Introduction To AngularJS

VERNALIS

AngularJS

- AngularJS is a very powerful JavaScript Framework.
- It is a structural framework for dynamic web apps.
- AngularJS is open source.
- It is used in Single Page Application (SPA) projects.
- It extends HTML DOM with additional attributes and makes it more responsive to user actions.

The Angularjs Components

The AngularJS framework can be divided into following three major parts –

- ng-app – This directive defines and links an AngularJS application to HTML.
- ng-model – This directive binds the values of AngularJS application data to HTML input controls.
- ng-bind – This directive binds the AngularJS Application data to HTML tags.

Advantages of Angularjs

- AngularJS provides data binding capability to HTML thus giving user a rich and responsive experience
- AngularJS provides capability to create Single Page Application in a very clean and maintainable way.
- AngularJS code is unit testable.
- AngularJS provides reusable components.
- With AngularJS, developer write less code and get more functionality.

ng-app

- The ngApp directive designates the root element of the application
- It is typically placed near the root element of the page - e.g. on the <body> or <html> tags.

Syntax

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

Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js"></script>
<body>
<div ng-app="myApp" ng-controller="myCtrl">
<label>Full Name:</label> {{firstName + " " + lastName}}
</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
    $scope.firstName = "This is the example for";
    $scope.lastName = "Angularjs ng-app";
});
</script>
</body>
</html>
```

ng-controller

- AngularJS application mainly relies on controllers to control the flow of data in the application.
- A controller is defined using ng-controller directive.
- A controller is a JavaScript object containing attributes/properties and functions.
- Each controller accepts $scope as a parameter which refers to the application/module that controller is to control.

Syntax

<element ng-controller="expression"></element>

Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js">
</script>
<body>
<div ng-app="myApp" ng-controller="myCtrl">
<label>Name:</label> <input type="text" ng-model="name">
</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
   $scope.name = "Angularjs";
});
</script>
</body>
</html>
```

ng-repeat

- The ng-repeat directive instantiates a template once per item from a collection. Each template instance gets its own scope, where the given loop variable is set to the current collection item, and $index is set to the item index or key.
- The collection must be an array or objects.
- ng-repeat directive is similar to the for loop.
- The ng-repeat will not take the duplicate items in the collection.

Syntax

<element ng-repeat="item in collection"></element>

The element may be any HTML element.

We can iterate the collection through the $index.

Example:

```
Index.html
<html >
<head>
<script src="AngularScript.js">
</head>
<body ng-app="MyApp">
    <table ng-controller="RepeatController">
          <tr ng-repeat="item in students">
               <td>{{item.Name}}</td>
               <td>{{item.Age}}</td>
          </tr>
   </table>
</body>
</html>

AngularScript.js
Var app=angular.module("MyApp",[])
app.controller("RepeatController",function($scope){
$scope.students=[
{
"Name" : "sadham"
```

```
"Age" : "22"
},
{
"Name" : "Ashwin"
"Age" : "25"
},
{
"Name" : "Leo"
"Age" : "24"
},
{
"Name" : "Raj"
"Age" : "24"
}
]
)};
```

Output

Sadham	22
Ashwin	25
Leo	24
Raj	24

Data Binding

The angularjs data binding can be done by the two ways.

1. ng-model
2. ng-bind

ng-bind

- The ng-bind attribute tells Angular 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.
- ng-bind is one way data binding used for displaying the value inside html component as innerHTML.

Syntax

<element ng-bind="expression"></element>

Example:

```
Bind.html
<html>
<script>
 Var app=angular.module('bindexample',[])
app.controller('bindcontroller',function($scope)
{
scope.name='Raj Kumar';
});
</script>
<body ng-app="bindexample">
    <div ng-controller="bindcontroller">
     Welcome <span ng-bind="name"></span>
    </div>
</body>
</html>
```

Output
Welcome Raj Kumar

ng-model

- The ng-model directive binds an input,select,textarea (or custom form control) to a property on the scope using ngmodelcontroller, which is created and exposed by this directive.
- This works as two way data binding. Two way binding means you have an input component and the value updated into that field must be reflected in other part of the application.
- We can add or remove the classes according to the status of the input.
    - ng-empty
    - ng-not-empty
    - ng-touched
    - ng-untouched
    - ng-valid
    - ng-invalid
    - ng-dirty
    - ng-pending
    - Ng-pristine

Ng-model is responsible for:

- Binding the view into the model, which other directives such as `input`, `textarea` or `select` require.
- Providing validation behavior (i.e. required, number, email, url).
- Keeping the state of the control (valid/invalid, dirty/pristine, touched/untouched, validation errors).
- Setting related css classes on the element (`ng-valid`, `ng-invalid`, `ng-dirty`, `ng-pristine`, `ng-touched`, `ng-untouched`,`ng-empty`, `ng-not-empty`) including animations.
- Registering the control with its parent form.

Example

Model.html:

```
<html>

<head>

<script>

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

app.controller('modelController',function($scope){

$scope.name="sadham hussain";

});

</script>

</head>

<body>

<div ng-controller="modelController">

<span>Name:</span><input ng-model="name">

</div>

</body>

</html>
```

ng-class

The ng-class directive allows you to dynamically set CSS classes on an HTML element by databinding an expression that represents all classes to be added.
The directive operates in three different ways, depending on which of three types the expression evaluates to:

1. If the expression evaluates to a string, the string should be one or more space-delimited class names.
2. If the expression evaluates to an object, then for each key-value pair of the object with a truthy value the corresponding key is used as a class name.
3. If the expression evaluates to an array, each element of the array should either be a string as in type 1 or an object as in type 2. This means that you can mix strings and objects together in an array to give you more control over what CSS classes appear. See the code below for an example of this.

Example

```
<!DOCTYPE html>
<html>
<scriptsrc="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js">
</script>
<style>
.red {
    color:white;
    background-color:red;
    padding:20px;
}
.yellow {
    background-color:yellow;
    padding:30px;
}
</style>
<body >
<p>select color to apply:</p>
<select ng-model="home">
<option value="sky">red</option>
```

```
<option value="tomato">yellolw</option>
</select>
<div ng-class="home">
  <h1>Welcome Guest</h1>
</div>
</body>
</html>
```

ng-style

The ng-style directive allows you to set CSS style on an HTML element conditionally.

Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js">
</script>
<body ng-app="myApp" ng-controller="myCtrl">
<h1 ng-style="MyObject">Hello</h1>
<script>
var app = angular.module("myApp", []);
app.controller("myCtrl", function($scope) {
  $scope.MyObject = {
    "color" : "white",
    "background-color" : "coral",
    "font-size" : "60px",
    "padding" : "50px"
  }
});
</script>
</body>
</html>
```

ng-class-odd

The ng-class-odd directive work exactly as ng-class except they work in conjunction with ng-repeat and take effect only on odd rows.

ng-class-even

The ng-class-even directive work exactly as ng-class except they work in conjunction with ng-repeat and take effect only on even rows.

Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js">
</script>
<style>
.odd {
    color:white;
    background-color:blue;
}
.even
{
Color:red;
Background-color:yellow;
}
</style>
<body ng-app="myApp">

<table ng-controller="myCtrl">
<tr ng-repeat="x in records" ng-class-odd="odd" ng-class-even=”even”>
  <td>{{x.Name}}</td>
</tr>
</table>
<script>
```

```
var app = angular.module("myApp", []);
app.controller("myCtrl", function($scope) {
  $scope.records = [
    {
      "Name" : "sadham hussain"
    },
    {
      "Name" : " kumuthan"
    },
    {
      "Name" : "kishore"
    },
    {
      "Name" : "Ashwin"
    }
  ]
});
</script>
</body>
</html>
```

$Scope

- 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.

Scope characteristics

- Scopes provide APIs ($watch) to observe model mutations.
- Scopes provide APIs ($apply) to propagate any model changes through the system into the view from outside of the "Angular realm" (controllers, services, Angular 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. A
- Scopes provide context against which expressions are evaluated. For example {{username}} expression is meaningless, unless it is evaluated against a specific scope which defines the username property.
- Each Angular application has exactly one root scope, but may have several child scopes.

Example

Index.html:

```
<html>
<head>
<script src="script.js">
<link rel="stylesheet" href="style.css">
</head>
<body>
<div class="scopeSample">
<div ng-controller="rootController">
<span>Hello</span>{{name}}
</div>
<div ng-controller="childController">
<ol>
<li ng-repeat="x in names">{{name}} from {{department}}</li>
<ol>
</div>
</div>
```

```
</body>
</html>
Script.js:
Var app=angular.module('scopeexample',[])
app.controller('rootController',['$scope','$rootscope',function($scope,$rootscope){
$scope.name="sadham";
$rootscope.department='computer Science';
}]);
app.controller('childController',[$scope',function($scope){
$scope.names=['sajith','ashwin','bala'];
}]);
Style.css:
.scopeSample.ng-scope{
border:1px solid black;
margin:3px;
}
```

AngularJS Pre-Defined Filters

Filters are used to manipulate and filter the presentation of your views. A filter in AngularJS, allows you to transform or format the data in a specified criteria, without changing original data.

Filters can be added to expressions by using the pipe character " | ", followed by a filter.

AngularJs provides nine predefined filters.They are listed below

NAME	DESCRIPTION
Currency	It is used to format number as currency.
Date	It is used to set date format.
Filter	It is used to filter or search an item from array.
Json	It is used to javascript object to Json object.
LimitTo	It is used to extract subset of array with specified number of items.
Lowercase	It is used convert any string to lowercase.
Uppercase	It is used convert any string to uppercase.
Number	It is used to format any number as string.
Orderby	It is used to set array order ascending or descending.

Example

Uppercase Filter

```
<div ng-app="myApp" ng-controller="myCtrl">

<p>The name is {{ name | uppercase }}</p>

</div>
```

CHAINING FILTERS

Filters can also be chained, by adding the pipe (|) character between each filter so if we wanted to apply multiple filters to a single expression.

Example

```
<div ng-app="myApp" ng-controller="myCtrl">

<p>The name is {{ name | uppercase|limitTo:4 }}</p>

</div>
```

You can also create your own custom filters in angularjs.

AngularJS Pre-Defined Services

In AngularJS,Services are functions and are responsible to do a specific tasks only. Services are normally injected using dependency injection mechanism of AngularJS. AngularJS has about 30 built-in services.

Some of the commonly used services are listed below.

NAME	DESCRIPTION
$http	This service makes a request to the server, and lets your application handle the response.
$timeout	This service performs the specified operation after the specified time delay for only one time. The $timeout service is AngularJS' version of the window.setTimeout function.
$interval	This service performs the specified operation after the specified time delay continuously. The $interval service is AngularJS' version of the window.setInterval function.
$log	Simple service for logging the message into the browser's console.
$location	The $location service parses the URL in the browser address bar and makes the URL available to your application. The $location service is AngularJS' version of the window.location function.

You can also create your own custom services in angularjs.

Example

$http service

```
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope, $http) {
        $http.get("yourpage.html").then(function (response) {
```

```
        //success function operation goes here
        });
});
```

$timeout service

```
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope, $timeout) {
        $timeout(function () {
        //operation has to be performed goes here
        }, 2000);
});
```

$interval service

```
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope, $interval) {
        $interval(function () {
        //operation has to be performed goes here
        }, 1000);
});
```

$location service

```
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope, $location) {
$scope.myUrl=$location.absUrl();//gets the full url window  object
});
```

AngularJS Pre-Defined Directive

AngularJS directives are used to extend HTML.Directives are extended HTML attributes with the prefix "ng-". AngularJs has various predefined directives.
Some of the commonly used directives are listed below.

NAME	DESCRIPTION
ng-app	Initializes an AngularJS application.
ng-controller	Defines the controller object for an application.
ng-model	Binds the content of an HTML element to application data.
ng-init	Defines initial values for an application.
ng-class	Specifies CSS classes on HTML elements.
ng-click	Specifies an expression to evaluate when an element is being clicked.
ng-cloak	Prevents flickering when your application is being loaded.
ng-readonly	Specifies the readonly attribute of an element.
ng-disabled	Specifies if an element is disabled or not.
ng-show	Shows HTML elements.
ng-hide	Hides HTML elements.
ng-value	Specifies the value of an input element.
ng-if	Removes the HTML element if a condition is false.
ng-focus	Specifies a behavior on focus events.
ng-blur	Specifies a behavior on blur events.
ng-change	Specifies an expression to evaluate when content is being changed by the user.

