ANGULARJS CHEAT SHEET

AngularJS is an extensible and exciting new JavaScript MVC framework developed by Google for building well-designed, structured and interactive single-page applications (SPA). It lays strong emphasis on Testing and Development best practices such as templating and declarative bi-directional data binding.

This cheat sheet co-authored by Ravi Kiran and Suprotim Agarwal, aims at providing a quick reference to the most commonly used features in AngularJS. It will also make you quickly productive with Angular.

BEGINNER

01 Important AngularJS Components and their usage:

- angular.module() defines a module
- Module.controller() defines a controller
- Module.directive() defines a directive
- Module.filter() defines a filter
- Module.service() or Module.factory() or Module.provider() defines a service
- Module.value() defines a service from an existing object Module
- ng-app attribute sets the scope of a module
- ng-controller attribute applies a controller to the view
- $scope service passes data from controller to the view
- $filter service uses a filter
- ng-app attribute sets the scope of the module

02 Bootstrapping AngularJS application:

Bootstrapping in HTML:

```
<div ng-app="moduleName"></div>
```

Manual bootstrapping:

```
angular.bootstrap(document,["moduleName"])
```

03 Expressions:

`{{ 4+5 }}` -> yields 9

`{{ name }}` -> Binds value of name from current scope and watches for changes to name

`{{ ::name }}` -> Binds value of name from current scope and doesn't watch for change (Added in AngularJS 1.3)

04 Module:

Create a module named myModule1 that depends on myModule2 and myModule2:

```
angular.module("myModule1",["myModule2",
"myModule2"])
```

Get reference to the module myModule1

```
angular.module("myModule1")
```

05 Defining a Controller and using it:

i. With $scope:

```
angular.module("myModule").
controller("SampleController",
function($scope,){
  //Members to be used in view for binding
  $scope.city="Hyderabad";
});
```

In the view:

```
<div ng-controller="SampleController">
  <!-- Template of the view with binding
  expressions using members of $scope -->
  <div>{{city}}</div>
</div>
```

ii. Controller as syntax:

```
angular.module("myModule").
controller("SampleController", function(){
  var controllerObj = this;
  //Members to be used on view for binding
  controllerObj.city="Hyderabad";
});
```

In the view:

```
<div ng-controller="SampleController as ctrl">
  <div>{{ctrl.city}}</div>
</div>
```

06 Defining a Service:

```
angular.module("myModule").
service("sampleService", function(){
  var svc = this;
  var cities=["New Delhi", "Mumbai",
  "Kolkata", "Chennai"];
  svc.addCity = function(city){
    cities.push(city);
  };

  svc.getCities = function(){
    return cities;
  }
});
```

The members added to instance of the service are visible to the outside world. Others are private to the service. Services are singletons, i.e. only one instance of the service is created in the lifetime of an AngularJS application.

07 Factory:

```
angular.module("myModule").
factory("sampleFactory", function(){
  var cities = ["New Delhi", "Mumbai",
  "Kolkata", "Chennai"];
  function addCity(city){cities.push(city);}
  function getCities(){return cities;}
  return{
    getCities: getCities,
    addCity:addCity
  };
});
```

A factory is a function that returns an object. The members that are not added to the returning object, remain private to the factory. The factory function is executed once and the result is stored. Whenever an application asks for a factory, the application returns the same object. This behavior makes the factory a singleton.

08 Value:

```
angular.module("myModule").value("sampleValue", {
  cities : ["New Delhi", "Mumbai", "Kolkata",
  "Chennai"],
  addCity: function(city){cities.push(city);},
  getCities: function(){return cities;}
});
```

A value is a simple JavaScript object. It is created just once, so value is also a singleton. Values can't contain private members. All members of a value are public.

09 Constant:

```
angular.module("myModule").
constant("sampleConstant",{pi: Math.PI});
```

A constant is also like a value. The difference is, a constant can be injected into config blocks, but a value cannot be injected.

