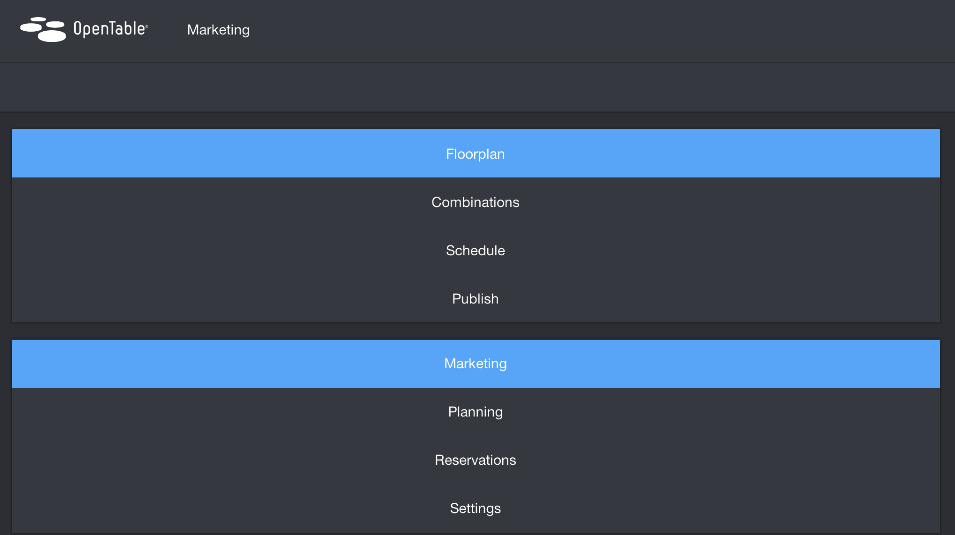
})

When scope is true, each instance of the directive will create its own child scope in the outer scope. So each instance of ot-list here has its own copy of items and selected. As sibling scopes, they won’t affect or overwrite each other’s variables.



Design credit: Simon Attley

If we look at the output now that scope is true, we can see that our problem has been fixed. Each list has its own set of data, and the selections move independently of one another.



[See code demo here](http://codepen.io/kara/pen/VLVEOZ)

This is undoubtedly an improvement, but it too has its drawbacks. Having created a child scope, the list is still part of the prototype chain. While we’ve fixed any leaks from the list into its outer environment, what about leaks from its outer environment into the list?

I'll give an example. Let’s say we wanted to add an optional header section to our list that would describe what the list contained. If you added a header attribute to the directive and passed in header text, the list would display a header. If there was no header property, the header section of the list wouldn't appear at all. With a child scope, this setup would fail if a header property happened to exist anywhere above the directive in the prototype chain.

Why? Let's say there was a header property on the controller scope for a different purpose, and we set up our ot-list directive without passing in a header. We'd expect that no header section would appear on our list because we didn't pass one. However, the header property from the controller would leak down to the directive scope through inheritance. The header property correctly wouldn't be found on the directive scope, but once that lookup failed, JavaScript would check the scope it inherits from - the controller - and would find and use that header variable. Thus, the directive would always show the text from the controller.

Any time you use a child scope, the child scope will always be vulnerable to pollution from up the prototype chain. So if the directive user happens to forget to add an attribute - or, like in this case, deliberately omits one - it might inherit an unrelated one from its environment.

### Isolate scope

For reusable components, we want complete assurance that there won’t be any leaks in either direction - it shouldn’t be able to affect its environment and its environment shouldn’t be able to affect it. That way, we can be sure it will work in any context. So what we need here is a scope that is outside this prototype chain, that won’t inherit anything directly from its environment - in other words, an isolate scope.

We can create an isolate scope as soon as we pass an object in to the scope property. It can simply be an empty object, as Angular is just checking its type.

**otList.js**

angular.module("ot-components")

.directive("otList", function() {

return {

scope: {},

templateUrl: "ot-list.html"

link: function (scope, elem, attrs) {

scope.items = JSON.parse(attrs.items);

scope.selected = attrs.selected;

scope.selectItem = function(item) {

scope.selected = item;

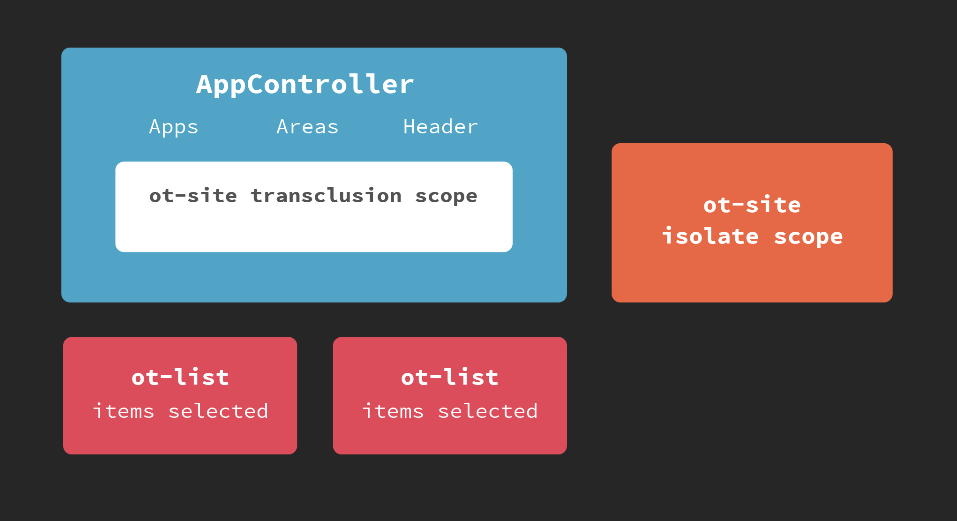
};

}

};

})

What does this do to our scope hierarchy? It takes each ot-list instance out of the prototype chain and completely isolates it.



Design credit: Simon Attley

It can’t inherit anything directly. If we want our directives to have access to any variable, we will have to pass it in explicitly through the scope object. This creates a type of "whitelist", and has the added benefit of allowing us to remove this laborious movement of attributes one by one to the scope.

**otList.js**

angular.module("ot-components")

.directive("otList", function() {

return {

scope: {

items: "=items",

selected: "=selected"

},

templateUrl: "ot-list.html"

link: function (scope, elem, attrs) {

scope.selectItem = function(item) {

scope.selected = item;

};

}

};

})

If you use the scope object, you can use its shorthand instead. On the left, you add the variables you want on the scope, and on the right, you place the attribute names that correspond to those variables.

If the names will be the same, you shorten it further by omitting the names and keeping the binding strategy:

**otList.js**

angular.module("ot-components")

.directive("otList", function() {

return {

scope: {

items: "=",

selected: "="

},

templateUrl: "ot-list.html"

link: function (scope, elem, attrs) {

scope.selectItem = function(item) {

scope.selected = item;

};

}

};

})

Another advantage of this syntax is that it simplifies setting up two way binding. Instead of manually setting up a scope.$watch, you can use the = binding strategy to accomplish the same thing.

This also allows us to remove the curly braces from our markup and pass our variables in directly for two-way binding:

**index.html**

<ot-site>

<ot-list

items="areas.list"

selected="areas.current">

</ot-list>

<ot-list

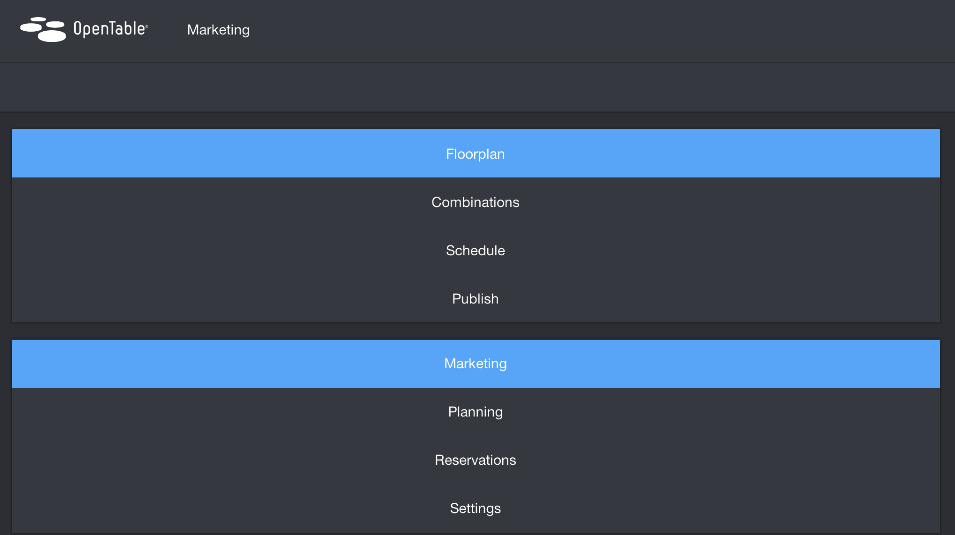
items="apps.list"

selected="apps.current">

</ot-list>

</ot-site>

If we run the code after all our improvements, the result will still work as expected:



[See code demo here](http://codepen.io/kara/pen/bdQQbW)

Isolate scope is pretty great in that it protects your directive from any outside influence. However, it’s worth noting that there are some specific cases where isolate scope might not be the right choice. For instance, if you are creating an attribute directive designed to work with other directives on the same element, an isolate scope doesn't really make sense. Only one isolate scope is allowed per element, so Angular would throw an error.

## Angular 2 directive scope

### Scope differences in Angular 2

With Angular 2 on the horizon, it's important to ensure that directives we write now are easily migratable. To make our ot-list directive definition more portable, it would be wise to reduce our reliance on the scope and move that logic to the controller.

This is because in Angular 2, views will be automatically bound to the component class directly, which allows you to maintain any necessary state or functionality on the class itself. As such, scope is superfluous as a concept and won't be a part of writing directives.

### Migrating directives to Angular 2

So how can we reduce our reliance on scope?

1. First, we can move our models from the scope to the controller by setting the bindToController property to true. This shifts our two-way bindings of items and selected from the scope to the controller itself (so from $scope.items to ctrl.items, etc). Now we are saving all state on our controller.
2. Next, we can move our selectItem function to the controller by moving it into the controller function of the DDO and setting it to this.
3. We can use the controllerAs property to give our template a reference to the controller (here we've set that reference as ctrl).
4. Lastly, in our template, we just have to update our references to ctrl.items, ctrl.selected, and ctrl.selectItem.

**otList.js**

angular.module("ot-components")

.directive("otList", function() {

return {

scope: {

items: "=",

selected: "="

},

bindToController: true,

controllerAs: "ctrl",

templateUrl: "ot-list.html"

controller: function() {

this.selectItem = function(item) {

this.selected = item;

};

}

};

})

**ot-list.html**

<ul>

<li ng-repeat="item in ctrl.items" ng-bind="item"

ng-class="{'ot-selected': item === ctrl.selected}"

ng-click="ctrl.selectItem(item)">

</li>

</ul>

[See the code demo here](http://codepen.io/kara/pen/rVRMrX)

We can actually take this a step further. Currently, we are setting up the controller as an anonymous function. To get as close as we can to Angular 2 component class syntax, we should pull it out into its own, named function, ListController.

**otList.js**

angular.module("ot-components")

.directive("otList", function() {

return {

scope: {

items: "=",

selected: "="

},

bindToController: true,

controllerAs: "ctrl",

templateUrl: "ot-list.html"

controller: ListController

};

})

function ListController(){

this.selectItem = function(item) {

this.selected = item;

}

}

[See updated code demo](http://codepen.io/kara/pen/EjMXGO)

This way, when migrating to Angular 2, you already have your component class set up and ready to go.

## Debugging tricks

### scope.$parent gotcha

One thing to watch out for when debugging scope problems is $parent property on the scope object. At first glance, you may assume that this property points to the parent that the scope inherits from - and this is sometimes true.

However, that's not a guarantee. Though a transclusion scope inherits from the directive's outer context, its parent actually points to the directive scope. It does **not** inherit from that scope. The reference is set up this way to ensure that the transclusion scope is properly destroyed when the directive scope is destroyed.

## Summary

We've explored the various types of scope that exist in directives and their strengths and weaknesses.

Shared scope is risky for any directive with bindings, as it has the potential to overwrite properties and even break its outer environment.

Child scope can be a happy medium between shared scope and isolate scope - inheriting from its parent, but not able to influence anything outside of itself. That said, there is potential for properties in its outer context to leak inside and disrupt its functionality.

The only way to ensure that a directive's functionality is protected is to set up an isolate scope. Given that data must be explicitly passed into the directive through the scope object, it is the safest choice. However, it's not possible to use in all situations, given that only one isolate scope can be created per HTML element.

Lastly, we can't forget that transcluding directives have a second scope, a transclusion scope that will always be able to access models from the broader application.

I hope this overview was helpful. For more information on Angular scopes, you may also want to check out [[3](https://www.airpair.com/#r3)].

Happy scoping!

References:

https://www.airpair.com/angularjs/posts/transclusion-template-scope-in-angular-directives

## ****What are Directives?****

Directives are 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), or even to transform the DOM element and its children. Angular comes with a set of these directives built-in, like ngBind, ngModel, and ngClass. Much like you create controllers and services, you can create your own directives for Angular to use. When Angular bootstraps your application, the HTML compiler traverses the DOM matching directives against the DOM elements.

This question is important because directives define the UI while defining a single page app. You need to be very clear about how to create a new custom directive or use the existing ones already pre-build in AngularJS.

## ****What is DDO Directive Definition Object?****

“DDO is an object used while creating a custome directive. A standard DDO object has following parameters.

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( ... ) { ... }

};"

This question mainly judges whether candidate knows about creating custom directives.

## ****What is a singleton pattern and where we can find it in Angularjs?****

Is a great pattern that restricts the use of a class more than once. We can find singleton pattern in angular in dependency injection and in the services.

In a sense, if you do 2 times ‘new Object()‘ without this pattern, you will be alocating 2 pieces of memory for the same object. With singleton pattern, if the object exists, you reuse it.

## ****What is an interceptor? What are common uses of it?****

An interceptor is a middleware code where all the $http requests go through.

The interceptor is a factory that are registered in $httpProvider. You have 2 types of requests that go through the interceptor, request and response (with requestError and responseErrorrespectively). This piece of code is very useful for error handling, authentication or middleware in all the requests/responses.

**$http:** The $http service is a core AngularJS service that facilitates communication with the remote HTTP servers via the browser's [XMLHttpRequest](https://developer.mozilla.org/en/xmlhttprequest) object or via [JSONP](http://en.wikipedia.org/wiki/JSONP).

* For unit testing applications that use $http service, see [$httpBackend mock](https://docs.angularjs.org/api/ngMock/service/$httpBackend).
* For a higher level of abstraction, please check out the [$resource](https://docs.angularjs.org/api/ngResource/service/$resource) service.
* The $http API is based on the [deferred/promise APIs](https://docs.angularjs.org/api/ng/service/$q) exposed by the $q service. While for simple usage patterns this doesn't matter much, for advanced usage it is important to familiarize yourself with these APIs and the guarantees they provide.

**General usage**

The $http service is a function which takes a single argument — a [configuration object](https://docs.angularjs.org/api/ng/service/$http#usage) — that is used to generate an HTTP request and returns a [promise](https://docs.angularjs.org/api/ng/service/$q).