Refer the link for other directive : https://docs.angularjs.org/api/ng/directive

Example

```
<!DOCTYPE html>
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js"></script>
</head>
<body>

<div ng-app="myApp" ng-controller="myCtrl" ng-init="Name='Sam'" ng-cloak>
```

```
<p>Input something in the input box:</p>
<p>Name: <input type="text" ng-model="Name"/>
<p>Above field value is displayed using ng-init</p>

<p>Dept:<input type="text"  ng-model="dept" ng-init="dept='CSE'"ng-readonly=true></p>
<p>Above field is in readonly mode</p>

<p>ID  :<input type="text" ng-model="id" ng-init="id='5'" ng-false-value=""  ng-
disabled=true></p>
<p>Above field is in disabled state</p>

<p>somevalue:<input ng-value="myVar"></p>
<p>Above field value is displayed using ng-value</p>

<p>On focus:<input  ng-focus="count = count + 1" ng-model="count"ng-
init="count=0"></p>
<p>Value of on focus field increments by 1 for each time you select the on focus field </p>

<p>on blur:<input ng-blur="count2 = count2 + 1" ng-model="count2" ng-
init="count2=0"></p>
<p>Value of on blur field increments by 1 for each time you leave the on blur field </p>

<p>on change:<input ng-change="myFunc()" ng-model="changeField"   ></p>
<p>The on change input field has changed {{Count}} times.</p>

<p><input type="checkbox" ng-model="checkBox" ng-init="checkBox = false">Select the
check box</p>

<div ng-if="checkBox">
   <p>This Div is displayed only if the above checkbox is selected</p>
</div>

</div>



<script>
var app = angular.module('myApp', []);
```

```
app.controller('myCtrl', function($scope) {
    $scope.myVar = 5;
    $scope.Count = 0;
    $scope.myFunc = function () {
        $scope.Count++;
    };
});
</script>
</body>
</html>
```

$location Service

In angularjs $location service is same as window.location in javascript and it's having extra features which is used to get information related to url of current web page.
By using $location service in angularjs we can get information like full url of current web page, part of url of web page, protocol of web page, host information of current url and port of current url.

Property	Description
$location.absUrl()	We can get full url of current web page
$location.url()	It returns url without base prefix
$location.protocol()	It returns protocol of current web page url
$location.host()	It returns host of current web page url
$location.port()	By using this we can get port of current url
$location.path()	It returns path of current url without any parameters
$location.search()	Returns search part of current url when called without any parameter
$location.hash()	Returns hash part of current url without any parameters

Examples

```
<head>
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js"></script>
<script type="text/javascript">
var app = angular.module('locationApp', []);
app.controller('locationCtrl', function ($scope, $location) {
$scope.weburl = $location.absUrl();
$scope.urlhost = $location.host();
$scope.urlport = $location.port();
$scope.urlprotocol = $location.protocol();
});
</script>
</head>
<body>
<div ng-app="locationApp" ng-controller="locationCtrl">
<p>Url of Webpage: {{weburl}}</p>
<p>Host of Webpage: {{urlhost}}</p>
```

```
<p>Port of Webpage: {{urlport}}</p>
<p>Protocol of Webpage: {{urlprotocol}}</p>
</div>
</body>
</html>
```

Output

Url of Webpage: http://localhost:3036/AngularSamples/AngularjsSamples.aspx

Host of Webpage: localhost

Port of Webpage: 3036

Protocol of Webpage: http

$timeout

The $timeout service can be used to call another JavaScript function after a given time delay. The$timeout service only schedules a single call to the function. For repeated calling of a function, see $interval.

Scheduling a Function Call

Once the $timeout service is injected into your controller function, you can use it to schedule function calls. Here is an example that used the $timeout service to schedule a function call 3 seconds later:

Example

```
var myapp = angular.module("myapp", []);
myapp.controller("MyController", function($scope, $timeout){
   $timeout(callAtTimeout, 3000);
});
function callAtTimeout() {
   console.log("Timeout occurred");
}
```

This example schedules a function call to callAtTimeout() after 3 seconds (3000 milliseconds).

Methods

flush([delay]);

- Flushes the queue of pending tasks.

verifyNoPendingTasks();

- Verifies that there are no pending tasks that need to be flushed.

Parent, child, sibling controllers

The controller's structure is set by the html tree. When using the dot notation in javascript they all get created equally on the apps module.
Example

```
<div ng-conntroller="mainctrl">
        <div ng-controller="childctrl"></div>
</div>
<div ng-controller="sibctrl">
</div>
```

- The above example has three controllers
- mainctrl and sibctrl both are sibilings
- childctrl is child of mainctrl

Root Scope

All application have a $rootScope .
It is the scope created on the HTML element that contains the 'ng-app' directive . Hence , it is available in the entire application.
If a variable has the same name in both the current scope and in the rootScope , the application use the one in the current scope.

Example

```
<body ng-app="myApp">
        <p>The rootScope's favourite color .</p>
         <h1>{{Color}}</h1>                          //blue

        <div ng-controller="myCtrl">
            <p>The scope of the controller's favourite color:</p>
            <h1>{{Color}}</h1>                        //red
        </div>

        <p>The rootScope's favourite color is still:</p>
        <h1>{{Color}}</h1>                          //blue
```

```
</body>

<script>
      Var app=angular.module("myApp",[]);
      app.run(function($rootScope){
        $rootScope.Color="blue";
       });
      app.controller("myCtrl",function($scope){
            $scope.Color="red";
         });
</script>
```

$Parent

It is one of the property of $scope .This property is used to refer to the parent scope from child controller.

Example

For a nested controller,the HTML markup looks as follows.

```
<div ng-controller="parentCtrl">
        <div ng-controller="childCtrl">
              {{$parent.City}}                    //Chennai
         </div>
</div>
```

The script looks as follows.

```
var app=angular.module("myApp",[]);

app.controller("parentCtrl",function($scope){
    $scope.City="Chennai";
});
```

```
app.controller("childCtrl",function($scope){
    $scope.place=$scope.$parent.City
});
```

$Broadcast,$emit and $on

AngularJS applications may need to communicate across the controllers .Such a communication might be needed to send notifications or to pass data between the controllers. When it comes to communicating between two or more AngularJS controllers there can be two possible arrangements.

- Controllers under consideration are nested controllers .That means they have parent-child relationship.
- Controllers under consideration are sibling controllers. That means they are at the same level without any parent-child relationship.

Nested Controllers

Three methods namely $broadcast, $emit and $on facilitate event-driven model for sending passing data between nested controllers .

