**AngularJS**

* AngularJS is a JavaScript framework made by Google for building complex client-side applications. AngularJS is an open source project licensed under the [MIT license](http://github.com/angular/angular.js/blob/master/LICENSE)
* Angular JS is like JQuery but JQuery is mainly focused on DOM manipulation and then update data whereas Angular updates data and then DOM gets updated.
* Angular is designed to extend HTML to make it the language you need for building complex web applications. Adding your own tags and attributes allows you to write simple HTML tags that do very complex things.
* Angular supports the entire development process, provides structure for your web apps, and works with the best JS libraries. With angular, the view and data model are always in sync — there is no need for manual DOM manipulation.
* Angular is small, weighing in at 60kb, is compatible with all modern browsers and works great with jQuery.

|  |  |
| --- | --- |
| Concept | Description |
| [Template](http://docs.angularjs.org/guide/concepts#template) | HTML with additional markup |
| [Directives](http://docs.angularjs.org/guide/concepts#directive) | extend HTML with custom attributes and elements |
| [Model](http://docs.angularjs.org/guide/concepts#model) | the data that is shown to the user and with which the user interacts |
| [Scope](http://docs.angularjs.org/guide/concepts#scope) | context where the model is stored so that controllers, directives and expressions can access it |
| [Expressions](http://docs.angularjs.org/guide/concepts#expression) | access variables and functions from the scope |
| [Compiler](http://docs.angularjs.org/guide/concepts#compiler) | parses the template and instantiates directives and expressions |
| [Filter](http://docs.angularjs.org/guide/concepts#filter) | formats the value of an expression for display to the user |
| [View](http://docs.angularjs.org/guide/concepts#view) | what the user sees (the DOM) |
| [Data Binding](http://docs.angularjs.org/guide/concepts#databinding) | sync data between the model and the view |
| [Controller](http://docs.angularjs.org/guide/concepts#controller) | the business logic behind views |
| [Dependency Injection](http://docs.angularjs.org/guide/concepts#di) | Creates and wires objects / functions |
| [Injector](http://docs.angularjs.org/guide/concepts#injector) | dependency injection container |
| [Module](http://docs.angularjs.org/guide/concepts#module) | configures the Injector |
| [Service](http://docs.angularjs.org/guide/concepts#service) | reusable business logic independent of views |

* Basics:

<html ng-app> The ng-app will cause Angular to auto initialize your application.

For IE7 <html ng-app id=”ng-app”>

**HTML compilation:**

var html = '<div ng-bind="exp"></div>';

var template = angular.element(html); // Step 1: parse HTML into DOM element

var linkFn = $compile(template); // Step 2: compile the template

var element = linkFn(scope); // Step 3: link the compiled template with the scope.

parent.appendChild(element); // Step 4: Append to DOM (optional)

* **Compiler**

Compiler is an angular service which traverses the DOM looking for attributes. The compilation process happens in two phases.

**Compile:** traverse the DOM and collect all of the directives. The result is a linking function.

**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. This makes the scope model the single source of truth.

**Link** means setting up listeners on the DOM and setting up $watch on the Scope to keep the two in sync.

* **Controllers** (ng-controller)

Controllers are the behavior behind the DOM elements.

Controllers are tied to particular html elements.

Don’t use controller for any kind of DOM manipulation.

Controllers should contain only business logic.

They contain data and functions that the html can interact with, and can interact with other service objects, which can handle things such as communicating with the server

<div ng-controller="TodoCtrl">

We define TodoCtrl in our JS file as below:

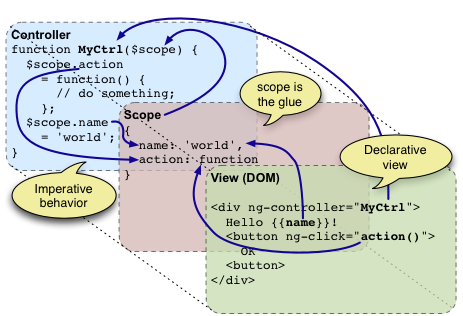
function TodoCtrl($scope) { // Where $scope contains model data

$scope.name=”Pravin”;

}

We can use the function as below:

$scope.saveData=function(){//write your logic here};



Its job is to construct the model and publish it to the view along with callback methods. The view is a projection of the scope onto the template (the HTML). The scope is the glue which marshals the model to the view and forwards the events to the controller.

The separation of the controller and the view is important because:

* The controller is written in JavaScript. JavaScript is imperative. Imperative is a good fit for specifying application behavior. The controller should not contain any rendering information (DOM references or HTML fragments).
* The view template is written in HTML. HTML is declarative. Declarative is a good fit for specifying UI. The View should not contain any behavior.
* Since the controller is unaware of the view, there could be many views for the same controller. This is important for re-skinning, device specific views (i.e. mobile vs desktop), and testability.
* **Databinding**: It’s a two-way binding. Data-binding is an automatic way of updating the view whenever the model changes, as well as updating the model whenever the view changes

<body ng-app>

<div>

<input type='text' ng-model='name' />

<h2>{{name}}</h2>

</div>

</body>

Where, ng-app indicates that everything within this tag will be treated as Angular application.