// Simple GET request example:

$http({

method: 'GET',

url: '/someUrl'

}).then(function successCallback(response) {

// this callback will be called asynchronously

// when the response is available

}, function errorCallback(response) {

// called asynchronously if an error occurs

// or server returns response with an error status.

});

The response object has these properties:

* **data** – {string|Object} – The response body transformed with the transform functions.
* **status** – {number} – HTTP status code of the response.
* **headers** – {function([headerName])} – Header getter function.
* **config** – {Object} – The configuration object that was used to generate the request.
* **statusText** – {string} – HTTP status text of the response.

A response status code between 200 and 299 is considered a success status and will result in the success callback being called. Any response status code outside of that range is considered an error status and will result in the error callback being called. Also, status codes less than -1 are normalized to zero. -1 usually means the request was aborted, e.g. using a config.timeout. Note that if the response is a redirect, XMLHttpRequest will transparently follow it, meaning that the outcome (success or error) will be determined by the final response status code.

**Shortcut methods**

Shortcut methods are also available. All shortcut methods require passing in the URL, and request data must be passed in for POST/PUT requests. An optional config can be passed as the last argument.

$http.get('/someUrl', config).then(successCallback, errorCallback);

$http.post('/someUrl', data, config).then(successCallback, errorCallback);

Complete list of shortcut methods:

* [$http.get](https://docs.angularjs.org/api/ng/service/$http#get)
* [$http.head](https://docs.angularjs.org/api/ng/service/$http#head)
* [$http.post](https://docs.angularjs.org/api/ng/service/$http#post)
* [$http.put](https://docs.angularjs.org/api/ng/service/$http#put)
* [$http.delete](https://docs.angularjs.org/api/ng/service/$http#delete)
* [$http.jsonp](https://docs.angularjs.org/api/ng/service/$http#jsonp)
* [$http.patch](https://docs.angularjs.org/api/ng/service/$http#patch)

**Writing Unit Tests that use $http**

When unit testing (using [ngMock](https://docs.angularjs.org/api/ngMock)), it is necessary to call [$httpBackend.flush()](https://docs.angularjs.org/api/ngMock/service/$httpBackend#flush) to flush each pending request using trained responses.

$httpBackend.expectGET(...);

$http.get(...);

$httpBackend.flush();

**Setting HTTP Headers**

The $http service will automatically add certain HTTP headers to all requests. These defaults can be fully configured by accessing the $httpProvider.defaults.headers configuration object, which currently contains this default configuration:

* $httpProvider.defaults.headers.common (headers that are common for all requests):

Accept: application/json, text/plain, \*﻿/﻿\*

* $httpProvider.defaults.headers.post: (header defaults for POST requests)

Content-Type: application/json

* $httpProvider.defaults.headers.put (header defaults for PUT requests)

Content-Type: application/json

To add or overwrite these defaults, simply add or remove a property from these configuration objects. To add headers for an HTTP method other than POST or PUT, simply add a new object with the lowercased HTTP method name as the key, e.g.$httpProvider.defaults.headers.get = { 'My-Header' : 'value' }.

The defaults can also be set at runtime via the $http.defaults object in the same fashion. For example:

module.run(function($http) {

$http.defaults.headers.common.Authorization = 'Basic YmVlcDpib29w';

});

In addition, you can supply a headers property in the config object passed when calling $http(config), which overrides the defaults without changing them globally.

To explicitly remove a header automatically added via $httpProvider.defaults.headers on a per request basis, Use the headers property, setting the desired header to undefined. For example:

var req = {

method: 'POST',

url: 'http://example.com',

headers: {

'Content-Type': undefined

},

data: { test: 'test' }

}

$http(req).then(function(){...}, function(){...});

**Transforming Requests and Responses**

Both requests and responses can be transformed using transformation functions: transformRequest and transformResponse. These properties can be a single function that returns the transformed value (function(data, headersGetter, status)) or an array of such transformation functions, which allows you to push or unshift a new transformation function into the transformation chain.

**Note:** AngularJS does not make a copy of the data parameter before it is passed into the transformRequest pipeline. That means changes to the properties of data are not local to the transform function (since Javascript passes objects by reference). For example, when calling $http.get(url, $scope.myObject), modifications to the object's properties in a transformRequest function will be reflected on the scope and in any templates where the object is data-bound. To prevent this, transform functions should have no side-effects. If you need to modify properties, it is recommended to make a copy of the data, or create new object to return.

* **Default Transformations**

The $httpProvider provider and $http service expose defaults.transformRequest and defaults.transformResponse properties. If a request does not provide its own transformations then these will be applied.

You can augment or replace the default transformations by modifying these properties by adding to or replacing the array.

AngularJS provides the following default transformations:

Request transformations ($httpProvider.defaults.transformRequest and $http.defaults.transformRequest):

* + If the data property of the request configuration object contains an object, serialize it into JSON format.

Response transformations ($httpProvider.defaults.transformResponse and $http.defaults.transformResponse):

* + If XSRF prefix is detected, strip it (see Security Considerations section below).
  + If JSON response is detected, deserialize it using a JSON parser.

**Overriding the Default Transformations Per Request**

If you wish to override the request/response transformations only for a single request then provide transformRequest and/or transformResponse properties on the configuration object passed into $http.

Note that if you provide these properties on the config object the default transformations will be overwritten. If you wish to augment the default transformations then you must include them in your local transformation array.

The following code demonstrates adding a new response transformation to be run after the default response transformations have been run.

function appendTransform(defaults, transform) {

// We can't guarantee that the default transformation is an array

defaults = angular.isArray(defaults) ? defaults : [defaults];

// Append the new transformation to the defaults

return defaults.concat(transform);

}

$http({

url: '...',

method: 'GET',

transformResponse: appendTransform($http.defaults.transformResponse, function(value) {

return doTransform(value);

})

});

**Caching**

[$http](https://docs.angularjs.org/api/ng/service/$http) responses are not cached by default. To enable caching, you must set the config.cache value or the default cache value to TRUE or to a cache object (created with [$cacheFactory](https://docs.angularjs.org/api/ng/service/$cacheFactory)). If defined, the value of config.cache takes precedence over the default cache value.

In order to:

* cache all responses - set the default cache value to TRUE or to a cache object
* cache a specific response - set config.cache value to TRUE or to a cache object

If caching is enabled, but neither the default cache nor config.cache are set to a cache object, then the default $cacheFactory("$http")object is used.

The default cache value can be set by updating the [$http.defaults.cache](https://docs.angularjs.org/api/ng/service/$http#defaults) property or the [$httpProvider.defaults.cache](https://docs.angularjs.org/api/ng/provider/$httpProvider#defaults) property.

When caching is enabled, [$http](https://docs.angularjs.org/api/ng/service/$http) stores the response from the server using the relevant cache object. The next time the same request is made, the response is returned from the cache without sending a request to the server.

Take note that:

* Only GET and JSONP requests are cached.
* The cache key is the request URL including search parameters; headers are not considered.
* Cached responses are returned asynchronously, in the same way as responses from the server.
* If multiple identical requests are made using the same cache, which is not yet populated, one request will be made to the server and remaining requests will return the same response.
* A cache-control header on the response does not affect if or how responses are cached.

**Interceptors**

Before you start creating interceptors, be sure to understand the [$q and deferred/promise APIs](https://docs.angularjs.org/api/ng/service/$q).

For purposes of global error handling, authentication, or any kind of synchronous or asynchronous pre-processing of request or postprocessing of responses, it is desirable to be able to intercept requests before they are handed to the server and responses before they are handed over to the application code that initiated these requests. The interceptors leverage the [promise APIs](https://docs.angularjs.org/api/ng/service/$q) to fulfill this need for both synchronous and asynchronous pre-processing.

The interceptors are service factories that are registered with the $httpProvider by adding them to the $httpProvider.interceptorsarray. The factory is called and injected with dependencies (if specified) and returns the interceptor.

There are two kinds of interceptors (and two kinds of rejection interceptors):

* request: interceptors get called with a http [config](https://docs.angularjs.org/api/ng/service/$http#usage) object. The function is free to modify the config object or create a new one. The function needs to return the config object directly, or a promise containing the config or a new config object.
* requestError: interceptor gets called when a previous interceptor threw an error or resolved with a rejection.
* response: interceptors get called with http response object. The function is free to modify the response object or create a new one. The function needs to return the response object directly, or as a promise containing the response or a new responseobject.
* responseError: interceptor gets called when a previous interceptor threw an error or resolved with a rejection.

// register the interceptor as a service

$provide.factory('myHttpInterceptor', function($q, dependency1, dependency2) {

return {

// optional method

'request': function(config) {

// do something on success

return config;

},

// optional method

'requestError': function(rejection) {

// do something on error

if (canRecover(rejection)) {

return responseOrNewPromise

}

return $q.reject(rejection);

},

// optional method

'response': function(response) {

// do something on success

return response;

},

// optional method

'responseError': function(rejection) {

// do something on error

if (canRecover(rejection)) {

return responseOrNewPromise

}

return $q.reject(rejection);

}

};

});

$httpProvider.interceptors.push('myHttpInterceptor');

// alternatively, register the interceptor via an anonymous factory

$httpProvider.interceptors.push(function($q, dependency1, dependency2) {

return {

'request': function(config) {

// same as above

},

'response': function(response) {

// same as above

}

};

});

**Security Considerations**

When designing web applications, consider security threats from:

* [JSON vulnerability](http://haacked.com/archive/2008/11/20/anatomy-of-a-subtle-json-vulnerability.aspx)
* [XSRF](http://en.wikipedia.org/wiki/Cross-site_request_forgery)

Both server and the client must cooperate in order to eliminate these threats. AngularJS comes pre-configured with strategies that address these issues, but for this to work backend server cooperation is required.

**JSON Vulnerability Protection**

A [JSON vulnerability](http://haacked.com/archive/2008/11/20/anatomy-of-a-subtle-json-vulnerability.aspx) allows third party website to turn your JSON resource URL into [JSONP](http://en.wikipedia.org/wiki/JSONP) request under some conditions. To counter this your server can prefix all JSON requests with following string ")]}',\n". AngularJS will automatically strip the prefix before processing it as JSON.

For example if your server needs to return:

['one','two']

which is vulnerable to attack, your server can return:

)]}',

['one','two']

AngularJS will strip the prefix, before processing the JSON.

**Cross Site Request Forgery (XSRF) Protection**

[XSRF](http://en.wikipedia.org/wiki/Cross-site_request_forgery) is an attack technique by which the attacker can trick an authenticated user into unknowingly executing actions on your website. AngularJS provides a mechanism to counter XSRF. When performing XHR requests, the $http service reads a token from a cookie (by default, XSRF-TOKEN) and sets it as an HTTP header (X-XSRF-TOKEN). Since only JavaScript that runs on your domain could read the cookie, your server can be assured that the XHR came from JavaScript running on your domain. The header will not be set for cross-domain requests.

To take advantage of this, your server needs to set a token in a JavaScript readable session cookie called XSRF-TOKEN on the first HTTP GET request. On subsequent XHR requests the server can verify that the cookie matches X-XSRF-TOKEN HTTP header, and therefore be sure that only JavaScript running on your domain could have sent the request. The token must be unique for each user and must be verifiable by the server (to prevent the JavaScript from making up its own tokens). We recommend that the token is a digest of your site's authentication cookie with a [salt](https://en.wikipedia.org/wiki/Salt_(cryptography)) for added security.

The name of the headers can be specified using the xsrfHeaderName and xsrfCookieName properties of either $httpProvider.defaults at config-time, $http.defaults at run-time, or the per-request config object.

In order to prevent collisions in environments where multiple AngularJS apps share the same domain or subdomain, we recommend that each application uses unique cookie name.

**Dependencies**

* [$httpBackend](https://docs.angularjs.org/api/ng/service/$httpBackend)
* [$cacheFactory](https://docs.angularjs.org/api/ng/service/$cacheFactory)
* [$rootScope](https://docs.angularjs.org/api/ng/service/$rootScope)
* [$q](https://docs.angularjs.org/api/ng/service/$q)
* [$injector](https://docs.angularjs.org/api/auto/service/$injector)

**Arguments**

| **Param** | **Type** | | | **Details** | |
| --- | --- | --- | --- | --- | --- |
| config | | [**object**](https://docs.angularjs.org/) | Object describing the request to be made and how it should be processed. The object has following properties:   * **method** – {string} – HTTP method (e.g. 'GET', 'POST', etc) * **url** – {string|TrustedObject} – Absolute or relative URL of the resource that is being requested; or an object created by a call to $sce.trustAsResourceUrl(url). * **params** – {Object.<string|Object>} – Map of strings or objects which will be serialized with the paramSerializer and appended as GET parameters. * **data** – {string|Object} – Data to be sent as the request message data. * **headers** – {Object} – Map of strings or functions which return strings representing HTTP headers to send to the server. If the return value of a function is null, the header will not be sent. Functions accept a config object as an argument. * **eventHandlers** - {Object} - Event listeners to be bound to the XMLHttpRequest object. To bind events to the XMLHttpRequest upload object, use uploadEventHandlers. The handler will be called in the context of a $apply block. * **uploadEventHandlers** - {Object} - Event listeners to be bound to the XMLHttpRequest upload object. To bind events to the XMLHttpRequest object, use eventHandlers. The handler will be called in the context of a $apply block. * **xsrfHeaderName** – {string} – Name of HTTP header to populate with the XSRF token. * **xsrfCookieName** – {string} – Name of cookie containing the XSRF token. * **transformRequest** – {function(data, headersGetter)|Array.<function(data, headersGetter)>} – transform function or an array of such functions. The transform function takes the http request body and headers and returns its transformed (typically serialized) version. See [Overriding the Default Transformations](https://docs.angularjs.org/api/ng/service/$http#overriding-the-default-transformations-per-request) * **transformResponse** –{function(data, headersGetter, status)|Array.<function(data, headersGetter, status)>} – transform function or an array of such functions. The transform function takes the http response body, headers and status and returns its transformed (typically deserialized) version. See [Overriding the Default Transformations](https://docs.angularjs.org/api/ng/service/$http#overriding-the-default-transformations-per-request) * **paramSerializer** - {string|function(Object<string,string>):string} - A function used to prepare the string representation of request parameters (specified as an object). If specified as string, it is interpreted as function registered with the [$injector](https://docs.angularjs.org/api/auto/service/$injector), which means you can create your own serializer by registering it as a [service](https://docs.angularjs.org/api/auto/service/$provide#service). The default serializer is the [$httpParamSerializer](https://docs.angularjs.org/api/ng/service/$httpParamSerializer); alternatively, you can use the [$httpParamSerializerJQLike](https://docs.angularjs.org/api/ng/service/$httpParamSerializerJQLike) * **cache** – {boolean|Object} – A boolean value or object created with [$cacheFactory](https://docs.angularjs.org/api/ng/service/$cacheFactory) to enable or disable caching of the HTTP response. See [$http Caching](https://docs.angularjs.org/api/ng/service/$http#caching) for more information. * **timeout** – {number|Promise} – timeout in milliseconds, or [promise](https://docs.angularjs.org/api/ng/service/$q) that should abort the request when resolved. * **withCredentials** - {boolean} - whether to set the withCredentials flag on the XHR object. See [requests with credentials](https://developer.mozilla.org/docs/Web/HTTP/Access_control_CORS#Requests_with_credentials) for more information. * **responseType** - {string} - see [XMLHttpRequest.responseType](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest#xmlhttprequest-responsetype). | |

**Usage:-** $http(config);

Returns: Returns a [Promise](https://docs.angularjs.org/api/ng/service/$q) that will be resolved to a response object when the request succeeds or fails.