10 Provider:

```
angular.module("myModule").
provider("samplePrd", function(){
  this.initCities = function(){
    console.log("Initializing Cities…");
  };

  this.$get = function(){
    var cities = ["New Delhi", "Mumbai",
    "Kolkata", "Chennai"];
    function addCity(city){
      cities.push(city);
    }
    function getCities(){return cities;}
      return{ getCities: getCities,addCity:addCity
    };
  }
});
```

A provider is a low level recipe. The $get() method of the provider is registered as a factory. Providers are available to config blocks and other providers. Once application configuration phase is completed, access to providers is prevented.

After the configuration phase, the $get() method of the providers are executed and they are available as factories. Services, Factories and values are wrapped inside provider with $get() method returning the actual logic implemented inside the provider.

11 Config block:

```
angular.module("myModule").config(function
(samplePrdProvider, sampleConstant){
  samplePrdProvider.init();
  console.log(sampleConstant.pi);
});
```

Config block runs as soon as a module is loaded. As the name itself suggests, the config block is used to configure the application. Services, Factories and values are not available for config block as they are not created by this time. Only providers and constants are accessible inside the config block. Config block is executed only once in the lifetime of an Angular application.

12 Run block:

```
angular.module("myModule").run(function(<any
services, factories>){
  console.log("Application is configured. Now inside run
  block");
});
```

Run block is used to initialize certain values for further use, register global events and anything that needs to run at the beginning of the application. Run block is executed after config block, and it gets access to services, values and factories. Run block is executed only once in the lifetime of an Angular application.

13 Filters:

```
angular.module("myModule").
filter("dollarToRupeee",  function(){
  return function(val){
    return "Rs. " + val * 60;
  };
});
```

Usage:

```
<span>{{price | dollarToRupee}}</span>
```

Filters are used to extend the behavior of binding expressions and directives. In general, they are used to format values or to apply certain conditions. They are executed whenever the value bound in the binding expression is updated.

14 Directives:

```
myModule.directive("directiveName", function
(injectables) {
  return {
    restrict: "A",
    template: "<div></div>",
    templateUrl: "directive.html",
    replace: false,
    transclude: false,
    scope: false,
```

```
      require: ["someOtherDirective"],
      controller: function($scope, $element,
      $attrs, $transclude, otherInjectables) { ... },
      link: function postLink(scope, iElement,
     iAttrs) { ... },
      priority: 0,
      terminal: false,
      compile: function compile(tElement, tAttrs,
     transclude) {
      return {
         pre: function preLink(scope, iElement,
         iAttrs, controller) { ... },
         post: function postLink(scope,
         iElement, iAttrs, controller) { ... }
      }
    }
  };
});
```

Directives add the capability of extending HTML. They are the most complex and the most important part of AngularJS. A directive is a function that returns a special object, generally termed as *Directive Definition Object*. The Directive Definition Object is composed of several options as shown in the above snippet. Following is a brief note on them:

- **restrict:** Used to specify how a directive can be used. Possible values are: E (element), A (Attribute), C (Class) and M (Comment). Default value is A

- **template:** HTML template to be rendered in the directive

- **templateUrl:** URL of the file containing HTML template of the element

- **replace:** Boolean value denoting if the directive element is to be replaced by the template. Default value is false

- **transclude:** Boolean value that says if the directive should preserve the HTML specified inside directive element after rendering. Default is false

- **scope:** Scope of the directive. It may be same as the scope of surrounding element (default or when set to false), inherited from scope of the surrounding element (set to true) or an isolated scope (set to {})

- **require:** A list of directive that the current directive needs. Current directive gets access to controller of the required directive. An object of the controller is passed into link function of the current directive.