- $broadcast and $emit allows to raise an event in the AngularJS application.
- $broadcast sends the event from the current controller to all of its child conntrollers. It would look like,

```
$scope.$broadcast("MyEvent",data);
```

Whereas, $emit sends the events upwards i.e, from current controller to all of its parent controllers.
For example,

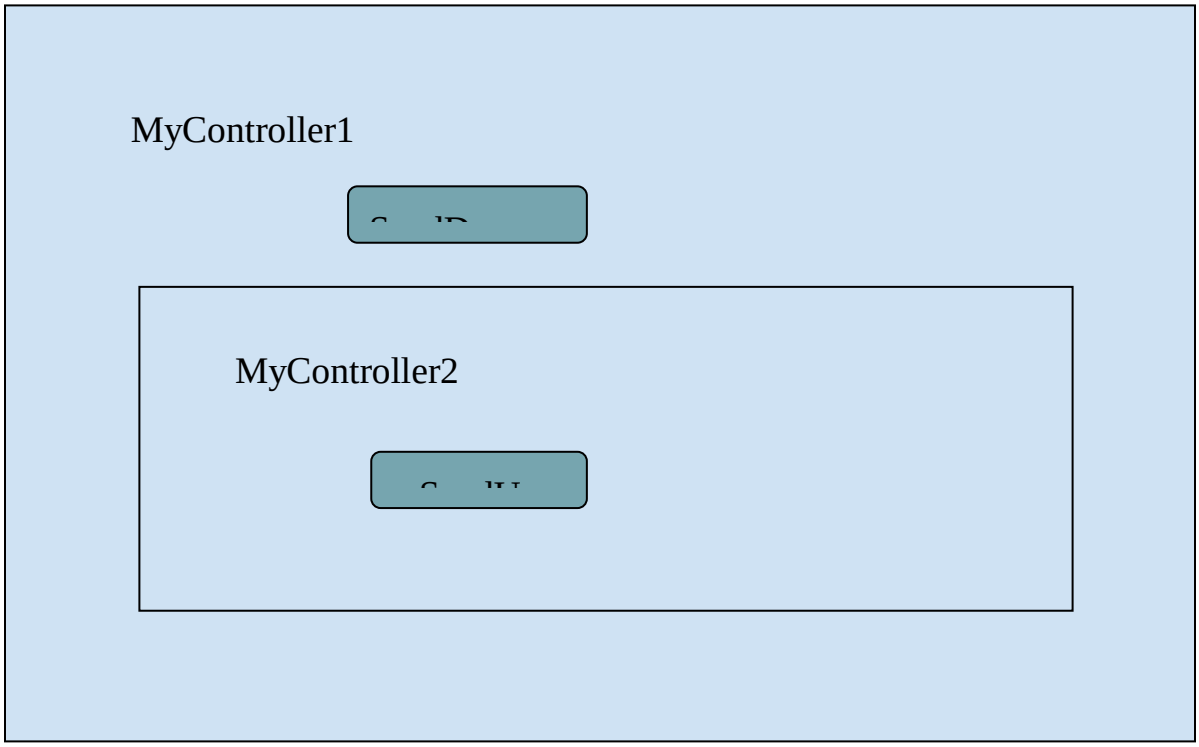
```
$scope.$emit("MyEvent",data);
```

Here,"MyEvent" is the user-defined event and the optional data parameter can be any type of data to be passed when "MyEvent" is dispatched by the system. An event raised by $broadcast and $emit can be handled by wiring an event handler using $on method. A typical call to $on() will look like this:

```
$scope.$on("MyEvent",function(evt,data){
        //Handler code here
```

```
});
```

In case of nested controllers,$broadcast ,$emit and $on works as follows.



The above application consists of two controllers-MyController1 and MyController2 nested inside each other.The outermost (MyController1) has a button for raising an event handled “SentDown”.Similarly, the innermost (MyController2) has a button for raising an event named “SentUp”.Both the events send a string data “some data” when the corresponding event is dispatched.

Now, the script would like as follows.

```
app.controller(“MyController1”,function($scope){
    //broadcast the event down
    $scope.OnClick=function(evt){
        $scope.$broadcast(“SendDown”,”some data”);
    }
     $scope.$on(“SendDown”,function(evt,data){
            $scope.Message=”The message from the  same controller is:”+data;
```

```
    });
    $scope.$on("SendUp",function(evt,data){
            $scope.Message="The message from the  child controller is:"+data;
        });
});

app.controller("MyController2",function($scope){
   //emit the event upwards
    $scope.OnClick=function(evt){
        $scope.$emit("SendUp","some data");
      }
```

The HTML markup of the page will look like this:

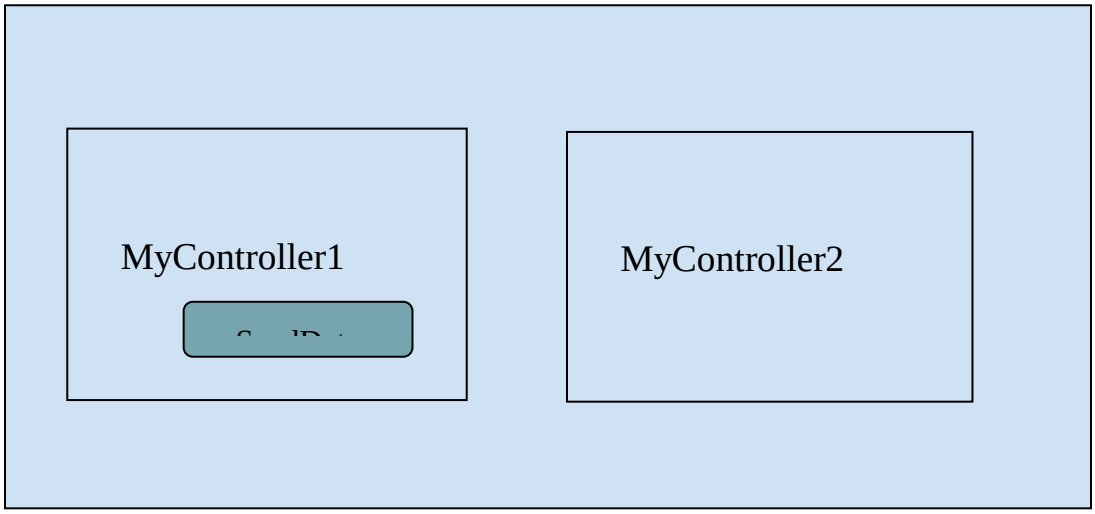
```
<body ng-app="MyApp">
        <div ng-controller=MyController1">
                <input type="button" value="Broadcast Down"
                                ng-click="OnClick($event)"/>
            <h4>{{Message}}</h4>
              <div ng-controller=MyController2">
               <input type="button" value="Emit  Down"
                                ng-click="OnClick($event)"/>
            <h4>{{Message}}</h4>
        </div>
   </div>
</body>
```

Thus, $broadcast and $emit facilitates event publisher-subscriber model for nested controllers i.e, work under both parent-child and ancestor-grandchild scenario.