Methods

* get(url, [config]);

Shortcut method to perform GET request.

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| url | [**stringTrustedObject**](https://docs.angularjs.org/) | Absolute or relative URL of the resource that is being requested; or an object created by a call to $sce.trustAsResourceUrl(url). |
| config  (optional) | [**Object**](https://docs.angularjs.org/) | Optional configuration object |

#### **Returns**

|  |  |
| --- | --- |
| [**HttpPromise**](https://docs.angularjs.org/) | Future object |

* delete(url, [config]);

Shortcut method to perform DELETE request.

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| url | [**stringTrustedObject**](https://docs.angularjs.org/) | Absolute or relative URL of the resource that is being requested; or an object created by a call to $sce.trustAsResourceUrl(url). |
| config  (optional) | [**Object**](https://docs.angularjs.org/) | Optional configuration object |

#### **Returns**

|  |  |
| --- | --- |
| [**HttpPromise**](https://docs.angularjs.org/) | Future object |

* head(url, [config]);

Shortcut method to perform HEAD request.

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| url | [**stringTrustedObject**](https://docs.angularjs.org/) | Absolute or relative URL of the resource that is being requested; or an object created by a call to $sce.trustAsResourceUrl(url). |
| config  (optional) | [**Object**](https://docs.angularjs.org/) | Optional configuration object |

#### **Returns**

|  |  |
| --- | --- |
| [**HttpPromise**](https://docs.angularjs.org/) | Future object |

* jsonp(url, [config]);

Shortcut method to perform JSONP request.

Note that, since JSONP requests are sensitive because the response is given full access to the browser, the url must be declared, via [$sce](https://docs.angularjs.org/api/ng/service/$sce) as a trusted resource URL. You can trust a URL by adding it to the whitelist via [$sceDelegateProvider.resourceUrlWhitelist](https://docs.angularjs.org/api/ng/provider/$sceDelegateProvider#resourceUrlWhitelist) or by explicitly trusting the URL via [$sce.trustAsResourceUrl(url)](https://docs.angularjs.org/api/ng/service/$sce#trustAsResourceUrl).

JSONP requests must specify a callback to be used in the response from the server. This callback is passed as a query parameter in the request. You must specify the name of this parameter by setting the jsonpCallbackParam property on the request config object.

$http.jsonp('some/trusted/url', {jsonpCallbackParam: 'callback'})

You can also specify a default callback parameter name in $http.defaults.jsonpCallbackParam. Initially this is set to 'callback'.

You can no longer use the JSON\_CALLBACK string as a placeholder for specifying where the callback parameter value should go.

If you would like to customise where and how the callbacks are stored then try overriding or decorating the [$jsonpCallbacks](https://docs.angularjs.org/api/ng/service/$jsonpCallbacks) service.

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| url | [**stringTrustedObject**](https://docs.angularjs.org/) | Absolute or relative URL of the resource that is being requested; or an object created by a call to $sce.trustAsResourceUrl(url). |
| config  (optional) | [**Object**](https://docs.angularjs.org/) | Optional configuration object |

#### **Returns**

|  |  |
| --- | --- |
| [**HttpPromise**](https://docs.angularjs.org/) | Future object |

* post(url, data, [config]);

Shortcut method to perform POST request.

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| url | [**string**](https://docs.angularjs.org/) | Relative or absolute URL specifying the destination of the request |
| data | [**\***](https://docs.angularjs.org/) | Request content |
| config  (optional) | [**Object**](https://docs.angularjs.org/) | Optional configuration object |

#### **Returns**

|  |  |
| --- | --- |
| [**HttpPromise**](https://docs.angularjs.org/) | Future object |

* put(url, data, [config]);

Shortcut method to perform PUT request.

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| url | [**string**](https://docs.angularjs.org/) | Relative or absolute URL specifying the destination of the request |
| data | [**\***](https://docs.angularjs.org/) | Request content |
| config  (optional) | [**Object**](https://docs.angularjs.org/) | Optional configuration object |

#### **Returns**

|  |  |
| --- | --- |
| [**HttpPromise**](https://docs.angularjs.org/) | Future object |

* patch(url, data, [config]);

Shortcut method to perform PATCH request.

#### **Parameters**

| **Param** | **Type** | **Details** |
| --- | --- | --- |
| url | [**string**](https://docs.angularjs.org/) | Relative or absolute URL specifying the destination of the request |
| data | [**\***](https://docs.angularjs.org/) | Request content |
| config  (optional) | [**Object**](https://docs.angularjs.org/) | Optional configuration object |

#### **Returns**

|  |  |
| --- | --- |
| [**HttpPromise**](https://docs.angularjs.org/) | Future object |

## Properties

### **pendingRequests**

|  |  |
| --- | --- |
| [**Array.<Object>**](https://docs.angularjs.org/) | Array of config objects for currently pending requests. This is primarily meant to be used for debugging purposes. |

### **defaults**

|  |  |
| --- | --- |
|  | Runtime equivalent of the $httpProvider.defaults property. Allows configuration of default headers, withCredentials as well as request and response transformations.  See "Setting HTTP Headers" and "Transforming Requests and Responses" sections above. |

## ****How would you programatically change or adapt the template of a directive before it is executed and transformed?****

You would use the compile function. The compile function gives you access to the directive’s template before transclusion occurs and templates are transformed, so changes can safely be made to DOM elements. This is very useful for cases where the DOM needs to be constructed based on runtime directive parameters.

# $compile

Compiles an HTML string or DOM into a template and produces a template function, which can then be used to link [scope](https://docs.angularjs.org/api/ng/type/$rootScope.Scope) and the template together.

The compilation is a process of walking the DOM tree and matching DOM elements to [directives](https://docs.angularjs.org/api/ng/provider/$compileProvider#directive).

**Note:** This document is an in-depth reference of all directive options. For a gentle introduction to directives with examples of common use cases, see the [directive guide](https://docs.angularjs.org/guide/directive).

## 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)](https://docs.angularjs.org/api/ng/service/$compile#directive-definition-object) 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](https://docs.angularjs.org/api/ng/service/$compile#-priority-): 0,

[template](https://docs.angularjs.org/api/ng/service/$compile#-template-): '<div></div>', // or // function(tElement, tAttrs) { ... },

// or

// [templateUrl](https://docs.angularjs.org/api/ng/service/$compile#-templateurl-): 'directive.html', // or // function(tElement, tAttrs) { ... },

[transclude](https://docs.angularjs.org/api/ng/service/$compile#-transclude-): false,

[restrict](https://docs.angularjs.org/api/ng/service/$compile#-restrict-): 'A',

[templateNamespace](https://docs.angularjs.org/api/ng/service/$compile#-templatenamespace-): 'html',

[scope](https://docs.angularjs.org/api/ng/service/$compile#-scope-): false,

[controller](https://docs.angularjs.org/api/ng/service/$compile#-controller-): function($scope, $element, $attrs, $transclude, otherInjectables) { ... },

[controllerAs](https://docs.angularjs.org/api/ng/service/$compile#-controlleras-): 'stringIdentifier',

[bindToController](https://docs.angularjs.org/api/ng/service/$compile#-bindtocontroller-): false,

[require](https://docs.angularjs.org/api/ng/service/$compile#-require-): 'siblingDirectiveName', // or // ['^parentDirectiveName', '?optionalDirectiveName', '?^optionalParent'],

[multiElement](https://docs.angularjs.org/api/ng/service/$compile#-multielement-): false,

[compile](https://docs.angularjs.org/api/ng/service/$compile#-compile-): function compile(tElement, tAttrs, transclude) {

return {

[pre](https://docs.angularjs.org/api/ng/service/$compile#pre-linking-function): function preLink(scope, iElement, iAttrs, controller) { ... },

[post](https://docs.angularjs.org/api/ng/service/$compile#post-linking-function): function postLink(scope, iElement, iAttrs, controller) { ... }

}

// or

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

},

// or

// [link](https://docs.angularjs.org/api/ng/service/$compile#-link-): {

// [pre](https://docs.angularjs.org/api/ng/service/$compile#pre-linking-function): function preLink(scope, iElement, iAttrs, controller) { ... },

// [post](https://docs.angularjs.org/api/ng/service/$compile#post-linking-function): function postLink(scope, iElement, iAttrs, controller) { ... }

// }

// or

// [link](https://docs.angularjs.org/api/ng/service/$compile#-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) { ... }

});

### **Life-cycle hooks**

Directive controllers can provide the following methods that are called by AngularJS at points in the life-cycle of the directive:

* $onInit() - Called on each controller after all the controllers on an element have been constructed and had their bindings initialized (and before the pre & post linking functions for the directives on this element). This is a good place to put initialization code for your controller.
* $onChanges(changesObj) - Called whenever one-way (<) or interpolation (@) bindings are updated. The changesObj is a hash whose keys are the names of the bound properties that have changed, and the values are an object of the form { currentValue, previousValue, isFirstChange() }. Use this hook to trigger updates within a component such as cloning the bound value to prevent accidental mutation of the outer value. Note that this will also be called when your bindings are initialized.
* $doCheck() - Called on each turn of the digest cycle. Provides an opportunity to detect and act on changes. Any actions that you wish to take in response to the changes that you detect must be invoked from this hook; implementing this has no effect on when $onChanges is called. For example, this hook could be useful if you wish to perform a deep equality check, or to check a Date object, changes to which would not be detected by AngularJS's change detector and thus not trigger $onChanges. This hook is invoked with no arguments; if detecting changes, you must store the previous value(s) for comparison to the current values.
* $onDestroy() - Called on a controller when its containing scope is destroyed. Use this hook for releasing external resources, watches and event handlers. Note that components have their $onDestroy() hooks called in the same order as the $scope.$broadcast events are triggered, which is top down. This means that parent components will have their $onDestroy()hook called before child components.
* $postLink() - Called after this controller's element and its children have been linked. Similar to the post-link function this hook can be used to set up DOM event handlers and do direct DOM manipulation. Note that child elements that contain templateUrldirectives 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.

#### **Comparison with life-cycle hooks in the new Angular**

The new Angular also uses life-cycle hooks for its components. While the AngularJS life-cycle hooks are similar there are some differences that you should be aware of, especially when it comes to moving your code from AngularJS to Angular:

* AngularJS hooks are prefixed with $, such as $onInit. Angular hooks are prefixed with ng, such as ngOnInit.
* AngularJS hooks can be defined on the controller prototype or added to the controller inside its constructor. In Angular you can only define hooks on the prototype of the Component class.
* Due to the differences in change-detection, you may get many more calls to $doCheck in AngularJS than you would to ngDoCheckin Angular.
* Changes to the model inside $doCheck will trigger new turns of the digest loop, which will cause the changes to be propagated throughout the application. Angular does not allow the ngDoCheck hook to trigger a change outside of the component. It will either throw an error or do nothing depending upon the state of enableProdMode().

## ****How would you validate a text input field for a twitter username, including the @ symbol?****

You would use the ngPattern directive to perform a regex match that matches Twitter usernames. The same principal can be applied to validating phone numbers, serial numbers, barcodes, zip codes and any other text input.

# ngPattern

ngPattern adds the pattern [validator](https://docs.angularjs.org/api/ng/type/ngModel.NgModelController#$validators) to [ngModel](https://docs.angularjs.org/api/ng/directive/ngModel). It is most often used for text-based [input](https://docs.angularjs.org/api/ng/directive/input) controls, but can also be applied to custom text-based controls.

The validator sets the pattern error key if the [ngModel.$viewValue](https://docs.angularjs.org/api/ng/type/ngModel.NgModelController#$viewValue) does not match a RegExp which is obtained by evaluating the AngularJS expression given in the ngPattern attribute value:

* If the expression evaluates to a RegExp object, then this is used directly.
* If the expression evaluates to a string, then it will be converted to a RegExp after wrapping it in ^ and $ characters. For instance, "abc" will be converted to new RegExp('^abc$').

**Note:** Avoid using the g flag on the RegExp, as it will cause each successive search to start at the index of the last search's match, thus not taking the whole input value into account.

**Note:** This directive is also added when the plain pattern attribute is used, with two differences:

1. ngPattern does not set the pattern attribute and therefore HTML5 constraint validation is not available.
2. The ngPattern attribute must be an expression, while the pattern value must be interpolated.

## Directive Info

* This directive executes at priority level 0.

## Usage

* as element: (This directive can be used as custom element, but be aware of [IE restrictions](https://docs.angularjs.org/guide/ie)).
* <ng-pattern>
* ...

</ng-pattern>

* as attribute:
* <ANY>
* ...

</ANY>

## Examples

Index.html

<script>

angular.module('ngPatternExample', [])

.controller('ExampleController', ['$scope', function($scope) {

$scope.regex = '\\d+';

}]);

</script>

<div ng-controller="ExampleController">

<form name="form">

<label for="regex">Set a pattern (regex string): </label>

<input type="text" ng-model="regex" id="regex" />

<br>

<label for="input">This input is restricted by the current pattern: </label>

<input type="text" ng-model="model" id="input" name="input" ng-pattern="regex" /><br>

<hr>

input valid? = <code>{{form.input.$valid}}</code><br>

model = <code>{{model}}</code>

</form>

</div>

Protractor.js

var model = element(by.binding('model'));

var input = element(by.id('input'));

it('should validate the input with the default pattern', function() {

input.sendKeys('aaa');

expect(model.getText()).not.toContain('aaa');

input.clear().then(function() {

input.sendKeys('123');

expect(model.getText()).toContain('123');

});

});

## ****How would you implement application-wide exception handling in your Angular app?****

Angular has a built-in error handler service called $exceptionHandler which can easily be overriden as seen below:

myApp.factory('$exceptionHandler', function($log, ErrorService) {

return function(exception, cause) {

if (console) {

$log.error(exception);

$log.error(cause);

}

ErrorService.send(exception, cause);

};

});

This is very useful for sending errors to third party error logging services or helpdesk applications. Errors trapped inside of event callbacks are not propagated to this handler, but can manually be relayed to this handler by calling $exceptionHandler(e) from within a try catch block.

## ****How do you hide an HTML element via a button click in AngularJS?****

You can do this by using the ng-hide directive in conjunction with a controller we can hide an HTML element on button click.

<div ng-controller="MyCtrl">

<button ng-click="hide()">Hide element</button>

<p ng-hide="isHide">Hello World!</p>

</div>

function MyCtrl($scope){

$scope.isHide = false;

$scope.hide = function(){

$scope.isHide = true;

}

}

## ****How would you react on model changes to trigger some further action? For instance, say you have an input text field called**** email ****and you want to trigger or execute some code as soon as a user starts to type in their email.****

We can achieve this using $watch function in our controller.

function MyCtrl($scope) {

$scope.email = "";

$scope.$watch("email", function(newValue, oldValue) {

if ($scope.email.length > 0) {

console.log("User has started writing into email");

}

});

}

## ****How do you disable a button depending on a checkbox’s state?****

We can use the ng-disabled directive and bind its condition to the checkbox’s state.