- **controller:** Controller for the directive. Can be used to manipulate values on scope or as an API for the current directive or a directive requiring the current directive

- **priority:** Sets priority of a directive. Default value is 0. Directive with higher priority value is executed before a directive with lower priority

- **terminal:** Used with priority. If set to true, it stops execution of directives with lower priority. Default is false

- **link:** A function that contains core logic of the directive. It is executed after the directive is compiled. Gets access to scope, element on which the directive is applied (jqLite object), attributes of the element containing the directive and controller object. Generally used to perform DOM manipulation and handling events

- **compile:** A function that runs before the directive is compiled. Doesn't have access to scope as the scope is not created yet. Gets an object of the element and attributes. Used to perform DOM of the directive before the templates are compiled and before the directive is transcluded. It returns an object with two link functions:

  - **pre link:** Similar to the link function, but it is executed before the directive is compiled. By this time, transclusion is applied

  - **post link:** Same as link function mentioned above

15 Most used built-in directives:

- **ng-app:** To bootstrap the application

- **ng-controller:** To set a controller on a view

- **ng-view:** Indicates the portion of the page to be updated when route changes

- **ng-show / ng-hide:** Shows/hides the content within the directive based on boolean equivalent of value assigned

- **ng-if:** Places or removes the DOM elements under

this directive based on boolean equivalent of value assigned

- **ng-model:** Enables two-way data binding on any input controls and sends validity of data in the input control to the enclosing form

- **ng-class:** Provides an option to assign value of a model to CSS, conditionally apply styles and use multiple models for CSS declaratively

- **ng-repeat:** Loops through a list of items and copies the HTML for every record in the collection

- **ng-options:** Used with HTML select element to render options based on data in a collection

- **ng-href:** Assigns a model as hyperlink to an anchor element

- **ng-src:** Assigns a model to source of an image element

- **ng-click:** To handle click event on any element

- **ng-change:** Requires ng-model to be present along with it. Calls the event handler or evaluates the assigned expression when there is a change to value of the model

- **ng-form:** Works same as HTML form and allows nesting of forms

- **ng-non-bindable:** Prevents AngularJS from compiling or binding the contents of the current DOM element

- **ng-repeat-start** and **ng-repeat-end:** Repeats top-level attributes

- **ng-include:** Loads a partial view

- **ng-init:** Used to evaluate an expression in the current scope

- **ng-switch** conditionally displays elements

- **ng-cloak** to prevent Angular HTML to load before bindings are applied.

16 AngularJS Naming Conventions

- While naming a file say an authentication *controller*, end it with the object suffix. For eg: an authentication controller can be renamed as auth–controller.js. Similar *service* can be called as auth-service.js, *directive* as auth-directive.js and a *filter* as auth-filter.js

- Create meaningful & short lower case file names that also reflect the folder structure. For eg: if we have a login controller inside the login folder which is used for creating users, call it login-create-controller.js

- Similar a testing naming convention that you could follow is if the filename is named as login-directive.js, call its test file counterpart as login-directive_test.js. Similarly a test file for login-service.js can be called as login-service_test.js
Use a workflow management tool like *Yeoman* plugin for Angular that automates a lot of these routines and much more for you. Also look at *ng-boilerplate* to get an idea of the project and directory structure.

17 Dependency Injection:

AngularJS has a built-in dependency injector that keeps track of all components (services, values, etc.) and returns instances of needed components using dependency injection. Angular's dependency injector works based on names of the components.

A simple case of dependency injection:

```
myModule.controller("MyController", function($scope,
$window, myService){
});
```

Here, $scope, $window and myService are passed into the controller through dependency injection. But the above code will break when the code is minified. Following approach solves it:

```
myModule.controller("MyController", ["$scope",
"$window", "myService",
        function($scope, $window, myService){
}]);
```

18 Routes

Routes in AngularJS application are defined using $routeProvider. We can define a list of routes and set one of the routes as default using otherwise() method; this route will respond when the URL pattern doesn't match any of the configured patterns.