Sibling Controllers

In case of sibling controllers no parent-child relationship exists so $rootScope has to be used instead of $scope .That's because, each controller will have its own scope and there won't be any inherited scope. Moreover, $emit won't serve much purpose in case of sibling controllers.This is because $rootScope is a container scope for all other scopes and can dispatch events only to its child controllers.So,only $broadcast() will serve some useful purpose in this case.

For example,



In above example, the script would look like this.

```
app.controller("MyController1",function($rootScope){
//broadcast the event down
$scope.OnClick=function(evt){
                $rootScope.$broadcast("SendDown","some data");
        }
         $scope.$on("SendDown",function(evt,data){
                $scope.Message="The message from the  same controller
                        is:"+data;
                 });
});

app.controller("MyController2",function($scope){
            $scope.$on("SendDown",function(evt,data){
                $scope.Message="The message inside the  MyController3 is:"+data;
                    });
```

```
});
```

Custom Directive

Directives are the most important components of any AngularJS application. Directives are markers on a DOM element that tell angularjs HTML compiler to attach specific behaviour to that element. We know the use of some commonly used built-in directives like ng-app, ng-controller, ng-repeat, ng-show, ng-hide and so on. All these directives attach special behaviour to DOM elements.

Although AngularJS ships with a wide range of directives, you will often need to create application-specific directives since some of the requirements in the application requirements vary. The custom directive helps the developer in creating a reusable component that can be maintained easily.

Angularjs has its own way of manipulating the DOM element, it is not the same way as in jQuery. It is the directive where we need to do the DOM Manipulation. Don't mix up jQuery with angularjs it will create a mess up code which is not a good practice.

Custom directives are used in AngularJS to extend the functionality of HTML. Custom directives are defined using the "directive" function. A custom directive simply replaces the element for which it is activated. An AngularJS application during bootstrap finds the matching elements defined in the custom directive and does a one-time activity using its compile() method of the custom directive then processes the element using the link() method of the custom directive based on the scope of the directive. AngularJS provides support to create custom directives for the following types of elements.

AngularJS Directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tells the AngularJS HTML compiler ($compile) to attach a specified behaviour to that DOM element or even transform the DOM element and its children.

Custom directives in AngularJS is your own directive with their own core functions that run when the DOM is compiled by the compiler. Suppose we want to reuse a specific lines of code in multiple pages of our application without code duplication, then we simply put that specific piece of code in a separate file and call that code using a custom directive instead of writing it over and over again. This way code is better to understand.
Let us look at the syntax of creating a custom directive as in the following:

```
myModule.directive("directiveName", function(injectables component)
{
   return
   {
      restrict: "A/E/C/M", //Attribute Element Class Comment
      template: "<div></div>", // Html to be added
      templateUrl: "directive.html", // template to be added
      replace: false, // replace yes/No
      transclude: false, // Transclude yes/No
      scope: false, // Current scope or isolated scope
      require: ["someOtherDirective"], // if any other directive is needed
controller: function($scope, $element, $attrs, $transclude,otherInjectables)
      {...   }, // controller
      link: function postLink(scope, iElement, iAttrs) {...
      },
      priority: 0, //Priority of directive
      terminal: false,
      compile: function compile(tElement, tAttrs, transclude)
      {
         return
         {
            pre: function preLink(scope, iElement, iAttrs, controller) {...
            },
            post: function postLink(scope, iElement, iAttrs, controller)
            {...
            } // compile functions
         }
      }
   };
});
```

When writing a custom directive we don't need all the parameters, we need some of them depending on our needs. Let us understand what these parameters are and what role they play in custom directives.

restrict: Specifies how a directive can be used. Possible values are: E (element), A (Attribute), C (Class) and M (Comment). The default value is A. This provides a way to specify how a directive should be used in HTML (remember a directive can appear in four ways). In this case we have set it to "AE". So, the directive can be used as a new HTML element or an attribute. To allow this directive to be used as a class we can set restrict to "AEC".

template: HTML template to be rendered in the custom directive. This specifies the HTML markup that will be produced when the directive is compiled and linked by Angular codes. This does not need to be a simple string. The template can be complex, often involving other directives, expressions ({{ }}) and so on. In most cases you want to use templateUrl instead of template. So, ideally you should place the template in a separate HTML file and make templateUrl point to it.

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. The default value is false.

transclude: Boolean value that says if the directive should preserve the HTML specified inside the directive element after rendering. The default is false. Let's understand a bit more on transclude.

Example of transclude

Consider a directive called myDirective in an element and that element is enclosing some other content, let's say:

```
<div my-directive>
  <button>some button</button>
  <a href="#">and a link</a>
</div>
```

If myDirective is using a template, you'll see that the content of <div my-directive> will be replaced by your directive template. So having:

```
app.directive('myDirective', function()
{
   return
   {
```

```
        template: '<div class="something"> This is my directive content</div>'
    }
});
```

This will result in this render:

```
<div class="something"> This is my directive content</div>
```

Observe here, the content of your original element <div my-directive> will be lost.

```
        <button>some button</button>
        <a href="#">and a link</a>
```

So, what if you want to keep your page as in the following:

```
        <button>some button</button>
        <a href="#">and a link</a>
```

You'll need something called transclusion.

Include the content from one place into another. So now your directive will look something like this:

```
app.directive('myDirective', function()
{
    return
    {
        transclude: true,
     template: '<div class="something" ng-transclude> This is my directive content</div>'
    }
});
```

This would render:

```
<div class="something"> This is my directive content
  <button>some button</button>
  <a href="#">and a link</a>
</div>
```

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

By default, a directive gets the parent's scope. But we don't want that in all cases. If we are exposing the parent controller's scope to the directives, they are free to modify the scope properties. In some cases your directive may want to add several properties and functions to the scope that are for internal use only. If we are doing these things to the parent's scope, we are polluting it. So, we have the following two other options:

- A child scope: This scope prototypically inherits the parent's scope.
- An isolated scope: A new scope that does not inherit from the parent and exists on its own.