<body ng-app>

<label><input type="checkbox" ng-model="checked"/>Disable Button</label>

<button ng-disabled="checked">Select me</button>

</body>

## When to use directives, controllers, or services in Angular JS

[http://kirkbushell.me/assets/img/logo-angularjs.png](http://kirkbushell.me/when-to-use-directives-controllers-or-services-in-angular/)

Angular JS is a very powerful front-end MVC framework. It also introduces many concepts that may be unfamiliar. A few of these are:

* Directives
* Controllers
* Services

Let’s look at each one in turn and investigate why each concept is great at what they’ve been designed for, and why we’d use them in that fashion. Let’s start from the bottom.

**Services**

If you’ve worked with Angular JS already, you may have come across services - which are basically a nice name for Angular singletons. These puppies get passed around regularly, ensuring that you’re dealing with the same object each time, unlike factories. With that in mind, it allows us to do some pretty cool things, like have various controllers or directives affect its values. This is another common question in the #angularjs channel, and is how you share data between chunks of code in your application. Let’s have a look.

First we’ll setup a module. We’ll use this throughout the article.

var module = angular.module( "my.new.module", [] );

Next, we’ll create a new service. Let’s say this module will be used for managing books. So let’s create a book service, and add to it an array of JSON objects containing various pieces of book data.

1 module.service( 'Book', [ '$rootScope', function( $rootScope ) {

2 var service = {

3 books: [

4 { title: "Magician", author: "Raymond E. Feist" },

5 { title: "The Hobbit", author: "J.R.R Tolkien" }

6 ],

7

8 addBook: function ( book ) {

9 service.books.push( book );

10 $rootScope.$broadcast( 'books.update' );

11 }

12 }

13

14 return service;

15 }]);

This is a very simple service (but sometimes that’s all you need). All we’re doing is managing an array of books, with a helper method to add more books should the need arise. This also broadcasts an event to the application to notify anything using our service that the array has been updated, and likewise - they should also do an update. Now, what we can do is pass this service around to the various controllers, directives, filters or whatever else may need it - and they’ll have access to these methods and properties. So let’s do that now.

1 var ctrl = [ '$scope', 'Book', function( scope, Book ) {

2 scope.$on( 'books.update', function( event ) {

3 scope.books = Book.books;

4 });

5

6 scope.books = Book.books;

7 }];

8

9 module.controller( "books.list", ctrl );

Again, pretty straight forward. What we’ve done is create a new controller for our module. This is then provided with both the $scope provider, and the Book service we’ve just created. Notice what we’ve done? We’ve assigned the books array we created earlier on the Book service, to a property called books on the controller’s local scope object. Cool, huh?

So, what’s the point? We could have saved some time and just created the array on the controller. Correct - we could have. And it may have saved us time - but what if we needed to handle those books elsewhere? Managing data via scopes is reckless. Scopes can easily become corrupted or dirtied by other controllers, directives, models and the like. It gets messy quickly. Having a central channel (in this case a service) to manage all book data, and requests to modify it any way, not only is a lot cleaner - it’s also much easier to manage as the application grows. Lastly - it keeps your code modular (something Angular excels at). Should you ever need that service for another project, you don’t need to dig through scopes, controllers, filters, etc. to find relevant code - it’s all there in the service!

So. When do we use services? Whenever we want to share data across domains. Additionally, thanks to Angular’s dependency injection system, this is both very easy to do and very clean.

**Controllers**

That brings us to controllers! Unless you’ve worked with front-end MVC before, moving from the server to the client can be a bit of a head twist. Why? Because although controllers for front-end development serve a very similar purpose, it’s also done very differently to what you may be used to on the server. Controllers in Angular do not handle “requests” per se, unless it’s to handle routes (many are calling this type of setup a route controller), more specifically, especially if used as part of your application’s interface - they may just be managing a small subset of data.

Controllers should be used purely to wire up services, dependencies and other objects, and assign them to the view via scope. They’re also an excellent choice when needing to handle complex business logic in your views. Taking our books example earlier, there isn’t really anything I’d want to add to the controller.

But Kirk, what about if i want to add a book? Shouldn’t I add a method on the controller to handle that? No, and here’s why. It’s part of DOM interaction/modification. Put that in a directive. How? I’m glad you asked.

**Directives**

In the various applications I’ve written in Angular so far, by far the majority of the application’s complexity is in the directives. They are a powerful tool for working with and modifying the DOM, and that’s what we’re going to do here. To wrap this article, we’ll provide a button for a user to be able to add a new book to the service.

A common anti-pattern (in my humble opinion) is adding DOM interactions to the controller. Angular defines directives as chunks of code you use for DOM manipulation, but I feel it’s also great for interactions. Let’s extend out example above by providing the user with the ability to add a book to the service.

1 module.directive( "addBookButton", [ 'Book', function( Book ) {

2 return {

3 restrict: "A",

4 link: function( scope, element, attrs ) {

5 element.bind( "click", function() {

6 Book.addBook( { title: "Star Wars", author: "George Lucas" } );

7 });

8 }

9 }

10 }]);

Simple stuff. We’re creating a directive, whose sole purpose is to simply add a book to the list of books already registered on our Book service. Let’s incorporate that in our view.

1 <button add-book-button>Add book</button>

As you can see, we’re just adding the directive as an attribute. Every time this button is now clicked, it will add the book Star Wars to our Book service. Simple, easy, modular - and easily transferable. Now, why wouldn’t we just add a method called something like addBook on our controller, that does something like the following:

1 $scope.addBook = function() {

2 Book.addBook( { title: "Star Wars", author: "George Lucas" } );

3 };

This would give us the same result, right? Yes, it would - with one major caveat. Should I need to add books again from another location - I have to copy that code (very un-DRY!) or refactor (which itself isn’t a bad thing). By building a directive straight off the bat, we don’t need to worry about that later - and it takes us next to no time at all to do so. By building directives for DOM interaction and modification, we’re immediately setting ourself up to handle increasing application complexity as business requirements roll in. This is a good thing, as it ensures we fight our own implementations less, and write DRYer code all the time.

Angular’s philosophy of modular development really lends its weight to non-trivial applications. It allows us to write our front-end code in such a way that we don’t end up tripping over ourselves, or the framework - and this is probably its greatest strength.

Hopefully I have shown when and where you would like to use these various Angular concepts, in order to get the most out of your own code.

# SERVICE VS FACTORY ONCE AND FOR ALL

Wait, what? Yet another article that answers the big question: Service vs Factory, what should I use? Yes, it seems that this is not needed anymore, since there are a ton of resources in the internet that discuss that topic. It turns out that this question still pops up every week or so on different channels, and even after reading the top ten answers on StackOverflow, it’s still not very clear. Despite that, it also appears that the current resources on the web don’t really promote the actual best practice, especially if we consider the recent movements of the web platform. ES6 I’m looking at you!

This article explains once and for all the difference between services and factories and why **we want to prefer services over factories**.

### [TABLE OF CONTENTS](https://blog.thoughtram.io/angular/2015/07/07/service-vs-factory-once-and-for-all.html#table-of-contents)

* [The difference between services and factories](https://blog.thoughtram.io/angular/2015/07/07/service-vs-factory-once-and-for-all.html#the-difference-between-services-and-factories)
* [Which one to use?](https://blog.thoughtram.io/angular/2015/07/07/service-vs-factory-once-and-for-all.html#which-one-to-use)
* [Services allow us to use ES6 classes](https://blog.thoughtram.io/angular/2015/07/07/service-vs-factory-once-and-for-all.html#services-allow-us-to-use-es6-classes)

## [The difference between services and factories](https://blog.thoughtram.io/angular/2015/07/07/service-vs-factory-once-and-for-all.html#the-difference-between-services-and-factories)

Okay, so what is the difference between a service and a factory in AngularJS? As we all know, we can define a service like this:

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

this.sayHello = function () {

console.log('hello');

};

});

.service() is a method on our module that takes a name and a function that defines the service. Pretty straight forward. Once defined, we can inject and use that particular service in other components, like controllers, directives and filters, like this:

app.controller('AppController', function (MyService) {

MyService.sayHello(); // logs 'hello'

});

Okay, clear. Now the same thing as a factory:

app.factory('MyService', function () {

return {

sayHello: function () {

console.log('hello');

}

}

});

Again, .factory() is a method on our module and it also takes a name and a function, that defines the factory. We can inject and use that thing exactly the same way we did with the service. Now what is the difference here?

Well, you might see that instead of working with this in the factory, we’re returning an object literal. Why is that? It turns out, **a service is a constructor function** whereas a factory is not. Somewhere deep inside of this Angular world, there’s this code that calls Object.create()with the service constructor function, when it gets instantiated. However, a factory function is really just a function that gets called, which is why we have to return an object explicitly.

To make that a bit more clear, we can simply take a look at the Angular source code. Here’s what the factory() function looks like:

function factory(name, factoryFn, enforce) {

return provider(name, {

$get: enforce !== false ? enforceReturnValue(name, factoryFn) : factoryFn

});

}

It takes the name and the factory function that is passed and basically returns a provider with the same name, that has a $get method which is our factory function. So what is it with this provider thing? Well, whenever you ask the injector for a specific dependency, it basically asks the corresponding provider for an instance of that service, by calling the $get() method. That’s why $get() is required, when creating providers.

In other words, if we inject MyService somewhere, what happens behind the scenes is:

MyServiceProvider.$get(); // return the instance of the service

Alright, factory functions just get called, what about the service code? Here’s another snippet:

function service(name, constructor) {

return factory(name, ['$injector', function($injector) {

return $injector.instantiate(constructor);

}]);

}

Oh look, it turns out that when we call service() it actually calls factory(). But it doesn’t just pass our service constructor function to the factory as it is. It passes a function that asks the injector to instantiate and object by the given constructor. In other words: a service calls a predefined factory, which ends up as $get() method on the corresponding provider. $injector.instantiate() is the method that ultimately calls Object.create() with the constructor function. That’s why we use this in services.

Okay, so it turns out that, no matter what we use, service() or factory(), it’s always a factory that is called which creates a provider for our service. Which brings us to the mostly asked question in the Angular history: Which one should I use?

## [Which one to use?](https://blog.thoughtram.io/angular/2015/07/07/service-vs-factory-once-and-for-all.html#which-one-to-use)

Asking that question on the internet takes us to a couple of articles and StackOverflow answers. The first is [this](https://stackoverflow.com/questions/13762228/confused-about-service-vs-factory) answer. It says:

“Basically the difference between the service and factory is as follows:”

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

// service is just a constructor function

// that will be called with 'new'

this.sayHello = function(name) {

return "Hi " + name + "!";

};

});

app.factory('myFactory', function() {

// factory returns an object

// you can run some code before

return {

sayHello : function(name) {

return "Hi " + name + "!";

}

}

});

We now already know what happens behind the scenes, but this answer adds another comment. It says we can run code **before** we return our object literal. That basically allows us to do some configuration stuff or conditionally create an object or not, which doesn’t seem to be possible when creating a service directly, which is why **most resources recommend to use factories over services, but the reasoning is inappreciable.**

What if I told you, **we can do the exact same thing with services too?**

Yeap, correct. A service is a constructor function, however, that doesn’t prevent us from doing additional work and return object literals. In fact, constructor functions in JavaScript can return whatever they want. So we can take our service code and write it in a way that it basically does the exact same thing as our factory:

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

// we could do additional work here too

return {

sayHello: function () {

console.log('hello');

};

}

});

Hoppla, so what now? We just realised that, depending on how we write our services, there’s no difference between the two at all anymore. The big question remains: Which one should we use?

## [Services allow us to use ES6 classes](https://blog.thoughtram.io/angular/2015/07/07/service-vs-factory-once-and-for-all.html#services-allow-us-to-use-es6-classes)

Of course, writing services in that way is kind of contra productive, since it’s called as a constructor function, so it should also be used like one. Is there any advantage over the other at all then? Yes, there is. It turns out that it’s actually better to use services where possible, when it comes to migrating to ES6. The reason for that is simply that a service is a constructor function and a factory is not. Working with constructor functions in ES5 allows us to easily use ES6 classes when we migrate to ES6.

For example, we can take our code and rewrite it in ES6 like this:

class MyService {

sayHello() {

console.log('hello');

}

}

app.service('MyService', MyService);

An ES6 class is really just a constructor function in ES5. We wrote about that in [Using ES6 with Angular today](http://blog.thoughtram.io/angularjs/es6/2015/01/23/exploring-angular-1.3-using-es6.html), if you haven’t read that article yet, I’d recommend checking that out.

With factories, this is not possible because they are simply called as functions. I hope this article made everything clear and encourages people to **not** use factories over services, if they don’t know what to use.

# ngBind

1. **- directive in module**[**ng**](https://docs.angularjs.org/api/ng)

The ngBind attribute tells AngularJS to replace the text content of the specified HTML element with the value of a given expression, and to update the text content when the value of that expression changes.

Typically, you don't use ngBind directly, but instead you use the double curly markup like {{ expression }} which is similar but less verbose.

It is preferable to use ngBind instead of {{ expression }} if a template is momentarily displayed by the browser in its raw state before AngularJS compiles it. Since ngBind is an element attribute, it makes the bindings invisible to the user while the page is loading.

An alternative solution to this problem would be using the [ngCloak](https://docs.angularjs.org/api/ng/directive/ngCloak) directive.

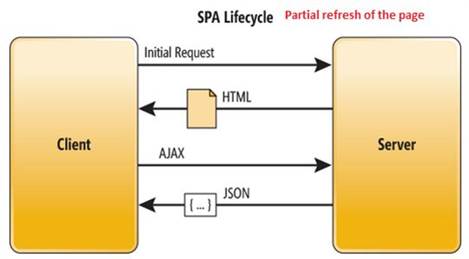
## Usage

* as attribute:
* <ANY
* ng-bind="expression">
* ...

</ANY>

* as CSS class:

<ANY class="ng-bind: expression;"> ... </ANY>

What is SPA (Single page application) in AngularJS?  
  
Single-Page Applications (SPAs) are web applications that load a single HTML page and dynamically update that page as the user interacts with the app. SPAs use AJAX and HTML to create fluid and responsive web apps, without constant page reloads. However, this means much of the work happens on the client side, in JavaScript.  
  
A single HTML page here means UI response page from the server. The source can be ASP, ASP.NET, ASP.NET MVC, JSP and so on.  
  
A single-page web application, however, is delivered as one page to the browser and typically does not require the page to be reloaded as the user navigates to various parts of the application. This results in faster navigation, more efficient network transfers, and better overall performance for the end user.  
  
  
  
**Key Points of Single-Page Applications**

* The application is responsive in the UI with no page flicker
* The Back/Forward buttons work as expected
* More JavaScript than actual HTML
* Dynamic data loading from the server-side API works with restful web service with JSON format
* Rich interaction among UI components
* Fewer data transfers from the server and most of the page processes in the UI occurs client-side.
* The application contains tabs and subtabs with multiple HTML containers on the click of the tabs or subtabs and the specific portions of the page that are loaded into the page (the page will be one using the application)
* Applications written in AngularJS are cross-browser compliant. Angular automatically handles the JavaScript code suitable for each browser.