19 Registering routes:

```
myModule.config(function($routeProvider){
  $routeProvider.when("/home",
  {templateUrl:"templates/home.html",
  controller: "HomeController"})
  .when("/details/:id", {template:
  "templates/details.html",
   controller:"ListController"})
  .otherwise({redirectTo: "/home"});
});
```

20 Registering services:

Angular provides us three ways to create and register our own Services – using Factory, Service, and Provider. They are all used for the same purpose. Here's the syntax for all the three:

```
Service: module.service( 'serviceName', function );
Factory: module.factory( 'factoryName', function );
Provider: module.provider( 'providerName', function );
```

The basic difference between a service and a factory is that service uses the constructor function instead of returning a factory function. This is similar to using the *new* operator. So you add properties to 'this' and the service returns 'this'.

With Factories, you create an object, add properties to it and then return the same object. This is the most common way of creating Services.

If you want to create module-wide configurable services which can be configured before being injected inside other components, use Provider. The provider uses the $get function to expose its behavior and is made available via dependency injection.

21 Some useful utility functions

- **angular.copy -** Creates a deep copy of source
- **angular.extend -** Copy methods and properties from one object to another
- **angular.element -** Wraps a raw DOM element or HTML string as a jQuery element
- **angular.equals -** Determines if two objects or two values are equivalent
- **angular.forEach -** Enumerate the content of a collection
- **angular.toJson -** Serializes input into a JSON-formatted string
- **angular.fromJson -** Deserializes a JSON string
- **angular.identity -** Returns its first argument
- **angular.isArray -** Determines if a reference is an Array
- **angular.isDate -** Determines if a value is a date
- **angular.isDefined -** Determines if a reference is defined
- **angular.isElement -** Determines if a reference is a DOM element
- **angular.isFunction -** Determines if a reference is a Function
- **angular.isNumber -** Determines if a reference is a Number
- **angular.isObject -** Determines if a reference is an Object
- **angular.isString -** Determines if a reference is a String
- **angular.isUndefined -** Determines if a reference is undefined
- **angular.lowercase -** Converts the specified string to lowercase
- **angular.uppercase -** Converts the specified string to uppercase

22 $http:

$http is Angular's wrapper around XmlHttpRequest. It provides a set of high level APIs and a low level API to talk to REST services. Each of the API methods return $q promise object.

Following are the APIs exposed by $http:

- **$http.$get(url):** Sends an HTTP GET request to the URL specified

- **$http.post(url, dataToBePosted):** Sends an HTTP POST request to the URL specified

- **$http.put(url, data):** Sends an HTTP PUT request to the URL specified

- **$http.patch(url, data):** Sends an HTTP PATCH request to the URL specified

- **$http.delete(url):** Sends an HTTP DELETE request to the URL specified

- **$http(config):** It is the low level API. Can be used to send any of the above request types and we can also specify other properties to the request. Following are the most frequently used config options:

  - **method:** HTTP method as a string, e.g., 'GET', 'POST', 'PUT', etc.
  - **url:** Request URL
  - **data:** Data to be sent along with request
  - **headers:** Header parameters to be sent along with the request
  - **cache:** caches the response when set to true

Following is a small snippet showing usage of $http:

```
$http.get('/api/data').then(function(result){
  return result.data;
  }, function(error){
    return error;
});
```

ANGULARJS CHEAT SHEET

INTERMEDIATE - ADVANCED

23 Manage Dependencies

Use a package management tool like *Bower* (bower.io/) to manage and update third-party web dependencies in your project. It is as simple as installing bower using `npm install bower;` then listing all the dependent libraries and versions in a Bower package definition file called bowerconfig.json and lastly run `bower install` or `bower update` in your project to get the latest versions of any web dependencies in your project.

