**AngularJS**

|  |  |
| --- | --- |
| **Component of Angular** | **Description** |
| Module | Modules serve as containers to assist in organizing code within your AngularJs application. Modules can contain sub-modules. |
| Services | Services are a point where you can put common functionality to an AngularJs application. For example if you would like to share data with more than one controller then the best way is to promote that data to a service and then make it available via the service. Services extend controllers and make them more globally accessible. |
| Routes | Routes allow us to determine ways to navigate to specific states within our application. It also allows us to define configuration options for each specific route, such as which template and controller to use. |
| View | The view in AngularJs is what exists after AngularJs has compiled and rendered the DOM. It's a representation of whatever outcome from |
| @scope | $scope is essentially the “glue” between the view and controller within an AngularJs application. And somewhere supports two-way binding within the application. |
| Controller | The controller is responsible for defining methods and properties that the view can bind to and interact with. Controllers should be lightweight and only focus on the view they're controlling. |
| directive | A directive is an extension of a view in AngularJs in that it allows us to create custom, reusable elements. You can also consider directives as decorators for your HTML. Directives are used to extend views and to make these extensions available for use in more than one place. |
| Config | The config block of an AngularJs application allows for the configuration to be applied before the application actually runs. This is useful for setting up routes, dynamically configuring services and so on. |

**An Overview of the AngularJS Life Cycle**

Life cycle, which has three phases: bootstrap, compilation, and runtime. Understanding the life cycle of an AngularJS application.

**The Bootstrap Phase**

The first phase of the AngularJS life cycle is the bootstrap phase, which occurs when the AngularJS JavaScript library is downloaded to the browser. AngularJS initializes its own necessary components and then initializes your module, which the ng-app directive points to. The module is loaded, and any dependencies are injected into your module and made available to code within the module.

**The Compilation Phase**

The second phase of the AngularJS life cycle is the HTML compilation stage. Initially when a web page is loaded, a static form of the DOM is loaded in the browser. During the compilation phase, the static DOM is replaced with a dynamic DOM that represents the AngularJS view.

This phase involves two parts: traversing the static DOM and collecting all the directives and then linking the directives to the appropriate JavaScript functionality in the AngularJS built-in library or custom directive code. The directives are combined with a scope to produce the dynamic or live view.

**The Runtime Data Binding Phase**

The final phase of the AngularJS application is the runtime phase, which exists until the user reloads or navigates away from a web page. At that point, any changes in the scope are reflected in the view, and any changes in the view are directly updated in the scope, making the scope the single source of data for the view.

AngularJS behaves differently from traditional methods of binding data. Traditional methods combine a template with data received from the engine and then manipulate the DOM each time the data changes. AngularJS compiles the DOM only once and then links the compiled template as necessary, making it much more efficient than traditional methods.

**AngularJS Dirty Checking**

Dirty checking is a very simply process to check if the value of an expression/variable has changed. Its basically just comparing old value with a new value to see if it has changed. AngularJS uses dirty checking to see if a value of a expression/variable in it’s scope has changed or not, and if it has changed it does the required operation (updating DOM etc).

**$watch**

$watch is angular method, for dirty checking. Any variable or expression assigned in $scope automatically sets up a $watchExpression in angular. You can create a watch express yourself as well

|  |  |
| --- | --- |
| 1  2  3 | $scope.$watch('variable',function(newValue,oldValue){    }); |

So assigning a variable to $scope or using directives like ng-if, ng-show, ng-repeat etc all create watches in angular scope automatically. e.g $scope.text = ''; creates a $watch for ‘text’ automatically in angular.

**$digest**

$digest() is angular method, which is invoked internally by angularjs in frequent intervals. In $digest method, angular iterates overall $watches in its scope/child scoples.If any changes are found the resulting DOM operation is done.

**$apply**

$apply() is a angular method, internally invokes $digest. This method is used when you want to tell angular manually start dirty checking (execute all $watches). [Read More here on when to use $apply](http://excellencenodejsblog.com/angularjs-access-scope-from-outside-function/)

**$destroy**

$destory is both a method and event in angularjs. $destory() method, removes a scope and all its children from dirty checking. $destory event is called by angular when ever a $scope or $controller is destroyed.

|  |  |
| --- | --- |
| 1  2  3 | $scope.$on('$destory',function(){     //do clean up here  }); |

So we see the $digest and $watch are critical to the working on angularjs.  
There a very good explanation on stack overflow about watches and performance which is a worth a read

**Stop watcher watching**

$watch returns a deregistration function. Calling it would deregister the $watcher.

var listener = $scope.$watch("quartz", function () {});

// ...

listener(); // Would clear the watch

For expert-led online Angular training courses you can’t go past Ultimate Angular by Todd Motto. Try his courses [here](https://ultimateangular.com/), and use the code SITEPOINT to get 25% off and to help support SitePoint.