How are validations implemented in AngularJS?One of the coolest features of AngularJS is client-side validation. There are so many form directives available in AngularJS. We will talk about some of them here, we will also explain custom validation. Using it you can create your own validations.  
  
Initial requirement is reference,

<script src="~/Scripts/angular.js"></script>

**Data type validation**In Html control use type field to specify the type of file.  
$error.{your data type} will help you to disply the message.

<p>

    <input type="number" name="StudentRollNumber" ng-model="StudentRollNumber" required>

    <span style="color:red" ng- show="myForm.StudentRollNumber.$dirty && myForm.StudentRollNumber.$invalid">

<span ng-show="myForm.StudentRollNumber.$error.required">Student Roll Number is required.</span>

    <span ng-show="myForm.StudentRollNumber.$error.number">Not valid number!</span>

    </span>

</p>

**Required filed validation**Put attribute as required in HTML control.  
$error.required helps you to display the required field message.

<p>

    <input type="text" name="Student" ng-model="Student" required>

    <span style="color:red" ng-show="myForm.Student.$dirty && myForm.Student.$invalid">

<span ng-show="myForm.Student.$error.required">Student Name is required.</span>

    </span>

</p>

**Date Validation**a. Specify the type as date and  
b. Format it will take as systems built-in format  
c. .$error.date helps you to display the required field message.

<p>

    Student Birth Date:<br>

    <input type="date" name="BirthDate" ng-model="BirthDate" required placeholder="yyyy-MM-dd">

    <span style="color:red" ng-show="myForm.BirthDate.$dirty && myForm.BirthDate.$invalid">

<span ng-show="myForm.BirthDate.$error.required">Student Birth Date is required.</span>

    <span ng-show="myForm.BirthDate.$error.date">Not a Valid Date.</span>

    </span>

</p>

**Email Validation**a. Specify the type as Email and  
b..$error.email helps you to display the required field message.

<input type="email" name="email" ng-model="email" required>

<span style="color:red" ng-show="myForm.email.$dirty && myForm.email.$invalid">

<span ng-show="myForm.email.$error.required">Email is required.</span>

<span ng-show="myForm.email.$error.email">Invalid email address.</span>

</span>

**Range Validation Max and Min**a. Specify Max or Min attribute  
b..$error.max or .$error.min helps you to display the error message.

<input type="number" name="marks" ng-model="marks" max="100" required>

<span style="color:red" ng-show="myForm.marks.$dirty && myForm.marks.$invalid">

<span ng-show="myForm.marks.$error.required">Email is required.</span>

<span ng-show="myForm.marks.$error.number">Invalid number.</span>

<span ng-show="myForm.marks.$error.max">Max Percentage is 100.</span>

</span>

What is Representational State Transfer(REST) in AngularJS.REST is a style of API that operates over HTTP requests. The requested URL identifies the data to be operated on, and the HTTP method identifies the operation that is to be performed. REST is a style of API rather than a formal specification, and there is a lot of debate and disagreement about what is and isn’t RESTful, a term used to indicate an API that follows the REST style. AngularJS is pretty flexible about how RESTful web services are consumed. You should use the services that I describe in this article when you are performing data operations on a RESTful API. You may initially prefer to use the $http service to make Ajax requests, especially if you are coming from a jQuery background. To that end, I describe the use of $http at the start of the article, before explaining its limitations when used with REST and the advantages of using the $resource service as an alternative. For this, we first need to create a RESTful web API.  
  
A REST web service is said to be RESTful when it adheres to the following constrants:

* It’s URL-based (e.g., <http://www.micbutton.com/rs/blogPost>).
* It uses an Internet media type such as JSON for data interchange.
* It uses standard HTTP methods (GET, PUT, POST, DELETE).

HTTP methods have a particular purpose when used with REST services. The following is the standard way that HTTP methods should be used with REST services,  
  
**POST should be used to,**

* Create a new resource.
* Retrieve a list of resources when a large amount of request data is required to be passed to the service.
* PUT should be used to update a resource.
* GET should be used to retrieve a resource or a list of resources.
* DELETE should be used to delete a resource.

**For doing this, we first create a model class with the below mention members**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Required** |
| name | String | Yes |
| category | String | Yes |
| price | number | Yes |

Explain ng-Switch Directive in AngularJS.  
  
This directive is used to swap DOM structure conditionally in our template based on a scope expression. ngSwitchWhen or ngSwitchDefault directives are used to show and hide the element within ngSwitch directive. We can show or hide the element inside this directive and are required to place a "when" attribute per element. The "when" attribute is used to inform the ngSwitch directive which element is to display based on expression, if the matching expression is found, the element is displayed, else it is hidden.  
  
**HTML**

<h4>ngSwitch Example</h4>

<div ng-controller="HelloController">

 Employee Name:

    <select ng-model="selection" ng-options="name for name in names"></select>

    <div ng-switch on="selection">

        <div ng-switch-when="Tejas">You have select "Tejas"</div>

        <div ng-switch-when="Rakesh">You have select "Rakesh"</div>

        <div ng-switch-when="Jignesh">You have select "Jignesh"</div>

        <div ng-switch-default>Please select name</div>

    </div>

</div>

**Controller**

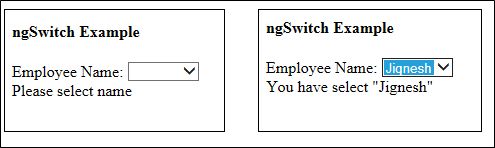
var app = angular.module("app", []);

app.controller("HelloController", function($scope)

{

    $scope.names = ['Tejas', 'Jignesh', 'Rakesh'];

});

**Output**  


The ngSwitch directive is used to conditionally swap DOM structure on your template based on a scope expression. Elements within ngSwitch but without ngSwitchWhen or ngSwitchDefault directives will be preserved at the location as specified in the template.

The directive itself works similar to ngInclude, however, instead of downloading template code (or loading it from the template cache), ngSwitch simply chooses one of the nested elements and makes it visible based on which element matches the value obtained from the evaluated expression. In other words, you define a container element (where you place the directive), place an expression on the **on="..." attribute** (or the **ng-switch="..." attribute**), define any inner elements inside of the directive and place a when attribute per element. The when attribute is used to inform ngSwitch which element to display when the on expression is evaluated. If a matching expression is not found via a when attribute then an element with the default attribute is displayed.

Be aware that the attribute values to match against cannot be expressions. They are interpreted as literal string values to match against. For example, **ng-switch-when="someVal"** will match against the string "someVal" not against the value of the expression $scope.someVal.

Why to use AngularJS Global Object services?  
  
The main reason that AngularJS includes these services is to make testing easier, but an important facet of unit testing is the need to isolate a small piece of code and test its behavior without testing the components it depends on—in essence, creating a focused test. The DOM API exposes functionality through global objects such as document and window.   
  
These objects make it hard to isolate code for unit testing without also testing the way that the browser implements its global objects. Using services such as $document allows AngularJS code to be written without directly using the DOM API global objects and allows the use of AngularJS testing services to configure specific test scenarios.  
  
The followings are the services that expose DOM API features.

|  |  |
| --- | --- |
| **Name** | **Descriptions** |
| $anchorScroll | Scrolls the browser window to a specified anchor |
| $document | Provides a jqLite object that contains the DOM window.document object |
| $interval | Provides an enhanced wrapper around the window.setInterval function |
| $location | Provides access to the URL |
| $log | Provides a wrapper around the console object |
| $timeout | Provides an enhanced wrapper around the window.setITimeout function |
| $window | Provides a reference to the DOM window object |

Angular JS built in services

|  |  |
| --- | --- |
| **Name** | **Descriptions** |
| $anchorScroll | Scrolls the browser window to a specified anchor |
| $animate | Animates the content transitions. |
| $compile | Processes an HTML fragment to create a function that can be used to generate content. |
| $controller | A wrapper around the $injector service that instantiates controllers |
| $document | Provides a jqLite objects that contains the DOM window.documentobject. |
| $exceptionHandler | Handles exceptions that arise in the application. |
| $filter | Provides access to filters |
| $http | Creates and manages Ajax requests |
| $injector | Creates instances of AngularJS components |
| $interpolate | Processes a string that contains binding expressions to create a function that can be used to generate content. |
| $interval | Provides an enhanced wrapper around the window.setInterval function. |
| $location | Provides a wrapper around the browser location object. |
| $log | Provides a wrapper around the global console object. |
| $parse | Processes an expression to create a function that can be used togenerate content. |
| $provide | Implements many of the methods that are exposed by Module. |
| $q | Provides deferred objects/promises. |
| $resource | Provides support for working with RESTful APIs. |
| $rootElement | Provides access to the root element in the DOM. |
| $rootScope | Provides access to the top of the scope hierarchy |
| $route | Provides support for changing view content based on the browser’sURL path. |
| $routeParams | Provides information about URL routes. |
| $sanitize | Replaces dangerous HTML characters with their display-safecounterparts. |
| $swipe | Recognizes swipe gestures. |
| $timeout | Provides an enhanced wrapper around the window.setITimeout function. |
| $window | Provides a reference to the DOM window object. |

What are the attributes can be used during creation of a new AngularJS Directives?  
  
The following attributes can be used during creation of a new AngularJS Directives,

1. **Restrict:** The restrict attribute is how AngularJS triggers the directive inside a template. The default value of the restrict option is “A”. The value of “A” causes the directives to be triggered on the attribute name. Other than “A”, restrict option has “E” (only match element name), “C” (only match class name) and “M” (only match the comment name) or any combination among four options.
2. **TemplateUrl:** The templateUrl attribute tells the AngularJS HTML compiler to replace custom directive inside a template with HTML content located inside a separate file. The link-Menu (say, our custom directive name) attribute will be replaced with the content of our original menu template file.
3. **Template:** Specify an inline template as a string. Not used if you’re specifying your template as a URL.
4. **Replace :** If true, replace the current element. If false or unspecified, append this directive to the current element.
5. **Transclude :** Lets you move the original children of a directive to a location inside the new template.
6. **Scope :** Create a new scope for this directive rather than inheriting the parent scope.
7. **Controller :** Create a controller which publishes an API for communicating across directives.
8. **Require:** Require that another directive be present for this directive to function correctly.
9. **Link:** Programmatically modify resulting DOM element instances, add event listeners, and set up data binding.
10. **Compile :** Programmatically modify the DOM template for features across copies of a directive, as when used in other directives. Your compile function can also return link functions to modify the resulting element instances.

Give the differences between AngularJS and Backbone and Knockout?Comparison with Backbone.js and Knockout.js,

|  |  |  |  |
| --- | --- | --- | --- |
| **Comparison** | **AngularJs** | **Backbone.js** | **Knockout.js** |
| File Size | ~142 KB total (compressed and minified) | ~ 7.3 KB total (gzip / minified) | ~21 KB total (gzip / minified) |
| Version & Licence | V1.4.2 & MIT (Open-source) | V1.2.1 & MIT (Open-source) | V3.3.0 & MIT (Open-source) |
| Dependencies | No Dependencies | Dependends on underscore.js and jQuery | No Dependencies |
| Data Binding | It supports full data binding and provides options for creating custom data bindings | Does not support data binding by default but does using plugins for data bindings | It fully supports data binding and can bind many attributes. It provides options for creating custom data bindings |
| Routing | It supports routing feature and it's very simple | It supports routing features and it's very simple | Does not support routing by defualt but it is available with some thrid-party libraries |
| Views | Uses HTML as the templating language | Does not have templates by default but we can add them easily by a thrid-party template like underscore.js and handlebars | It uses HTML as the templating language |
| Testing | Can support Test Driven Development (TDD) | Does not support testing by defualt but we can use some thrid-party tester like Jasmine and Sinon.JS | Does not support testing by defualt but we can use some thrid-party tester like Jasmine and Sinon.JS |
| Data | Does not support jQuery but we can use Angular's $http | Can support jQuery's $.ajax and is very easy to understand | It can support jQuery's $.ajax and knockout mapping |
| Design Pattern | Can support the MVC and MVVM design patterns | It can support MVP design pattern | It can support the MVVM design pattern |
| Browser | Can support IE 9, IE 10 and IE 11 | It dependends on jQuery supporting browsers like IE 6+, Chrome, Firefox, Safari 5.1+ and Opera | It can support all major browsers like IE 6+, Firefox 3.5+, Chrome, Opera and Safari |
| Third-party Integration | Does not support third-party integration | Does not support third-party integration | It supports third-party integration |
| Documentation | It has available documentation and community | To my knowledge there is no documentation | It has available documentation and community |

Why is scopeless controller used in AngularJS?Sometimes controller become complex by using $scope for providing data and behavior to view, in that situation we can use scopeless controller.   
  
But if you have designed your AngularJS application perfectly, there is no need to go for scopeless controllers.  
  
**Creating scope-less controller**

* angular module(app.js):
* angular.module('myApp', []);

**Controller (homeController.js)**

var app = angular.module("myApp");

app.controller("myController", function()

{

    this.title = 'scopeless Controller Test';

    this.name = 'Anupam';

    this.sayHello = function()

    {

        alert('Hello ' + this.name);

    }

});

As you can see I have used JavaScript for this keyword to add data and behavior in my controller.  
  
I would love to explain this here but I am still exploring what thi is in JavaScript.  
   
Here is how we can get data from controller to view.  
  
Binding Data to View using scope-less controller  
  
**View (index.html)**

<!DOCTYPE html>

<html ng-app="myApp" ng-controller="myController as ctrl">

<head>

    <script src="Scripts/angular.min.js"></script>

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

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

    <link href="Css/bootstrap.min.css" rel="stylesheet" />

    <title>{{ctrl.title}}</title>

</head>

<body>

    <nav role="navigation" class=" navbar navbar-default">

        <div class="navbar-header">

            <a href="#" class="navbar-brand">

 {{ctrl.title}}

 </a>

        </div>

    </nav>

    <div class="container body-content">

        <div class="col md-6">

            <div class="row">

                <div class="well-lg">

                    Hi {{ctrl.name}}

                </div>

            </div>

            <div class="row">

                <div class="well-lg">

                    <input type="button" ng-click="ctrl.sayHello()" value="Say Hello" class="btn" />

                </div>

            </div>

        </div>

    </div>

</body>

</html>

Here I have used a variable ctrl (myController as ctrl) which is an instance of myController.

## Is AngularJS a library, framework, plugin or a browser extension?

AngularJS fits the definition of a framework the best, even though it's much more lightweight than a typical framework and that's why many confuse it with a library.

AngularJS is 100% JavaScript, 100% client-side and compatible with both desktop and mobile browsers. So it's definitely not a plugin or some other native browser extension.

## Is AngularJS a templating system?

At the highest level, AngularJS does look like just another templating system. But there is one important reason why the AngularJS templating system is different, that makes it very good fit for application development: bidirectional data binding. The template is compiled in the browser and the compilation step produces a live view. This means you, the developers, don't need to write code to constantly sync the view with the model and the model with the view as in other templating systems.

## How big is the angular.js file that I need to include?

The size of the file is ~50KB compressed and minified.

## Can I use the open-source Closure Library with AngularJS?

Yes, you can use widgets from the [Closure Library](https://developers.google.com/closure/library/) in AngularJS.