24 Using AngularJS functions

Wherever possible, use AngularJS versions of JavaScript functionality. So instead of setInterval, use the `$interval` service. Similarly instead of setTimeout use the `$timeout` service. It becomes easier to mock them or write unit tests . Also make sure to clean it up when you have no use for it. Use the $destroy event to do so:

```
$scope.$on("$destroy", function (event) {
    $timeout.cancel(timerobj);
});
```

25 Services

If you need to share state across your application, or need a solution for data storage or cache, think of *Services*. Services are singletons and can be used by other components such as directives, controllers, filters and even other services. Services do not have a scope of their own, so it is permissible to add eventlisteners in Services using `$rootScope`.

26 Deferred and Promise

The $q service provides deferred objects/promises.

- **$q.all([array of promises]) -** creates a Deferred object that is resolved when all of the input promises in the specified array are resolved in future

- **$q.defer() -** creates a deferred object with a promise property that can be passed around applications, especially in scenarios where we are integrating with a 3rd-party library

- **$q.reject(reason) -** Creates a promise that is resolved as rejected with the specified reason. The return value ensures that the promise continues to the next error handler instead of a success handler.

- **deferredObject.resolve -** Resolves the derived promise with the value

- **deferredObject.reject -** Rejects the derived promise with the reason and triggers the failure handler in the promise.

27 Manipulating $scope

Do not make changes to the $scope from the View. Instead do it using a Controller. Let's see an example. The following piece of code controls the state of the dialog property directly from the ng-click directive.

```
<div>
   <button ng-click="response = false">Close Dialog
   </button>
</div>
```

Instead we can do this in a Controller and let it control the state of the dialog as shown here:

```
<div>
   <button ng-click="getResponse()">Close Dialog
   </button>
</div>
```

In dialog-controller.js file, use the following code:

```
dialog.controller("diagCtrl", function ($scope) {
    $scope.response = false;

    $scope.getResponse = function () {
        $scope.response = false;
    }
});
```

This reduces the coupling between the view and controller

28 Prototypal Inheritance

Always have a '.' in your ng-models which insures prototypal inheritance. So instead of

```
<input type="text" ng-model="someprop">
```

use

```
<input type="text" ng-model="someobj.someprop">
```

29 Event Aggregator:

$scope includes support for event aggregation. It is possible to publish and subscribe events inside an AngularJS application without need of a third party library. Following methods are used for event aggregation:

- **$broadcast(eventName, eventObject):** Publishes an event to the current scope and to all children scopes of the current scope

- **$emit(eventName, eventObject):** Publishes an event to the current scope and to all parent scopes of the current scope

- **$on(eventName, eventHandler):** Listens to an event and executes logic inside eventHandler when the event occurs**.**

30 $resource

$resource is a higher level service that wraps $http to interact with RESTful APIs. It returns a class object that can perform a default set of actions (get (GET), save (POST), query (GET), remove(DELETE), delete (DELETE)). We can add more actions to the object obtained.

```
//A factory using $resource
myModule.factory('Student', function($resource){
    return $resource('/api/students',  null, {
        change: {method: 'PUT'}
    });
});
```

The above operation returns a $resource object that has all default operations and the change method that performs a PUT operation.

31 $timeout and $interval

$timeout is used to execute a piece of code after certain interval of time. The $timeout function takes three parameters: function to execute after time lapse, time to wait in milliseconds, a flag field indicating whether to perform $scope.$apply after the function execution.

```
$timeout(function () {
    //Logic to execute
}, 1000, true);
```

$interval is used to keep calling a piece of code repeatedly after certain interval. If count is not passed, it defaults to 0, which causes the call to happen indefinitely.

```
$interval(function () {
  //Logic to execute
}, 1000, 10, true);
```

32 jqLite and jQuery

AngularJS uses a lighter version of jQuery called jqLite to perform DOM manipulations. The element we receive in compile and link functions of directive are jqLite objects. It provides most of the necessary operations of jQuery. Following snippet shows obtaining a jqLite object for all divs on a page using selector:

```
var divJqliteObject = angular.element('div');
```

But, if jQuery library is referred on the page before referring AngularJS, then Angular uses jQuery and all element objects are created as jQuery objects.