* **Services**:

Services are classes that can either contain **business logic** or **handle data**

All angular services are **singletons**.

It is important to realize that all Angular services are application singletons. This means that there is only one instance of a given service per injector.

*Syntax*: module.service( 'serviceName', function );

*Result*: When declaring serviceName as an injectable argument you will be provided with the **instance of a function** passed to module.service.

*Usage*: Could be useful for **sharing utility functions** that are useful to invoke by simply appending () to the injected function reference. Could also be run with injectedArg.call( this ) or similar.

**Factories**

Syntax: module.factory( 'factoryName', function );

Result: When declaring factoryName as an injectable argument you will be provided the **value that is returned by invoking the function reference** passed to module.factory.

Usage: Could be useful for returning a **'class'** function that can then be new'ed to create instances.

**Another way to inject in controller:**

* 1. ***Using $inject***

*function myController(scope, notifyService) {*

*scope.callNotify = function(msg) {*

*notifyService(msg);*

*};*

*}*

*myController****.$inject*** *= [‘$scope’,’myNotifyService’];*

* 1. ***Implicit***

*function myController($scope,$myNotifyService’)*

*{*

*// Do something*

*}*

* **Modules** (ng-module)

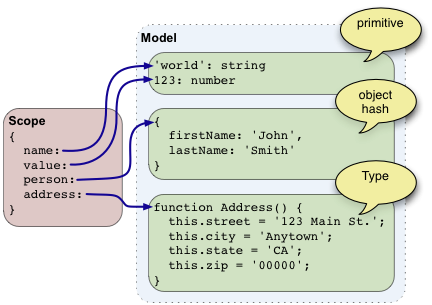
**Modules** are used to organize the objects in an AngularJS application

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

Where, 'myApp' is name of module and [] is array of other modules our module depends on.

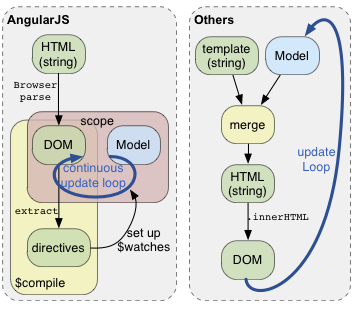
<body ng-app='myApp'>

* **Model**



The model is the data which is merged with the template to produce the view. To be able to render the model into the view, the model has to be able to be referenced from the scope. Unlike many other frameworks Angular makes no restrictions or requirements on the model. There are no classes to inherit from or special accessor methods for accessing or changing the model. The model can be primitive, object hash, or a full object Type. In short the model is a plain JavaScript object.

* **View**



The view is what the user sees. The view begins its life as a template, is merged with the model and finally rendered into the browser DOM. Angular takes a very different approach to rendering the view compared to most other templating systems.

Others - Most templating systems begin as an HTML string with special templating markup. Often the template markup breaks the HTML syntax which means that the template can not be edited by an HTML editor. The template string is then parsed by the template engine, and merged with the data. The result of the merge is an HTML string. The HTML string is then written to the browser using the innerHTML, which causes the browser to render the HTML. When the model changes the whole process needs to be repeated. The granularity of the template is the granularity of the DOM updates. The key here is that the templating system manipulates strings.

* **Directives**:

The compiler allows you to attach behavior to any HTML element or attribute and even create new HTML elements or attributes with custom behavior

A directive is a behavior which should be triggered when specific HTML constructs are encountered during the compilation process. The directives can be placed in element names, attributes, class names, as well as comments. E.g. ng-controller, ng-model, ng-click, ng-repeat etc.

***Best Practice:****Prefer using directives via tag name and attributes over comment and class names.*

### Custom Directive:

e.g. <div ng-controller=”ctrl”>

<div my-directive></div>

</div>

*Ctrl scope and myDirective scope can access each other’s properties and methods*

*If directive name is camel-case e.g. myDirective then in html use as my-directive*

### *Directive Definition Object:* It provides instructions to the compiler.

The attributes are:

* 1. **priotity**: default is 0. It is used to sort the directives before their compile functions get called
  2. **terminal**: If set to true then the current priority will be the last set of directives which will execute
  3. **scope**: default value is false.
     1. If set to ***true*,** then **a new scope** will be created for this directive. If multiple directives on the same element request a new scope, only one new scope is created. The new scope rule does not apply for the root of the template since the root of the template always gets a new scope.

*Because of new scope, Ctrl scope cannot access properties and methods of myDirective scope.*

* + 1. **If set to** {} **(object hash),** then **a new "isolate" scope** is created. The 'isolate' scope differs from normal scope in that it does not prototypically inherit from the parent scope.

This is useful when creating reusable components, which should not accidentally read or modify data in the parent scope.

But we can do some custom binding with parent scope.

Three ways to bind with parent scope:

1. @ or @attr - Text Binding
2. = or =attr - Two way binding
3. & or &attr – One way binding (using getter and setter method)
   1. **controller**: The controller is instantiated before the pre-linking phase and it is shared with other directives.