## What is the Closure Library?

The Closure Library is a broad, well-tested, modular, and cross-browser JavaScript library. You can pull just what you need from a large set of reusable UI widgets and controls, and from lower-level utilities for DOM manipulation, server communication, animation, data structures, unit testing, rich-text editing, and more.

The Closure Library is server-agnostic, and is intended for use with the [Closure Compiler](https://developers.google.com/closure/compiler).

## Who uses Closure Library?

The Closure Library serves as the base JavaScript library for many Google products, including:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [Search](http://www.google.com/) | [Gmail](http://mail.google.com/) | [Maps](http://maps.google.com/) | [Docs](http://docs.google.com/) | [Sites](http://sites.google.com/) | [Books](http://books.google.com/) | [Blogger](http://blogger.com/) | [Calendar](http://calendar.google.com/) | [Google+](https://plus.google.com/) | [Photos](https://photos.google.com/) |
| Search | Gmail | Maps | Docs | Sites | Books | Blogger | Calendar | Google+ | Photos |

Now that the Closure Library is open source, more and more developers outside Google are integrating the library in their own projects.

## What can the Closure Library do for me?

If you are developing a large or growing application, you may benefit from the Closure Library's breadth. A well-tested library can insulate you from cross-browser compatibility issues and the minutiae of client-side programming, letting you focus on the fun stuff.

## Does AngularJS use the jQuery library?

Yes, AngularJS can use [jQuery](http://jquery.com/) if it's present in your app when the application is being bootstrapped. If jQuery is not present in your script path, AngularJS falls back to its own implementation of the subset of jQuery that we call [jQLite](https://docs.angularjs.org/api/ng/function/angular.element).

AngularJS 1.3 only supports jQuery 2.1 or above. jQuery 1.7 and newer might work correctly with AngularJS but we don't guarantee that.

## How can be the permission denied errors handled with AngularJS? Explain with example

## Can we have nested controllers in AngularJS? In the case of nested controllers, does the $scope object is shared across all controllers?

Yes.

## With which methods you can bootstrap your angular app for multiple modules? Explain the methods in detail.

## AngularJS <script> Tag

This example shows the recommended path for integrating AngularJS with what we call automatic initialization.

<!doctype html>

<html xmlns:ng="http://angularjs.org" ng-app>

<body>

...

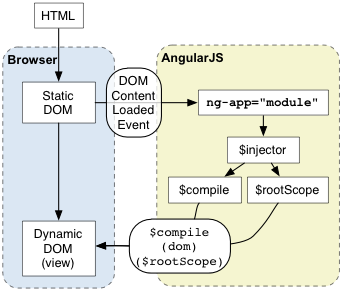
<script src="angular.js"></script>

</body>

</html>

1. Place the script tag at the bottom of the page. Placing script tags at the end of the page improves app load time because the HTML loading is not blocked by loading of the angular.js script. You can get the latest bits from [http://code.angularjs.org](http://code.angularjs.org/). Please don't link your production code to this URL, as it will expose a security hole on your site. For experimental development linking to our site is fine.
   * Choose: angular-[version].js for a human-readable file, suitable for development and debugging.
   * Choose: angular-[version].min.js for a compressed and obfuscated file, suitable for use in production.
2. Place ng-app to the root of your application, typically on the <html> tag if you want AngularJS to auto-bootstrap your application.
3. If you choose to use the old style directive syntax ng: then include xml-namespace in html to make IE happy. (This is here for historical reasons, and we no longer recommend use of ng:.)

## Automatic Initialization



AngularJS initializes automatically upon DOMContentLoaded event or when the angular.js script is evaluated if at that time document.readyState is set to 'complete'. At this point AngularJS looks for the [ngApp](https://docs.angularjs.org/api/ng/directive/ngApp) directive which designates your application root. If the [ngApp](https://docs.angularjs.org/api/ng/directive/ngApp) directive is found then AngularJS will:

* load the [module](https://docs.angularjs.org/guide/module) associated with the directive.
* create the application [injector](https://docs.angularjs.org/api/auto/service/$injector)
* compile the DOM treating the [ngApp](https://docs.angularjs.org/api/ng/directive/ngApp) directive as the root of the compilation. This allows you to tell it to treat only a portion of the DOM as an AngularJS application.

<!doctype html>

<html ng-app="optionalModuleName">

<body>

I can add: {{ 1+2 }}.

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

</body>

</html>

As a best practice, consider adding an ng-strict-di directive on the same element as ng-app:

<!doctype html>

<html ng-app="optionalModuleName" ng-strict-di>

<body>

I can add: {{ 1+2 }}.

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

</body>

</html>

This will ensure that all services in your application are properly annotated. See the [dependency injection strict mode](https://docs.angularjs.org/guide/di#using-strict-dependency-injection) docs for more.

## Manual Initialization

If you need to have more control over the initialization process, you can use a manual bootstrapping method instead. Examples of when you'd need to do this include using script loaders or the need to perform an operation before AngularJS compiles a page.

Here is an example of manually initializing AngularJS:

<!doctype html>

<html>

<body>

<div ng-controller="MyController">

Hello {{greetMe}}!

</div>

<script src="http://code.angularjs.org/snapshot/angular.js"></script>

<script>

angular.module('myApp', [])

.controller('MyController', ['$scope', function ($scope) {

$scope.greetMe = 'World';

}]);

angular.element(function() {

angular.bootstrap(document, ['myApp']);

});

</script>

</body>

</html>

Note that we provided the name of our application module to be loaded into the injector as the second parameter of the [angular.bootstrap](https://docs.angularjs.org/api/ng/function/angular.bootstrap) function. Notice that angular.bootstrap will not create modules on the fly. You must create any custom [modules](https://docs.angularjs.org/guide/module) before you pass them as a parameter.

You should call angular.bootstrap() after you've loaded or defined your modules. You cannot add controllers, services, directives, etc after an application bootstraps.

**Note:** You should not use the ng-app directive when manually bootstrapping your app.

This is the sequence that your code should follow:

1. After the page and all of the code is loaded, find the root element of your AngularJS application, which is typically the root of the document.
2. Call [angular.bootstrap](https://docs.angularjs.org/api/ng/function/angular.bootstrap) to [compile](https://docs.angularjs.org/guide/compiler) the element into an executable, bi-directionally bound application.

## Things to keep in mind

There a few things to keep in mind regardless of automatic or manual bootstrapping:

* While it's possible to bootstrap more than one AngularJS application per page, we don't actively test against this scenario. It's possible that you'll run into problems, especially with complex apps, so caution is advised.
* Do not bootstrap your app on an element with a directive that uses [transclusion](https://docs.angularjs.org/api/ng/service/$compile#transclusion), such as [ngIf](https://docs.angularjs.org/api/ng/directive/ngIf), [ngInclude](https://docs.angularjs.org/api/ng/directive/ngInclude) and [ngView](https://docs.angularjs.org/api/ngRoute/directive/ngView). Doing this misplaces the app [$rootElement](https://docs.angularjs.org/api/ng/service/$rootElement) and the app's [injector](https://docs.angularjs.org/api/auto/service/$injector), causing animations to stop working and making the injector inaccessible from outside the app.

## Deferred Bootstrap

This feature enables tools like [Batarang](https://github.com/angular/angularjs-batarang) and test runners to hook into angular's bootstrap process and sneak in more modules into the DI registry which can replace or augment DI services for the purpose of instrumentation or mocking out heavy dependencies.

If window.name contains prefix NG\_DEFER\_BOOTSTRAP! when [angular.bootstrap](https://docs.angularjs.org/api/ng/function/angular.bootstrap) is called, the bootstrap process will be paused until angular.resumeBootstrap() is called.

angular.resumeBootstrap() takes an optional array of modules that should be added to the original list of modules that the app was about to be bootstrapped with.

By default, an Angular application is initialized automatically by loading the **angular.js** script and placing the **ng-app** directive to the root element of your application (e.g. <html> tag), like the following:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <!doctype html>  <html ng-app>  <body>    ...    <script src="angular.js"></script>  </body>  </html> |

[Explore Angular Courses](https://appendto.com/courses/angular-js/)

Alternatively, you can also initialize your application manually if you need, using the **angular.bootstrap()** function. This is especially useful when you want your application to initialize only after a certain period or bound to another event taking place. See [this page](https://docs.angularjs.org/guide/bootstrap) for further details about the initialization methods in Angular.

## How to Use angular.bootstrap

**angular.bootstrap** is a function component in the core **ng** module that is used for starting up the Angular application manually, which gives you more control over how you initialize your application. The syntax for angular.bootstrap is as follows:



|  |  |
| --- | --- |
| 1 | angular.bootstrap(element, [modules], [config]); |

**element** is a DOM element (e.g. document) that is the root of the Angular application, **modules** (optional) is an array of modules to be loaded and **config** (optional) is an object used for configuration options. You can check [this page](https://docs.angularjs.org/api/ng/function/angular.bootstrap) on the official developer guide to read more about angular.bootstrap syntax.

## Example 1: Manually Bootstrap Your Angular Application

In the following example, we manually bootstrap our application which displays a welcome message once the user logs in.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | <!doctype html>  <html>  <body>    <div ng-controller="LoginMessage">{{message}}</div>    <script src="angular.js"></script>    <script>    angular.module('example', [])    .controller('LoginMessage', function($scope) {      $scope.message = 'Welcome to Members Area!';    });    angular.bootstrap(document, ['example']);    </script>  </body>  </html> |

Note how we selected **document** as the root of the application and how we passed the **example** module to the angular.bootstrap() function.

## Example 2: Bootstrap by Selecting a Specific Element

In the following example, we will display the current year in our application and bootstrap it by selecting a specific element as the application root.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | <!doctype html>  <html>  <body>    <div id="year" ng-controller="Year">{{CurrentYear}}</div>    <script src="angular.js"></script>    <script>    angular.module('example', [])    .controller('Year', function($scope) {      $scope.CurrentYear = new Date().getFullYear();    });    angular.bootstrap(document.getElementById('year'), ['example']);    </script>  </body>  </html> |

We selected the **div** element with the id **#year** as the root of our application.

## Example 3: Bootstrap Multiple Applications

[Explore Angular Courses](https://appendto.com/courses/angular-js/)

Using the angular.bootstrap function, you can bootstrap multiple angular applications on a single page. Let’s say we have two applications (**app1** and **app2**) defined by two div elements:



|  |  |
| --- | --- |
| 1  2 | <div id="app1">...</div>  <div id="app2">...</div> |

We use the following code to start up these two applications:



|  |  |
| --- | --- |
| 1  2  3  4  5 | angular.module('app1', []);  angular.module('app2', []);    angular.bootstrap(document.getElementById('app1'), ['app1']);  angular.bootstrap(document.getElementById('app2'), ['app2']); |

Before ending this tutorial, we will list a number of things you should pay attention while manually starting your applications using angular.bootstrap.

* If you use angular.bootstrap to start your application, you shouldn’t use the ng-app directive.
* You need to create the custom modules before passing them as a parameter to angular.bootstrap().
* When using multiple bootstraps on the same page, use them cautiously.

# How AngularJS compilation is different from other JavaScript frameworks?

AngularJS isn’t different from other frameworks, it just sets a few conventions and couple of things based on conventions, so you can spend less time for compilation of your code.

[From the docs:](http://docs.angularjs.org/guide/compiler)

**Compiler**

Compiler is an angular service which traverses the DOM looking for attributes. The compilation process happens into two phases.

1. **Compile:** traverse the DOM and collect all of the directives. The result is a linking function.
2. **Link:** combine the directives with a scope and produce a live view. Any changes in the scope model are reflected in the view, and any user interactions with the view are reflected in the scope model. Making the scope model a single source of truth.

Some directives such ng-repeat clone DOM elements once for each item in collection. Having a compile and link phase improves performance since the cloned template only needs to be compiled once, and then linked once for each clone instance.

AngularJS compilation process takes place in the Web Browser. No Server side or pre-compilation step is involved. Angular uses $compiler service to compile your Angular HTML page. The Angular compilation process begins after your HTML page (static DOM) is fully loaded. It happens in two phases.

How does Angular JS know when to perform dirty checking and update DOM output?

Angular constantly update the digest function so this won't match the source, but the general idea is still the same

{

// Versions

"angular": "1.2.20"

}

AngularJS implements dirty checking for two way data binding on $scope variables. Unlike dynamically setting up setters and getters, which is how Ember.js does two way data binding, dirty checking allows Angular to watch for variables that may or may not exist.

### $scope.$watch

|  |  |
| --- | --- |
| 1 | $scope.$watch( watchExp, listener, objectEquality ); |

To watch when a variable changes, you will use the $scope.$watch function. With this you give three arguments, what to watch (watchExp), what to do when it’s updated (listener), and whether or not you’re checking on a variable or on an object. As we are checking a variable, we can ommit this when we call the function. For example -

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $scope.name = 'Ryan';  $scope.$watch( function( ) {  return $scope.name;  }, function( newValue, oldValue ) {  console.log('$scope.name was updated!');  } ); |

Angular will register your watcher function in the $scope. You can see that these are registered by logging the $scope to the console. [I’ve created a test directive on jsFiddle to demonstrate this.](http://jsfiddle.net/ryanclark/SraRB/2/)

You’ll notice that the console logs the fact that $scope.name is updated - this is because $scope.name was previously undefined and we’ve updated it to equal Ryan!

You can also use a string instead of a function in $watch. This will do exactly the same as providing a function. In the Angular source code, if you provide a string, the following code is ran -

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | if (typeof watchExp == 'string' && get.constant) {  var originalFn = watcher.fn;  watcher.fn = function(newVal, oldVal, scope) {  originalFn.call(this, newVal, oldVal, scope);  arrayRemove(array, watcher);  };  } |

This will set our watchExp to a function, in which it will call our listener with the variable that you’ve given the name of.

### $$watchers

The $$watchers variable in $scope holds all of the watchers that you define. If you look into $$watchers in the jsFiddle, you’ll see that is an array of objects.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $$watchers = [  {  eq: false, // whether or not we are checking for objectEquality  fn: function( newValue, oldValue ) {}, // this is the listener function we've provided  last: 'Ryan', // the last known value for the variable  exp: function(){}, // this is the watchExp function we provided  get: function(){} // Angular's compiled watchExp function  }  ]; |

The $watch function returns the deregisterWatch function. This means that if we were to assign the initial $scope.$watch to a variable, we could just call it to stop watching. [View this in jsFiddle.](http://jsfiddle.net/ryanclark/SraRB/4/) Make sure you open and look at the first $scope that is logged before clicking on remove watcher!

However, [take a look at this.](http://jsfiddle.net/ryanclark/SraRB/5/) If we were to remove the watcher before the controller function is evaluated, there is no log that we updated the $scope.name variable, even though we have - why is this?

### $scope.$apply

Whenever a controller/directive/etc is ran in Angular, internally Angular runs a function called $scope.$apply. The $apply function will run a function given to it, before finally running the $digest function in the rootScope. More on digests later.

The Angular $apply function looks like this -

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | $apply: function(expr) {  try {  beginPhase('$apply');  return this.$eval(expr);  } catch (e) {  $exceptionHandler(e);  } finally {  clearPhase();  try {  $rootScope.$digest();  } catch (e) {  $exceptionHandler(e);  throw e;  }  }  } |

#### expr

The expr argument is just a function that you or Angular would pass through when calling $scope.$apply - most of the time you won’t even need to use $apply, let alone give it a function!