If a jQuery plugin library is referred on the page before referring AngularJS, then the element objects get capabilities of the extended features that the plugins bring in.

33 ngCookie:

ngCookie is a module from the AngularJS team that wraps cookies and provides an easier way to deal with cookies in an AngularJS application. It has two services:

i. **$cookieStore:** Provides a key-value pair kind of interface to talk to the cookies in the browser. It has methods to get value of a stored cookie, set value to a cookie and remove a cookie. The data is automatically serialized/de-serialized to/from JSON.

ii. **$cookies:** An object representing the cookies. Can be used directly to get or set values to cookies

34 Unit testing:

AngularJS is built with unit testing in mind. Every component defined in Angular is testable. Dependency injection is the key factor behind it. Take any kind of component in Angular, it can't be written without getting some of the external components

injected in. This gives freedom to programmers to pass any object of their choice instead of the actual component object while writing tests. The only thing to be taken care is to create an object with the same shim as the component.

AngularJS code can be unit tested using any JavaScript Unit Testing framework like QUnit, Jasmine, Mocha or any other framework. Jasmine is the most widely used testing framework with Angular. Tests can be run anywhere, in browser or even in console using Karma test runner.

The main difference between application code and unit tests is, application code is backed by the framework and browser, whereas unit tests are totally under our control. So, wherever we get objects automatically injected or created by AngularJS, these objects are not available in unit tests. They have to be created manually.

35 Bootstrapping a unit test:

Just like the case of Angular application, we need to bootstrap a module in unit tests to load the module. As the module has to be loaded fresh before any test runs, we load module while setting up the tests. In Jasmine tests, setup is doe using beforeEach block.

```
beforeEach(function(){
  module('myModule');
});
```

36 Creating $scope in unit tests:

$scope is a special injectable in AngularJS. It is unlike other objects as it is not already created to pass into a component when asked. A $scope can be injected only inside controllers and for every request of $scope, a new $scope object is created that is inherited from $rootScope. Framework takes care of creating the scope when the application is executed. We have to do it manually to get a $scope object in tests. Following snippet demonstrates creation of $scope:

```
var scope;

beforeEach(inject(function ($rootScope) {
    scope = $rootScope.$new();
}));
```

37 Testing controllers:

In an AngularJS application, we generally don't need to create an object of a controller manually. It gets created whenever a view loads or the template containing an ng-controller directive loads. To create it manually, we need to use the *$controller* service. To test the behavior of controller, we need to manually create object of the controller.

```
inject(function($controller){
  var controller = $controller('myController',
  { $scope: scope, service: serviceMock });
});
```

As we see, arguments to the controller are passed using a JavaScript object literal. They would be mapped to right objects according to names of the services. After this, the scope would have all properties and methods that are set in the controller. We can invoke them to test their behavior.

```
it('should return 10', function(){
  var val = scope.getValue();
  expect(val).toEqual(10);
});
```

38 Testing services:

Getting object of a service is easy as it is directly available to the inject() method.

```
var serviceObj;
beforeEach(inject(function (myService) {
  serviceObj = service;
}));
```

Now any public method exposed from the service can be called and the result can be tested using assertions.

```
it('should get some data', function(){
  var data = serviceObj.getCustomers();
  expect(data).not.toBe(null);
  expect(data).not.toBe(undefined);
});
```

39 ng-controller outside ng-view:

Though controllers are used with views in general, it doesn't mean that they can't be used outside a view. A controller can be made responsible to load menu items, show toast messages,

update user when a background task is completed or any such thing that doesn't depend on the view loaded inside ng-view.

```
<div ng-app=”myModule”>
    <div ng-controller=”menuController”>
        <!-- Mark-up displaying Menu -->
    </div>
    <div ng-view></div>
</div>
```