In Angular, this binding can be done by setting attributes on the directive element in HTML and configuring the scope property in the directive definition object.

Let's explore a few ways of setting up the binding.

Option 1: Use @ for One Way Text Binding

Binds the value of parent scope property (that is always a string) to the local scope. So the value you want to pass in should be wrapped in {{}}.

Option 2: Use = for two-way binding and binds the Parent scope property directly that will be evaluated before being passed in.

Option 3: Use and to Execute functions in the parent scope and binds an expression or method that will be executed in the context of the scope it belongs in.

```
app.directive('helloWorld', function()
{
   return
   {
      scope: {
         Name: '='
         Name: '@'
         Name: '='
      },
      // other codes go here
   };
});
```

By default a directive does not create a new scope and uses the parent's scope. But in many cases this is not what we want. If your directive manipulates the parent scope properties

heavily and creates new ones, it might pollute the scope. Letting all the directives use the same parent scope is not a good idea because anybody can modify our scope properties. So, the following guidelines may help you choose the right scope for your directive.

Parent Scope (scope: false): This is the default case. If your directive does not manipulate the parent scope properties you might not need a new scope. In this case, using the parent scope is okay.

Child Scope (scope:true): This creates a new child scope for a directive that prototypically inherits from the parent scope. If the properties and functions you set on the scope are not relevant to other directives and the parent, you should probably create a new child scope. With this you also have all the scope properties and functions defined by the parent.

Isolated Scope (scope:{}): This is like a sandbox! You need this if the directive you will build is self-contained and reusable. Your directive might be creating many scope properties and functions that are meant for internal use and should never be seen by the outside world. If this is the case, it's better to have an isolated scope. The isolated scope, as expected, does not inherit the parent scope.

require: A list of directives that the current directive needs. The current directive gets access to controller of the required directive. An object of the controller is passed into the 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 the priority of a directive. The default value is 0. A directive with a higher priority value is executed before a directive with a 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 the core logic of the directive. It is executed after the directive is compiled. Gets access to the scope and 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 the scope since the scope is not created yet. Gets an object of the element and attributes. Used to perform the DOM of the directive before the templates are compiled and before the directive is transcluded. It returns an object with the following 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. Used rarely. One of the use cases is when a child directive requires data from its parent, the parent directive should set it through its pre-link function. Set data required for its child directives. Safe to attach event handlers to the DOM element. Not safe to access DOM elements belonging to child directives. They're not linked yet.

post link: This is the most commonly used for data binding. Safe to attach event handlers to the DOM element. All children directives are linked, so it's safe to access them. Never set any data required by the child directive here, because the child directive's will be linked already.

Compilation in directives

When the application bootstraps, Angular starts parsing the DOM using the $compile service. This service searches for directives in the mark-up and matches them against registered directives. Once all the directives have been identified, Angular executes their compile functions. As previously said, the compile function returns a link function that is added to the list of link functions to be executed later.

This is called the compile phase. If a directive needs to be cloned multiple times (for example ng-repeat), we get a performance benefit since the compile function runs once for the cloned template, but the link function runs for each cloned instance. That's why the compile function does not receive a scope.

After the compile phase is over, next is the linking phase, where the collected link functions are executed one by one. This is where the templates produced by the directives are evaluated against correct scope and are turned into a live DOM that reacts to events.

Let's do some simple samples on custom directives now.

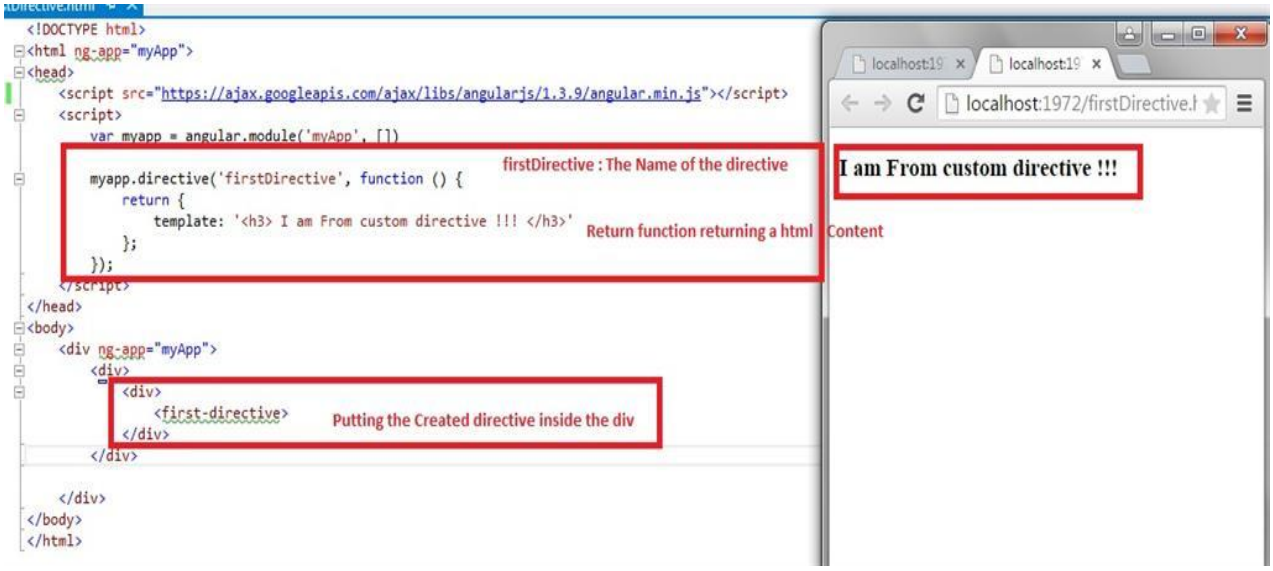
Example-1

We will create a directive called "firstDirective" that will return us a sample text.