Let’s look into how ng-keydown uses $scope.$apply. To register the directive, Angular uses the following code -

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | var ngEventDirectives = {};  forEach(  'click dblclick mousedown mouseup mouseover mouseout mousemove mouseenter mouseleave keydown keyup keypress submit focus blur copy cut paste'.split(' '),  function(name) {  var directiveName = directiveNormalize('ng-' + name);  ngEventDirectives[directiveName] = ['$parse', function($parse) {  return {  compile: function($element, attr) {  var fn = $parse(attr[directiveName]);  return function ngEventHandler(scope, element) {  element.on(lowercase(name), function(event) {  scope.$apply(function() {  fn(scope, {$event:event});  });  });  };  }  };  }];  }  ); |

What this does is loop through the different types of events that can be fired and create a new directive called ng(EventNameHere). In the compile function of the directive, it registers an event handler on the element, with the event being the directives name respectively. When that event is fired, Angular runs scope.$apply, giving it a function to run as well.

### This is only one way data binding?

This will update the $scope value with the elements value - this is only one way data binding. This is because we’ve called ng-keydown, only alerting us when the keydown event is fired, and giving us the new value!

### But we want two way data binding!

Let’s take a look at ng-model. When you use ng-model, this allows you to do two way data binding - exactly what we want. ng-model uses both $scope.$watch (view to model) and $scope.$apply (model to view) to offer this.

ng-model will attach the event handler directive (such as keydown) to the input you’ve applied it to - this is where $scope.$apply is called! $scope.$watch is called in the directive’s controller. You can see this here -

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | $scope.$watch(function ngModelWatch() {  var value = ngModelGet($scope);  // if scope model value and ngModel value are out of sync  if (ctrl.$modelValue !== value) {  var formatters = ctrl.$formatters,  idx = formatters.length;  ctrl.$modelValue = value;  while(idx--) {  value = formatters[idx](value);  }  if (ctrl.$viewValue !== value) {  ctrl.$viewValue = value;  ctrl.$render();  }  }  return value;  }); |

When you call $scope.$watch with only one argument, the function you provide it will be called regardless of what updates - perfect! The function that is given in ng-model checks if the model and view are out of sync, and if it is, it will update the model with its new value. The function returns the new value, so when it is ran in the $digest function, we will know what the new value is!

### So why does our listener not fire?

If we look back at the jsFiddle where we deregister the $scope.$watch function in the same function as we define it, we can now understand why we don’t get notified about us updating $scope.name even though we did.

As mentioned earlier, Angular runs $scope.$apply on every directive’s controller function. If we look into the $scope.$apply function, it only runs the $digest after the directive’s controller function has been evaluated - meaning that the $scope.$watch function never actually gets a chance to be called, as we’ve deregisted it before it could’ve been ran! But how is it ran?

### $digest

The $digest function is called on the $rootScope by $scope.$apply. This will run the digest cycle on the $rootScope and will then traverse down the scopes and run the digest cycle on that. In simple terms, the digest cycle will fire all of our watchExp functions in the $$watchers variable, compare them against the last known value, and if they’re different, fire the listener!

When the digest cycle runs, it loops through the watchers and then loops again, whilst the cycle is considered “dirty”. The cycle is considered dirty when the watchExp and last known value aren’t equal to each other. Ideally this will run once, but if it runs more than 10 times you will get an error.

So when $scope.$apply is ran, $digest is ran, it will then loop through the $$watchers and fire any listener event if the watchExp does not equal the last known value. $scope.$apply is ran by Angular on anything that could possibly contain a model value changing. This is why when you update the $scope outside of Angular, for instance in a setTimeout function, you need to run $scope.$apply(); in order to have Angular notice that the scope has been updated!

### Let’s create our own

We’ll create a small, basic version of dirty checking that we can use. Angular’s dirty checking is a bit more advanced, offering async queues and some other neat things.

#### Setup our scope

Scope will just be a function, containing any data that we wish to store in it. We’ll extend the prototype object on the function to replicate $digest and $watch. We don’t need $apply as we’ll be won’t need to evaluate any functions in the context of the Scope - we’ll just simply use $digest. Our Scope will look like this -

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var Scope = function( ) {  this.$$watchers = [];  };  Scope.prototype.$watch = function( ) {    };  Scope.prototype.$digest = function( ) {    }; |

Our $watch function needs to accept two parameters, watchExp and listener. When $watch is called, we’ll push these into the $$watcher value we’ve set in Scope.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | var Scope = function( ) {  this.$$watchers = [];  };  Scope.prototype.$watch = function( watchExp, listener ) {  this.$$watchers.push( {  watchExp: watchExp,  listener: listener || function() {}  } );  };  Scope.prototype.$digest = function( ) {    }; |

You’ll notice that I’ve set listener to an empty function if there is no listener provided - this way we can register a $watch for all variables!

Next we will work on the $digest. We need to check if the old value is equal to the new value, and fire the listener if it isn’t. We will then loop until they are equal to each other. This is where the dirty variable comes in - whether or not the values are equal to each other!

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | var Scope = function( ) {  this.$$watchers = [];  };  Scope.prototype.$watch = function( watchExp, listener ) {  this.$$watchers.push( {  watchExp: watchExp,  listener: listener || function() {}  } );  };  Scope.prototype.$digest = function( ) {  var dirty;  do {  dirty = false;  for( var i = 0; i < this.$$watchers.length; i++ ) {  var newValue = this.$$watchers[i].watchExp(),  oldValue = this.$$watchers[i].last;  if( oldValue !== newValue ) {  this.$$watchers[i].listener(newValue, oldValue);  dirty = true;  this.$$watchers[i].last = newValue;  }  }  } while(dirty);  }; |

Next, we need to create a new instance of our scope. We’ll assign this to $scope. We can then register a watch function, and $digest it after we update it!

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | var Scope = function( ) {  this.$$watchers = [];  };  Scope.prototype.$watch = function( watchExp, listener ) {  this.$$watchers.push( {  watchExp: watchExp,  listener: listener || function() {}  } );  };  Scope.prototype.$digest = function( ) {  var dirty;  do {  dirty = false;  for( var i = 0; i < this.$$watchers.length; i++ ) {  var newValue = this.$$watchers[i].watchExp(),  oldValue = this.$$watchers[i].last;  if( oldValue !== newValue ) {  this.$$watchers[i].listener(newValue, oldValue);  dirty = true;  this.$$watchers[i].last = newValue;  }  }  } while(dirty);  };  var $scope = new Scope();  $scope.name = 'Ryan';  $scope.$watch(function(){  return $scope.name;  }, function( newValue, oldValue ) {  console.log(newValue, oldValue);  } );  $scope.$digest(); |

Success - we have dirty checking (in it’s most basic form) implemented! [Check out the jsFiddle](http://jsfiddle.net/ryanclark/PVQts/) to mess around with what we’ve made. If you look at the console, you’ll notice it logs

|  |  |
| --- | --- |
| 1 | Ryan undefined |

Which is the exact behaviour we want - $scope.name was previously undefined and we’ve set it to Ryan - result!

Let’s attach our $digest function to a keyup event on an input. That way we don’t have to call it ourselves. This means we can have two way data binding too!

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | var Scope = function( ) {  this.$$watchers = [];  };  Scope.prototype.$watch = function( watchExp, listener ) {  this.$$watchers.push( {  watchExp: watchExp,  listener: listener || function() {}  } );  };  Scope.prototype.$digest = function( ) {  var dirty;  do {  dirty = false;  for( var i = 0; i < this.$$watchers.length; i++ ) {  var newValue = this.$$watchers[i].watchExp(),  oldValue = this.$$watchers[i].last;  if( oldValue !== newValue ) {  this.$$watchers[i].listener(newValue, oldValue);  dirty = true;  this.$$watchers[i].last = newValue;  }  }  } while(dirty);  };  var $scope = new Scope();  $scope.name = 'Ryan';  var element = document.querySelectorAll('input');  element[0].onkeyup = function() {  $scope.name = element[0].value;  $scope.$digest();  };  $scope.$watch(function(){  return $scope.name;  }, function( newValue, oldValue ) {  console.log('Input value updated - it is now ' + newValue);    element[0].value = $scope.name;  } );  var updateScopeValue = function updateScopeValue( ) {  $scope.name = 'Bob';  $scope.$digest();  }; |

Perfect - using this approach we can now update the input’s value and it will be reflected in $scope.name, as well as call updateScopeValue and have the input’s value reflect that!

How can we dynamically create forms in Angular?

Angular provides us a lot of tools to deal with forms. Just the fundamental way it binds data for us to a lot of the tools it provides like validation, Angular helps make lives easier.

In the past, we have gone over how to [validate basic forms](https://scotch.io/tutorials/javascript/angularjs-form-validation) using Angular. This was a simple process and just required adding a few directives here and there.

## [#Dynamically Building Forms](https://scotch.io/tutorials/building-dynamic-angular-forms-with-ngrepeat-and-ngform#dynamically-building-forms)

Now let’s talk about a different scenario where we won’t always know how many fields or which fields we will have. There could be many scenarios for needing a dynamically built form. Let’s say that our application pulls data from that API for a form that we need to edit.

For this example, we are going to get a list of users and we need to edit their email. The process for this would look something like:

* Get an array of users from an API call
* Display this list to our administrator
* Provide fields to edit the information of our users
* Validate those fields using Angular

To keep things simple, instead of pulling a list of users from an API, we will be creating the array right in the JavaScript file. Easy cheese. Here is our list of users and we’ll be editing their email addresses.

Here is the example JSON data we would get back from our API for our users.

// sample data we would get back from an api

var users = [

{

name: 'Chris',

email: ''

},

{

name: 'Holly',

email: ''

}

];

// assign this data to an object to store all our form data

$scope.formData = {};

$scope.formData.users = users;

We will create a form using these fields and we will use ng-repeat to loop over each user and display their email field. Here is the code for that:

<!-- we are using bootstrap for these styling classes -->

<form name="userForm" novalidate>

<div class="form-group" ng-repeat="user in formData.users">

<label>{{ user.name }}'s Email</label>

<input type="text" class="form-control" name="email" ng-model="user.email" required>

<p class="help-block" ng-show="userForm.email.$invalid">Valid Email Address Required</p>

</div>

</form>

We loop over each of these fields but there is a problem when building our form this way. Validation won’t work because of how we named the name attribute statically.

Remember that we are using the ng-class and ng-show directives to show our errors. More info [**here**](https://scotch.io/tutorials/javascript/angularjs-form-validation#showing-an-error-message-ng-show).

## [#Validation Without ngForm](https://scotch.io/tutorials/building-dynamic-angular-forms-with-ngrepeat-and-ngform#validation-without-ngform)

By default, Angular currently does not allow us to dynamically create the name attribute of an input field. From our last [experiments with validation](https://scotch.io/tutorials/javascript/angularjs-form-validation), we can see that validating a field using Angular requires the name attribute.

**The Problem**: Name attributes are not dynamically generated and we are not able to validate fields individually. Since name isn’t dynamically generated, then all the fields will validate together as one. To check if an input is valid, you would use {{ userForm.email.$valid }}.

<div class="container" ng-app="dynamicFormApp" ng-controller="mainController">

<div class="page-header">

<h3 class="text-danger">Non Dynamic Way</h3>

<p>All fields need to be filled in for the error to go away. It treats all of the fields as one input.</p>

</div>

<form name="userFormOne" novalidate>

<div class="form-group" ng-repeat="person in formDataOne.people" ng-class="{ 'has-error' : userFormOne.email.$invalid }">

<label>{{ person.name }}'s Email</label>

<input type="email" class="form-control" name="email" ng-model="person.email" required>

<p class="help-block" ng-show="userFormOne.email.$invalid">Valid Email Required</p>

</div>

</form>

<pre><code>

{{ formDataOne }}

</code></pre>

</div>

Script.js

angular.module('dynamicFormApp', [])

.controller('mainController', function($scope) {

var people = [

{

name: 'Chris',

email: ''

},

{

name: 'Holly',

email: ''

}

];

$scope.formDataOne = {};

$scope.formDataOne.people = people;

});

Typing into the form above, you can see that the fields are only valid when both become valid email addresses.

## [#Validation Using ngForm](https://scotch.io/tutorials/building-dynamic-angular-forms-with-ngrepeat-and-ngform#validation-using-ngform)

So how would we validate each field individually, the way a user would expect our form to validate? Since Angular uses the method of formName.fieldName.$valid to validate, we will need to have each input be part of its own form.

**The Solution**: ngForm allows us to create forms within our main form that will allow us to validate fields individually. Let’s modify the code from earlier to accommodate ng-form and see how that changes things.

<form name="userForm" novalidate>

<div class="form-group" ng-repeat="user in formData.users" ng-class="{ 'has-error' : userFieldForm.email.$invalid }">

<ng-form name="userFieldForm">

<label>{{ user.name }}'s Email</label>

<input type="text" class="form-control" name="email" ng-model="user.email" required>

<p class="help-block" ng-show="userFieldForm.email.$invalid">Valid Email Address Required</p>

</ng-form>

</div>

</form>

Just by adding the ng-form directive, now each field will believe it is part of a form within our main form. In this case, **userFieldForm**. Now we are able to validate each field individually by using userFieldForm.email.$invalid.

<div class="container" ng-app="dynamicFormApp" ng-controller="mainController">  
   
 <div class="page-header">  
 <h3 class="text-success">Dynamic Way</h3>  
 <p>Each field acts as its own input and validates by itself... like it should.</p>  
 </div>  
   
 <form name="userFormTwo" novalidate>  
   
 <div class="form-group" ng-repeat="user in formDataTwo.users" ng-class="{ 'has-error' : userFieldForm.email.$invalid }">  
   
 <ng-form name="userFieldForm">  
 <label>{{ user.name }}'s Email</label>  
 <input type="email" class="form-control" name="email" ng-model="user.email" required>  
 <p class="help-block" ng-show="userFieldForm.email.$invalid">Valid Email Required</p>  
 </ng-form>  
   
 </div>  
   
 </form>  
   
 <pre><code>  
 {{ formDataTwo }}  
 </code></pre>  
   
</div>

angular.module('dynamicFormApp', [])  
  
.controller('mainController', function($scope) {  
 var users = [  
 {   
 name: 'Chris',  
 email: ''  
 },  
 {  
 name: 'Holly',  
 email: ''  
 }  
 ];   
   
 $scope.formDataTwo = {};  
 $scope.formDataTwo.users = users;  
   
});

Now we are able to validate each field individually!

## [#Conclusion](https://scotch.io/tutorials/building-dynamic-angular-forms-with-ngrepeat-and-ngform#conclusion)

Hopefully this has helped show how the ngForm directive is to be used when building dynamic forms from ngRepeat.

The official [ngForm docs](https://docs.angularjs.org/api/ng/directive/ngForm) aren’t very elaborate so this should be a good starting point for anyone with the need for a form like this.

Go ahead and experiment further and go even crazier by nesting ngRepeats! As always, let us know if you have any questions or comments.

Recently we participated in a very large project: a new e-procurement system for the Russian government. Our team [worked on the Standards & References section](http://www.azoft.com/case-studies/eprocurement-software-development/), which is technically a database, and one of my specific tasks was to provide data visualization. As the project required creating a fair amount of similar pages, too time-consuming to do manually, I developed a dynamic form-generation [solution in AngularJS](http://www.azoft.com/angularjs-development/). Here was my approach.