40

To avoid controllers from getting too complicated, you can split the behavior by creating Nested Controllers. This lets you define common functionality in a parent controller and use it one or more child controllers. The child controller inherits all properties of the outside scope and in case of equality, overrides the properties.

```
<body ng-controller=”mainCtrller”>
  <div ng-controller=”firstChildCtrller”>
  </div>
</body>
```

41 Mocking services:

Mocking is one of the most crucial things in unit testing. It helps in keeping the system under test isolated from any dependency that it has. It is very common to have a component to depend on a service to get a piece of work done. This work has to be suppressed in unit tests and replaced with a mock or stub. Following snippet mocks a service:

Say, we have following service:

```
app.service(‘customersSvc’, function(){
  this.getCustomers = function(){
    //get customers and return
  };
  this.getCustomer = function(id){
    //get the customer and return
  };
  this.addCustomer = function(customer){
    //add the customer
  };
});
```

To mock this service, we need to create a simple object with three mocks with the names same as the ones in the service and ask Angular's injector to return the object whenever the service is requested.

```
var mockCustomersSvc;

beforEach(function(){
  mockCustomerService = {
    getCustomers: jasmine.createSpy(‘getCustomers’),
    getCustomer: jasmine.createSpy(‘getCustomer’),
    addCustomers: jasmine.createSpy(‘addCustomer’)
  };

  module(function($provide){
    $provide.value(‘customersSvc’, mockCustomersSvc);
  });
});
```

42 ngMock

The ngMock module provides useful tools for unit testing AngularJS components such as controllers, filters, directives and services.

- The *module* function of ngMock loads the module you want to test and it's inject method resolves the dependencies on the service to be tested

- We can mock the backend and test components depending on the *$http service* using the *$httpBackend* service in ngMocks

- We can mock timeouts and intervals using *$interval* and *$timeout* in ngMocks

- The *$log* service can be used for test logging

- The *$filter* service allows us to test filters

- Directives are complex to test. Use the $compile service and *jqLite* to test directives

43 ng-class:

ng-class can be used in multiple ways to dynamically apply one or more CSS classes to an HTML element. One of the very good features is, it supports some simple logical operators too. Following list shows different ways of using ng-class:

i. `<div ng-class=”dynamicClass”>some text</div>`

Assigns value of dynamicClass from scope to the CSS class. It is two-way bound, i.e. if value of dynamicClass changes, the style applied on the div also changes.

ii. `<div class="[class1, class2, class3]">some text</div>`

All classes mentioned in the array are applied

iii. `<div class="{''my-class-1'':value1, ''my-class-2'':value2}">some text</div>`

my-class-1 is applied when value1 is assigned with a truthy value (other than false, empty string, undefined or null)

iv. `<div ng-class="value ? 'my-class-1':'my-class-2'">some text</div>`

Value of class applied is based on result of the ternary operator.

v. `<div ng-class="{true: 'firstclass'}[applyfirstclass] || {true:'secondclass'}[applysecondclass]"></div>`

Here, applyFirstClass and applySecondClass are data bound variables. The expression applies firstClass if applyFirstClass is true. It applies secondClass only if applySecondClass is true and applyFirstClass is false.

44 Resolve blocks in routing:

Resolve blocks are used to load data before a route is resolved. They can be used to validate authenticity of the user, load initial data to be rendered on the view or to establish a real-time connection (e.g. Web socket connection) as it would be in use by the view. View is rendered only if all the resolve blocks of the view are resolved. Otherwise, the route is cancelled and the user is navigated to the previous view.

```
$routeProvider.when('/details', {
  templateUrl: 'detailsView.html',
  controller: 'detailsController',
  resolve: {
    loadData: function (dataSvc, $q) {
      var deferred = $q.defer;
      dataSvc.getDetails(10).then(
      function (data) { deferred.reslve(data);},
       function () { deferred.reject();});
            return deferred.promise;
    }
  }});
```

In the above snippet, the route won't be resolved if the promise is rejected. Resolve block can be injected into the controller and data resolved from the resolve block can be accessed using the injected object.