$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 scopemodel, 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.

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.

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 scopemodel? 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.

## How to avoid creating too many watchers

Each time your user interacts with your app, every single watcher in your app will be evaluated at least once. A big part of optimising an AngularJS app is reducing the number of watchers in your $scope tree. One easy way to do this is with one time binding.

If you have data which will rarely change, you can bind it only once using the :: syntax, like so:

<p>{{::person.username}}</p>

or

<p ng-bind="::person.username"></p>

The binding will only be triggered when the containing template is rendered and the data loaded into $scope.

This is especially important when you have an ng-repeat with many items.

<div ng-repeat="person in people track by username">

{{::person.username}}

</div>

## SERVICE VS FACTORY - ONCE AND FOR ALL

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.

**AngularJS Directive**

Directives are one of the most powerful components of AngularJS, helping you extend basic HTML elements/attributes and create reusable and testable code. In this tutorial, I will show you how to use AngularJS directives with real-life best practices.

What I mean here by directivesis mostly custom directives during the tutorial. I will not try to teach you how to use built-in directives like ng-repeat, ng-show, etc. I will show you how to use custom directives to create your own components.

In this widget there is a book image, title, description, comments, and rating. Collecting that information and putting in a specific dom element may be hard to do in every place you want to use it. Let's widgetize this view by using an AngularJS directive.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10 | angular.module('masteringAngularJsDirectives', [])  .directive('book', function() {      return {          restrict: 'E',          scope: {              data: '='          },          templateUrl: 'templates/book-widget.html'      }  }) |

A [directive](https://docs.angularjs.org/guide/directive) function has been used in the above example to create a directive first. The name of the directive is book**.**This directive returns an object, and let's talk a bit about this object. restrictis for defining the directive type, and it can be A(**A**ttribute),C(**C**lass), E (**E**lement), andM(co**M**ment). You can see the usage of each respectively below.

| **Type** | **Usage** |
| --- | --- |
| A | <div **book**></div> |
| C | <div class="**book**"></div> |
| E | <**book**data="book\_data"></**book**> |
| M | <!--directive:**book** --> |

scopeis for managing the directive scope. In the above case, book data is transferred to the directive template by using the "="scope type. I will talk about in detail about scope in the following sections. templateUrlis used for calling a view in order to render specific content by using data transferred to the directive scope. You can also use templateand provide HTML code directly, like this:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16 | .directive('booktest', function() {      return {          restrict: 'E',          scope: {              title: '@'          },          replace: true,          template: '<div>{{title}}</div>',          link: function(scope, element, attrs) {              element.bind("click", function() {                  console.log("book viewed!");                  scope.viewed = true;              });          }      }  }) |

I think article i wrote about $q might help you.

**Introduction to $q**

$q is an angular define a service. It’s same as new Promise(). But $q takes things to the next level by enhancing additional feature that developers can use to perform complex tasks more simply.

This is a sample for creating promise using $q

angular.module("app",[])

.controller("ctrl",function($scope,$q){

var work = "resolve";

var promise = $q(function(resolve, reject) {

if (work === "resolve") {

resolve('response 1!');

} else {

reject('Oops... something went wrong');

}

});

promise.then(function(data) {

alert(data)

})

})

**$q.defer()**

$q.defer() return the instance of the promise constructor. Once you create a defer object there are following methods and properties that you can access from that object

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.

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

See the example

angular.module("app",[])

.controller("ctrl",function($scope,$q){

var work = "resolve";

function getData(){

var obj = $q.defer();

if (work === "resolve") {

obj.resolve('response 1!');

} else {

obj.reject('Oops... something went wrong');

}

return obj.promise;

}

getData().then(function(data) {

alert(data)

})

})

**$q.all()**

If a user need to send multiple request one shot,then the user can use $q.all() service.

$q.all([$http.get('data1.json'),$http.get('data2.json')])

.then(function(response){

console.log(response[0].data) // data1.json response

console.log(response[1].data) // data1.json response

})

In here,there are two http request sent simultaneously to two separate json files to get data. The response comes as an array and response order is same as the http request order.