## Solution overview

Most of the project’s pages offered the user the same function set – CRUD. For each new entity, the configuration of its fields was written into the form generator (by myself) and it populates new web pages with unique content along with the basic function set mentioned above.

## Solution benefits

Using dynamic forms provides automation with a number of obvious benefits:

* It saves time writing code;
* It reduces the amount of code (thus reducing the number of possible bugs);
* It unifies the layout of web pages (e.g. it keeps the alignment of buttons and fields constant when navigating).

Additionally, if one wants to keep this dynamic form-generation method enhanced and updated, the enhancements are automatically applied to all generated pages. Pure automation!

## Here’s an example

The simple example below illustrates my approach. This can serve as a good starting point for your own customized solution. The sample code generates New/Edit and View forms and JSON queries on servers using the corresponding input fields. Download demo [here](http://cases.azoft.com/creating-dynamic-forms-using-angularJS-demo).

Note: The demo will not run in the Chrome browser because it will not load cross-domain content. Nonetheless, it works great in practically all other major browsers.

### JS structure

form-generation-app.js

[view plaincopy to clipboardprint?](http://cases.azoft.com/creating-dynamic-web-forms-angularjs/)

1. angular.module('app', []).config(**function**() { });
3. form-generation-controller.js
5. angular.module('app').controller('FormGenerationController', **function** ($scope) {
7. // entity to edit
8. $scope.entity = {
9. name: 'Max',
10. country: 2,
11. licenceAgreement: true,
12. description: 'I use AngularJS'
13. };
15. // fields description of entity
16. $scope.fields = [
17. {
18. name: 'name',
19. title: 'Name',
20. required: true,
21. type: {
22. view: 'input'
23. }
24. },
25. {
26. name: 'country',
27. title: 'Country',
28. type: {
29. view: 'select',
30. options: [
31. {id: 0, name: 'USA'},
32. {id: 1, name: 'German'},
33. {id: 2, name: 'Russia'}
34. ]
35. }
36. },
37. {
38. name: 'licenceAgreement',
39. title: 'Licence Agreement',
40. type: {
41. view: 'checkbox'
42. }
43. },
44. {
45. name: 'description',
46. title: 'Description',
47. type: {
48. view: 'textarea'
49. }
50. }
51. ];
53. });

### HTML structure

index.html

[view plaincopy to clipboardprint?](http://cases.azoft.com/creating-dynamic-web-forms-angularjs/)

1. <body ng-app="app" ng-controller="FormGenerationController">
3. <h1>Edit/New mode:</h1>
4. <div ng-repeat="field in fields">
5. <ng-**include** src="'html/edit-field.html'"/>
6. </div>
8. <div>
9. <h1>Your entity JSON:</h1>
10. {{ entity }}
11. </div>
13. <h1>View mode:</h1>
14. <div ng-repeat="field in fields">
15. <ng-**include** src="'html/view-field.html'"/>
16. </div>
18. </body>

edit-field.html

[view plaincopy to clipboardprint?](http://cases.azoft.com/creating-dynamic-web-forms-angularjs/)

1. <!--also suitable **for** creation-->
2. <label>{{ field.title }}<em ng-hide="!field.required">\*</em> :</label>
3. <span ng-**switch** on="field.type.view">
4. <span ng-**switch**-when="input">
5. <input
6. ng-model="entity[field.name]"
7. type="text"
8. />
9. </span>
10. <span ng-**switch**-when="checkbox">
11. <input
12. ng-model="entity[field.name]"
13. ng-checked="entity[field.name]"
14. type="checkbox"
15. />
16. </span>
17. <span ng-**switch**-when="textarea">
18. <textarea
19. ng-model="entity[field.name]"
20. ></textarea>
21. </span>
22. <span ng-**switch**-when="select">
23. <select
24. ng-model="entity[field.name]"
25. ng-options="item.id as item.name for item in field.type.options"
26. ></select>
27. </span>
28. <!--You can add more custom widget types needed **for** your project-->
29. <span ng-**switch**-**default**>
30. <span>Unknown widget</span>
31. </span>
32. </span>

view-field.html

[view plaincopy to clipboardprint?](http://cases.azoft.com/creating-dynamic-web-forms-angularjs/)

1. <label>{{ field.title }}:</label>
2. <span ng-**switch** on="field.type.view">
3. <span ng-**switch**-when="input">
4. {{ entity[field.name] }}
5. </span>
6. <span ng-**switch**-when="checkbox">
7. <input
8. ng-model="entity[field.name]"
9. ng-checked="entity[field.name]"
10. ng-disabled="true"
11. type="checkbox"
12. />
13. </span>
14. <span ng-**switch**-when="textarea">
15. {{ entity[field.name] }}
16. </span>
17. <span ng-**switch**-when="select">
18. {{ field.type.options[entity[field.name]].name }}
19. </span>
20. <!--You can add more custom widget types needed **for** your project-->
21. <span ng-**switch**-**default**>
22. <span>Unknown widget</span>
23. </span>
24. </span>

Which means of communication between modules of your application are easily testable?

Services

With more than one ng-app in an HTML document (an HTML page), are they automatically initialized? Describe the AngularJS application initialization process with multiple ng-app in an HTML document/page.

**Ans**: Only one AngularJS application can be auto-bootstrapped. The first ‘ng-app’ 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, one must manually bootstrap them using angular bootstrap service.

How to do Language Internationalization in AnglarJS?

Through i18n and i10n

What is the difference between Angular and Polymer? What problems does Polymer solve that AngularJS has not or will not? Are there plans to tie Polymer in with AngularJS in the future?

You're not the first to ask this question :) Let me clarify a couple of things before getting to your questions.

1. Polymer's webcomponents.js is a library that contains several polyfills for various W3C APIs that fall under the Web Components umbrella. These are:
   * Custom Elements
   * HTML Imports
   * <template>
   * Shadow DOM
   * Pointer Events
   * others

The left-nav in the documentation ([polymer-project.org](http://www.polymer-project.org/)) has a page for all of these "Platform technologies". Each of those pages also has a pointer to the individual polyfill.

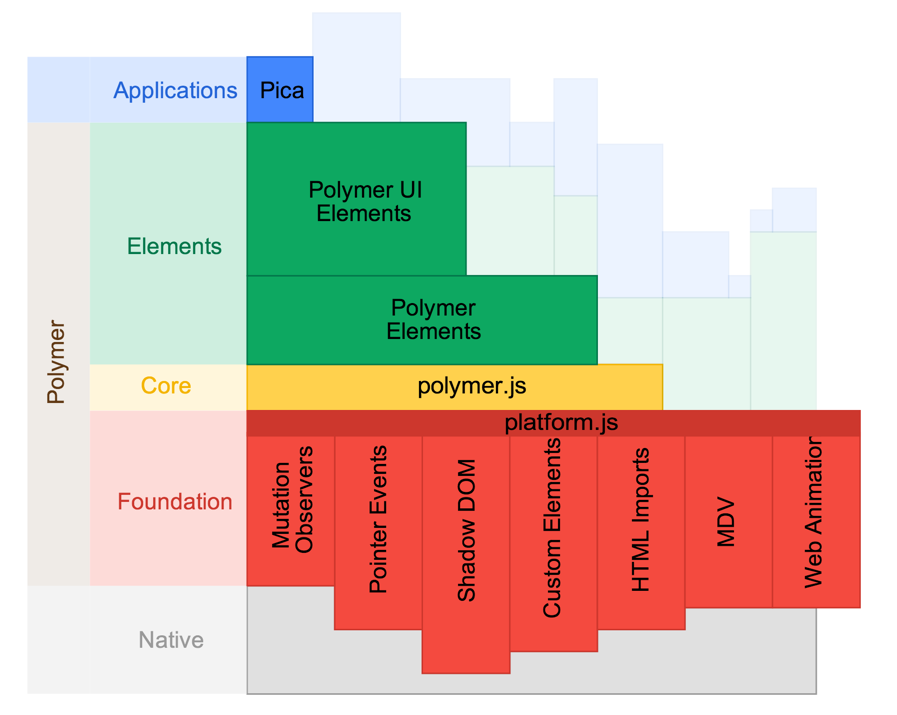
1. <link rel="import" href="x-foo.html"> is an HTML Import. Imports are a useful tool for including HTML in other HTML. You can include <script>, <link>, markup, or whatever else in an import.
2. Nothing "links" <x-foo> to x-foo.html. In your example, it's assumed the Custom Element definition of <x-foo> (e.g. <element name="x-foo">) is defined in x-foo.html. When the browser sees that definition, it's registered as a new element.

On to questions!

**What is the difference between Angular and Polymer?**

We covered some of this in our [Q&A video](http://www.youtube.com/watch?v=irGDN5Ysi_A&t=21m45s). In general, Polymer is a library that aims to use (and show how to use) Web Components. Its foundation is Custom Elements (e.g. everything you build is a web component) and it evolves as the web evolves. To that end, we only support the latest version of the modern browsers.

I'll use this image to describe Polymer's entire architecture stack:



RED layer: We get tomorrow's web through a set of polyfills. Keep in mind, those libraries go away over time as browsers adopt the new APIs.

YELLOW layer: Sprinkle in some sugar with polymer.js. This layer is our opinion on how to use the spec'd APIs, together. It also adds things like data-binding, syntatic sugar, change watchers, published properties...We think these things are helpful for building web component-based apps.

GREEN: The comprehensive set of UI components (green layer) is still in progress. These will be web components that use all of the red + yellow layers.

**Angular directives vs. Custom Elements?**

See Alex Russell's [answer](https://groups.google.com/forum/?fromgroups=#!searchin/polymer-dev/angular/polymer-dev/s761szb9WJc/Lb29XuoJdqAJ). Basically, Shadow DOM allows composing bits of HTML but also is a tool for encapsulating that HTML. This is fundamentally a new concept on the web and something other frameworks will leverage.

**What problems does Polymer solve that AngularJS has not or will not?**

Similarities: declarative templates, data binding.

Differences: Angular has high level APIs for services, filters, animations, etc., supports IE8, and at this point, is a much more robust framework for building production apps. Polymer is just starting out in alpha.

**Are there plans to tie Polymer in with AngularJS in the future?**

They're [separate projects](http://www.polymer-project.org/resources/faq.html#frameworks). That said, both the Angular and Ember teams [announced](https://groups.google.com/forum/#!msg/polymer-dev/4RSYaKmbtEk/uYnY3900wpIJ) they'll eventually move to using the underlying platform APIs in their own frameworks.

^ This is a huge win IMO. In a world where web developers have powerful tools (Shadow DOM, Custom Elements), framework authors also can utilize these primitives to create better frameworks. Most of them currently go through great hoops to "get the job done".

A year after pulling back the curtains on the Polymer Project, Google’s taken part of [I/O 2014](http://www.binpress.com/blog/2014/06/26/what-developers-need-to-know-google-io-2014/) to demo what the code can do. Right on cue, the web is abuzz with discussion comparing Polymer to other popular web frameworks, and it’s no surprise since it bears some resemblance to Google’s very own [AngularJS](https://angularjs.org/). Join me after the break as I introduce Polymer and explore just how it relates to Angular.

## What is Polymer?

Polymer is a library for creating [Web Components](http://css-tricks.com/modular-future-web-components/), which are a set of W3C standards and upcoming browser APIs for defining your own custom HTML elements. With the help of polyfills and sugar, it can create these custom elements and bring Web Component support to browsers that don’t play nice with the standard just yet.

Custom elements? Huh? They look like this:

<google-map lat="37.790" long="-122.390"></google-map>

Seem familiar? They’re are very similar to Angular directives. As you’d expect, the result would be a Google Map plugged directly into your webpage.

## So, how does Polymer differ from Angular?

Angular is a complete framework for building webapps, whereas Polymer is a library for creating Web Components. Those components, however, can then be used to build a webapp.

Angular has high-level APIs for things like services, routing, server communication and the like. Polymer, on the other hand, doesn’t provide these things except as separate web components from their core library. Instead, it focuses on allowing you to create rich, powerful, reusable web components, which could be used to build webapps like those built with Angular. In the future, the lines could be blurred further as frameworks like Angular may leverage Web Components.

Even though Angular and Polymer aim to do different things, there is currently some overlap. Web components and Angular’s element directives are very similar, and if there’s a comparison to be made it should be between Polymer’s Custom Elements and Angular’s directives.

## Angular directives vs. Custom Elements

Polymer (more specifically [Shadow DOM](http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom/)) provides the ability to compose encapsulated JS, CSS, and HTML as [Custom Elements](http://www.html5rocks.com/en/tutorials/webcomponents/customelements/), much like Angular element directives.

Angular directives are conceptually similar to Custom Elements but they are implemented without the use of the Web Components APIs. Angular directives are a way to build custom elements, but Polymer and the Web Components specification are the standards-based way to do it.

Similar to Angular, Polymer elements provide templating and bi-directional data binding. However, they also provide new functionality such as the Shadow DOM, which enables [encapsulation of CSS](http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom-201/). Angular directives don’t have any notion of style encapsulation, but Angular is expected incorporate that functionality eventually.

In the future, both will be used together. Because custom elements will be the DOM, they’ll work seamlessly with frameworks like Angular. The Angular team has said they will eventually use the underlying Web Components APIs (Custom Elements, Shadow DOM, etc.).

Below, I’ve put together a comparison of an element that renders a [Gravatar](https://en.gravatar.com/) written as a Polymer Custom Element and as an Angular directive.

**Both methods create a custom element usable like this:**

<user-gravatar email="name@example.com"></user-gravatar>

**Which when rendered or evaluated results in the following:**

<img src="https://gravatar.com/avatar/264aa0ecd52d37fc67b22eae24e13f5b" />

**Polymer Custom Element:**

<polymer-element name="user-gravatar" attributes="email">

<template>

<img src="https://secure.gravatar.com/avatar/{{gid}}" />

</template>

<script>

Polymer('user-gravatar', {

ready: function() {

this.gid = md5(this.email);

}

});

</script>

</polymer>

**Angular directive:**

app.directive('user-gravatar', ['md5', function() {

return {

restrict: 'E',

link: function(scope, element, attrs) {

scope.gid = md5(attrs.email);

},

template: '<img src="https://gravatar.com/avatar/{{gid}}" />'

};

}]);

As you can see, Polymer uses a declarative syntax (another element!) to create custom elements. Both accomplish the same thing but using different approaches.

## Can Angular and Polymer be used together?

Yes! You can use Polymer custom elements inside of an Angular app. Web components are just regular DOM elements, so they have attributes, emit events, and can contain child elements.

## What about the future of both projects?

Angular and Polymer will remain separate projects with their own goals. That said, Angular has [announced](https://groups.google.com/forum/#!msg/polymer-dev/4RSYaKmbtEk/uYnY3900wpIJ) they’ll eventually move to use the Web Components APIs in their underlying architecture.

For Angular 2.0, Web Components will work seamlessly within Angular apps and directives, and components written in Angular will export to Web Components to be used by Polymer or other libraries.

As the Web Components specification matures and browsers implement the various APIs, Polymer elements will utilize native browser functionality instead of  
polyfills. Eventually, once browser support has improved, Polymer’s polyfill layer can be gracefully removed.