45 $compile

Used to compile templates after the compilation phase. $compile is generally used in link function of directives or services. But, it should be used with caution as manual compilation is an expensive operation.

```
myModule.directive('sampleDirective', function(){
  return {
    link: function(scope, elem, attrs){
      var compiled = $compile('<div>{{person.name}}
      </div>')(scope);
                    elem.html(compiled);
    }
  };
});
```

46 $parse

$parse is used to transform plain text into expression. The expression can be evaluated against any object context to obtain the value corresponding to the object. Very common usage of parse is inside directives to parse the text received from an attribute and evaluate it against scope. The expression also can be used to add a watcher.

```
myModule.directive('sampleDirective', function($parse){
  return function(scope, elem, attrs){
    var expression = $parse(attrs.tempValue);

    var value = expression(scope);
    scope.$watch(expression, function(newVal, oldVal){
      //Logic to be performed
    });
  };
});
```

47 Route change events:

When a user navigates from one page to another, AngularJS broadcasts events at different phases. One can listen to these events and take appropriate action like verifying login status, requesting for data needed for the page or even to count the number of hits on a view. Following are the events raised:

- $routeChangeStart
- $routeChangeSuccess
- $routeChangeError
- $routeUpdate

48 Decorating

It is possible to modify the behavior or extend the functionality of any object in AngularJS through decoration. Decoration is applied in AngularJS using $provide provider. It has to be done in config block. Following example adds a method to the value:

```
angular.module('myModule',[])
  .config(function($provide) {
    $provide.decorator('someValue', function($delegate)
    {
      $delegate.secondFn = function(){
        console.log("Second Function");
      };

      return $delegate;
    });
  })
  .value('someValue',{
    firstFn: function(){console.log("First Function");}
  });
```

Note: *Constants cannot be decorated*

49 Exception handling

All unhandled exceptions in an AngularJS application are passed to a service $exceptionHandler, which logs the error message in the browser's console. In large business applications, you may want to log the error details on the server by calling an API. This can be done by decorating the $exceptionHandler service.

```
myApp.config(function ($provide) {
  $provide.decorator('$exceptionHandler', ['$log',
  '$http', '$delegate',
    function ($log, $http, $delegate) {
      return function (exception, cause) {
        $log.debug('Modified exception handler');
        $http.post('/api/clientExceptionLogger',
        exception);
        $delegate(exception, cause);
      };
    }
  ]);
});
```

50 HTTP Interceptors

Any HTTP request sent through $http service can be intercepted to perform certain operation at a given state. The state may be one of the following: before sending request, on request error, after receiving response and on response error. Interceptors are generally used to check authenticity of the request before sending to the server or to displaying some kind of wait indicator to the user when the user has to wait for the data to arrive. The intercepting methods may return either plain data or a promise.

```
myModule.config(function ($provide) {
  $provide.factory('myHttpInterceptor', function () {
    return {
      request: function (req) {
        //logic before sending request
      },
      response: function (res) {
        //logic after receiving response
      },
      requestError: function () {
        //logic on request error
      },
      responseError: function () {
        //logic on response error
      }
    };
  });
});
```

51 HTTP Transforms

Transforms allow us to tweak the data before sending to an HTTP request or after receiving response at the end of a request. It can be applied either globally using a config block or on a specific request using the $http config object.

Setting transform in config block:

```
myModule.config(function ($httpProvider) {
  $httpProvider.defaults.transformRequest.push(function
  (data) { //Operate on data });
});
```

In the individual request:

```
$http({
  url: '/api/values',
  method: 'GET',
  transformRequest: function (data) {//Operate on data}
});
```