$scope, $element, $attrs and $transclude

* 1. **require:**

Require another directive and inject its controller as the fourth argument to the linking function. The require takes a string name (or array of strings) of the directive(s) to pass in.

* 1. **controllerAs**: Controller alias at the directive scope. An alias for the controller so it can be referenced at the directive template. The directive needs to define a scope for this configuration to be used. Useful in the case when directive is used as component.
  2. **restrict**: [EACM] : which restricts the directive to a specific directive declaration style

**E** : Element. example usage: <my-directive></my-directive>

**A** : Attribute. example usage: <div my-directive></div>

**C** : Class. example usage: <div class='my-directive'></div>

**M** : Comment. example usage: <!-- directive:my-directive -->

* 1. **template**: replace the current element with the contents of the HTML.
  2. **templateUrl**: Same as template but specify url location
  3. **replace**: Specify where the template should be inserted. Defaults to false.

true - the template will replace the current element.

false - the template will replace the contents of the current element.

* 1. **transclude**: compile the content of the element and make it available to the directive.
  2. **compile**: The compile function deals with transforming the template DOM
  3. **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. Most of the directive logic will be put here.

#### 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. It is safe to do DOM transformation in the post-linking function.

* **Routing** :

Routing is performed by using the config function of the module

app.config(function($routeProvider){

$routeProvider.when('/', {

templateUrl: 'templates/home.html',

controller: 'homePageCtrl'

});

$routeProvider.when('/person/:id', {

templateUrl: 'templates/profile.html',

controller: 'profileCtrl'

});

});

* **Filters**

Filters are a smaller, but often useful part of AngularJS. Filters are used to transform data when it is displayed to the user. Like directives, you can build them yourself, and Angular also includes several useful ones built-in. Let’s look at the following HTML:

1. <span>Price:</span>300 O/P – 300

2. We can apply a couple filters to this to make it look different, by using the | operator:

<span>{{ "Price:" | uppercase }}</span> {{ "300" | currency }} O/P - $300

* **What is $broadcast, $emit and $on**

First of all, **parent-child** scope relation does matter. You have two possibilities to emit some event:

$broadcast -- dispatches the event downwards to all child scopes,

$emit -- dispatches the event upwards through the scope hierarchy.

If scope of firstCtrl is parent of the secondCtrl scope:

function firstCtrl($scope){

$scope.$broadcast('someEvent', [1,2,3]);

}

function secondCtrl($scope){

$scope.$on('someEvent', function(event, mass) {console.log(mass)});

}

In case there is no parent-child relation between your scopes, you can inject $rootScope into the controller and broadcast the event to all child scopes (i.e. also secondCtrl).

function firstCtrl($rootScope){

$rootScope.$broadcast('someEvent', [1,2,3]);

}

Finally, when you need to dispatch the event from child controller to scopes upwards you can use$scope.$emit. If scope of firstCtrl is parent of the secondCtrl scope:

function firstCtrl($scope){

$scope.$on('someEvent', function(event, data) { console.log(data); });

}

function secondCtrl($scope){

$scope.$emit('someEvent', [1,2,3]);

}

Angular is very good **Single Page Application** [*SPA*] framework.

Single Page Application is one in which we have a shell page and we can load multiple views into that.

What ng-model does is behind the scenes it’s going to add a property up in the memory called

“name” into what’s called “the scope”.

What is **$scope.$apply()**:

AngularJS provides wrappers for common native JS async behaviors:

* Events => ng-click
* Timeouts => $timeout
* jQuery.ajax() => $http

This is just a traditional async function with a $scope.$apply() called at the end, to tell AngularJS that an asynchronous event just occurred.

$scope.$apply() **should occur as close to the async event binding as possible.**

You can think of $apply **as of integration mechanism**. You see, each time you change some**variable attached to the** $scope object directly, Angular knows that the change has happened. So if it happens in code managed by the framework, the digest cycle will carry on. However, sometimes you want to **change some value outside of the Angular world** and see the changes propagate normally. Consider this - you have a $scope.myVar value which will be set in jQuery's $.ajax() handler. This will happen at some point in future. Angular can't wait for this to happen, since it hasn't been instructed to wait on jQuery. To tackle this, $apply has been introduced. It lets you to start the digestion cycle explicitly. You should only use this to migrate some data to Angular, but never use this method combined with regular Angular code, as Angular will throw error then.

**Angular Directives**

* + 1. **ng-class**:

**Using Object way- [key value pairing]**

*<div ng-class="{border:true ,backColor:false}">*

*Testing Angular CSS*

*</div>*

**Using Array way- [class names]**

*<div ng-class="[‘border’,’ backColor’]">*

*Testing Angular CSS*

*</div>*

*Where* ***border*** *and* ***backColor*** *are css classes defined in style tag.*

------------------------------

* 1. compile
  2. link
  3. angular.element
  4. watch
  5. directive
  6. module (angular.module)
  7. model (ng-model)
  8. controller
  9. view / template
  10. ng-transclude
  11. Dependency injection
  12. E-to-E testing
  13. Scope inheritance
  14. $eval or $parse
  15. $observe
  16. $timeout