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.3.9/angular.min.js"></script><script>
var myapp = angular.module('myApp', [])

myapp.directive('firstDirective', function () {
        return {
template: '<h3> I am From custom directive !!! </h3>'
};

});
</script>
</head>
<body>
<div ng-app="myApp">
        <div>
          <div>
            <first-directive>
          </div>
        </div>
    </div>
</body>
</html>
```

Note: "firstDirective" name must be a camel case while declaring a directive and while calling it should be written as <first-directive> . There are some other way of calling it as in the following:
<first_directive> or <first:directive>

Output



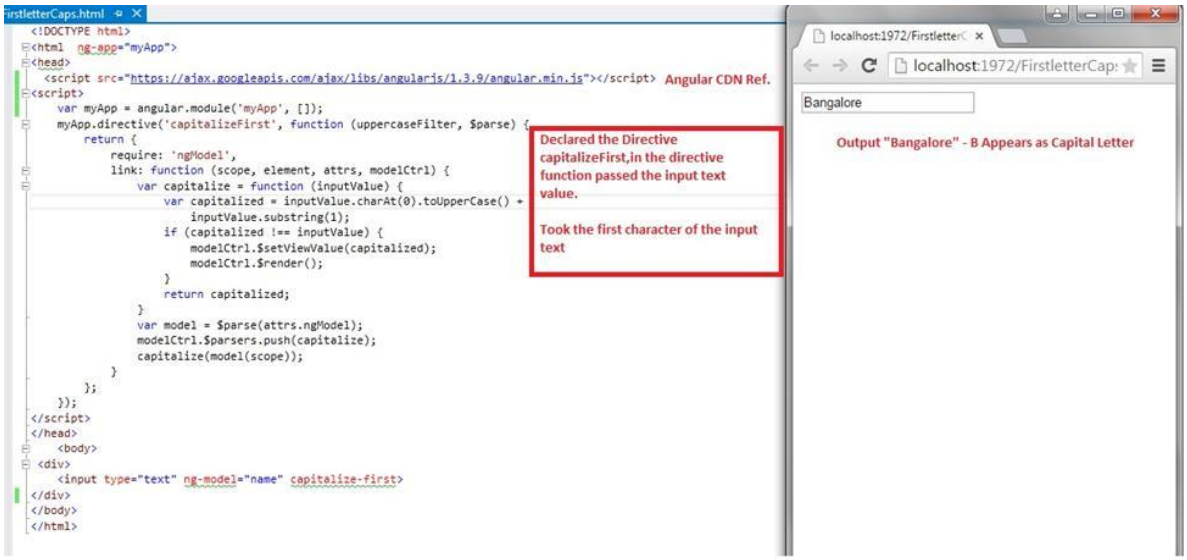
Example-2

We will create a directive that will convert the first letter of words to uppercase.

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.3.9/angular.min.js"></script>
<script>
var myApp = angular.module('myApp', []);
myApp.directive('capitalizeFirst', function (uppercaseFilter, $parse) {
return {
require: 'ngModel',
link: function (scope, element, attrs, modelCtrl) {
var capitalize = function (inputValue) {
var capitalized = inputValue.charAt(0).toUpperCase() +
inputValue.substring(1);
if (capitalized !== inputValue) {
modelCtrl.$setViewValue(capitalized);
modelCtrl.$render();
}
return capitalized;
}
var model = $parse(attrs.ngModel);
```

```
modelCtrl.$parsers.push(capitalize);
capitalize(model(scope));
}
};
});
</script>
</head>
<body>
<div>
        <input type="text" ng-model="name" capitalize-first>
    </div>
</body>
</html>
```

Output



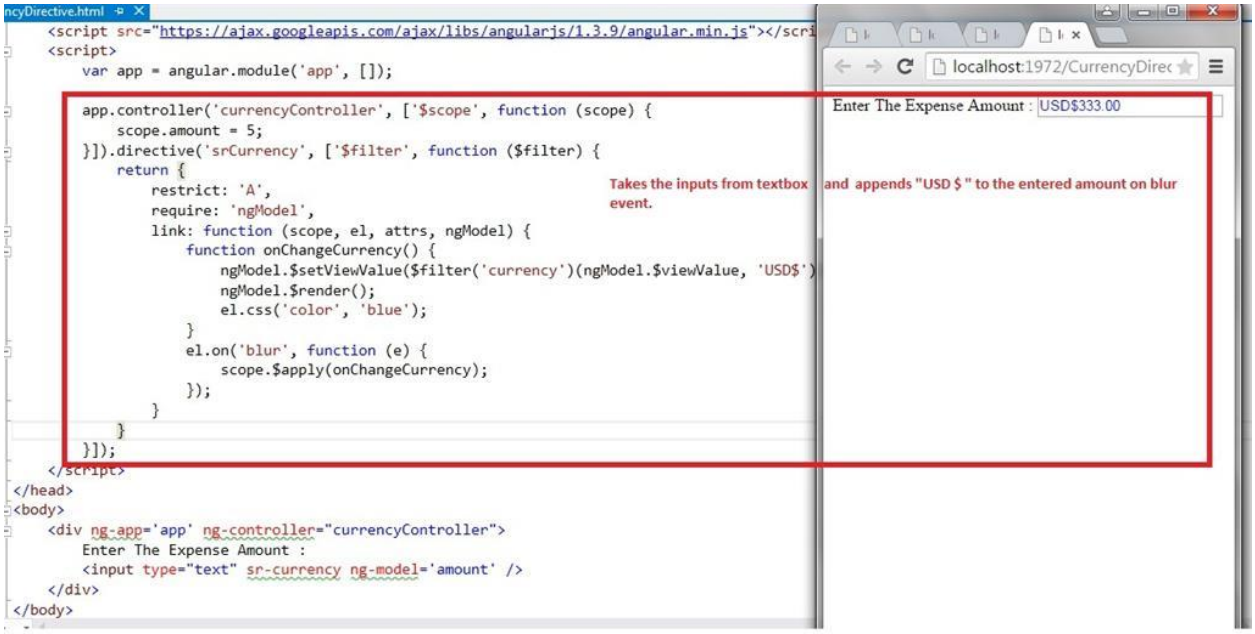
Example 3

In the third example we will create a directive that will define some extra “USD” to text on the blur event.

```
<!DOCTYPE html>
<html ng-app="app">
```

```
<head>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.3.9/angular.min.js"></script>
<script>
var app = angular.module('app', []);
app.controller('currencyController', ['$scope', function (scope) {
scope.amount = 5;
}]).directive('srCurrency', ['$filter', function ($filter) {
return {
restrict: 'A',
require: 'ngModel',
link: function (scope, el, attrs, ngModel) {
function onChangeCurrency() {
ngModel.$setViewValue($filter('currency')(ngModel.$viewValue, 'USD$'));
ngModel.$render();
el.css('color', 'blue');
}
el.on('blur', function (e) {
scope.$apply(onChangeCurrency);
});
}
}
}]);
</script>
</head>
<body>
   <div ng-app='app' ng-controller="currencyController">
         Enter The Expense Amount :
<input type="text" sr-currency ng-model='amount' />
    </div>
</body>
</html>
```

Output



Example 4

Let us do a directive that will read the value from the scope object and show it in a tabular format.

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
<style type="text/css">
table {
color:#666;
text-shadow: 1px 1px 0px #fff;
background:#eaebec;
margin:20px;
border:#ccc 1px solid;
-moz-border-radius:3px;
-webkit-border-radius:3px;
border-radius:3px;
-moz-box-shadow: 0 1px 2px #d1d1d1;
-webkit-box-shadow: 0 1px 2px #d1d1d1;
box-shadow: 0 1px 2px #d1d1d1;
}
table th {
padding:21px 25px 22px 25px;
border-top:1px solid #fafafa;
border-bottom:1px solid #e0e0e0;
background: #ededed;
```

```
background: -webkit-gradient(linear, left top, left bottom, from(#ededed), to(#ebebeb));
background: -moz-linear-gradient(top, #ededed, #ebebeb);
}
table th:first-child {
text-align: left;
padding-left:20px;
}
table tr:first-child th:first-child {
-moz-border-radius-topleft:3px;
-webkit-border-top-left-radius:3px;
border-top-left-radius:3px;
}
table tr:first-child th:last-child {
-moz-border-radius-topright:3px;
-webkit-border-top-right-radius:3px;
border-top-right-radius:3px;
}
table tr {
text-align: center;
padding-left:20px;
}
table td:first-child {
text-align: left;
padding-left:20px;
border-left: 0;
}
table td {
padding:18px;
border-top: 1px solid #ffffff;
border-bottom:1px solid #e0e0e0;
border-left: 1px solid #e0e0e0;
background: #fafafa;
background: -webkit-gradient(linear, left top, left bottom, from(#fbfbfb), to(#fafafa));
background: -moz-linear-gradient(top, #fbfbfb, #fafafa);
}
table tr.even td {
background: #f6f6f6;
background: -webkit-gradient(linear, left top, left bottom, from(#f8f8f8), to(#f6f6f6));
background: -moz-linear-gradient(top, #f8f8f8, #f6f6f6);
}
table tr:last-child td {
border-bottom:0;
}
table tr:last-child td:first-child {
-moz-border-radius-bottomleft:3px;
-webkit-border-bottom-left-radius:3px;
border-bottom-left-radius:3px;
}
```

```
table tr:last-child td:last-child {
-moz-border-radius-bottomright:3px;
-webkit-border-bottom-right-radius:3px;
border-bottom-right-radius:3px;
}
table tr:hover td {
background: #f2f2f2;
background: -webkit-gradient(linear, left top, left bottom, from(#f2f2f2), to(#f0f0f0));
background: -moz-linear-gradient(top, #f2f2f2, #f0f0f0);
}
.odd {
color: red;
}
.even {
color: blue;
}
</style>
<script src="scripts/angular.min.js"></script>
<script>
var myapp = angular.module('myApp', [])
.controller('countryctrl', ['$scope', function ($scope) {
$scope.data = [{
id: 1,
name: 'India',
capital: 'New Delhi',
curreny:"INR"
   }, {
id: 2,
name: 'USA',
capital: 'New York',
curreny: "USD"
   }, {
id: 3,
name: 'China',
capital: ' Beijing ',
curreny: "Renminbi"
   }, {
id: 4,
name: 'Japan',
capital: 'Tokyo',
curreny: "yen"
  }, {
id: 5,
name: 'Korea',
capital: 'Seoul',
curreny: "Won "
 }, {
id: 6,
```

```
name: 'Vietnam',
capital: 'Hanoi',
curreny: "dong"
 }
];
}])
.directive('trRow', function () {
return {
template: '
          <tr>
             <td ng-bind="row.id"></td>
             <td>
                <strong ng-bind="row.name"></strong>
             </td>
             <td ng-bind="row.capital"></td>
             <td ng-bind="row.curreny"></td>
          </tr>'
};
});
</script>
</head>
<body>
<div ng-app="myApp">
         <table ng-controller="countryctrl">
             <thead style="color:red; ">
                <th>Country Id</th>
                <th>Country Name</th>
                <th>Capital</th>
                <th>Currency</th>
             </thead>
             <tbody>
                <tr tr-row ng-repeat="row in data"></tr>
             </tbody>
         </table>
    </div>
</body>
</html>
```

Output

Country Id	Country Name	Capital	Currency
1	India	New Delhi	INR
2	USA	New York	USD
3	China	Beijing	Renminbi
4	Japan	Tokyo	yen
5	Korea	Seoul	Won
6	Vietnam	Hanoi	dong

These are some basic directive examples to start custom directives. In an application, if you observe, there can be many reusable UI components that we can make it a custom directive and make it usable across the team to use it.

CUSTOM FILTERS

Filters are used to modify or transform the format of data and can be clubbed in expression or directives using pipe character.
In spite of having pre-defined filters, it is also possible to create a customized filter and bind it to the view.

Syntax

// To declare a filter we pass in two parameters to app.filter
// The first parameter is the name of the filter
// second is a function that will return another function that does the actual work of the filter
// In the return function, we must pass in a single parameter which will be the data we will work on.
// We have the ability to support multiple other parameters that can be passed into the filter optionally

```
app.filter('myFilter', function()
{
 return function(input, optional1, optional2)
  {
        var output;
// Do filter work here
       return output;

   }
}
```

Example

```
<!DOCTYPE html>
<html>
<body ng-app="myapp">
```

```
                <li>{{example1 | capitalize}} - Default - Capitalizes the first letter</li>
                <li>{{example1 | capitalize:5}} - Capitalizes the 5th letter</li>
                <li>{{example2 | capitalize}} - Default - If first, letter is capital, does
nothing</li>
  <script>
 var app = angular.module('filters', []);

app.controller('demo', function($scope){
 $scope.example1 = "hello vernalis";
 $scope.example2 = "vernaliscompany";
})

app.filter('capitalize', function()
{
 return function(input, char){
        if(isNaN(input)){
        var char = char - 1 || 0;
        var letter = input.charAt(char).toUpperCase();
        var out = [];
        for(var i = 0; i < input.length; i++){
        if(i == char){
        out.push(letter);
        } else {
        out.push(input[i]);
        }
        }
        return out.join('');
        } else {
        return input;
        }
 }
})

</script>
</body>
</html>
```

CUSTOM SERVICES

The purpose of AngularJS service / factory function is to generate a single object or function that represents the service to rest of the application. That object or function is passed as a parameter to any other factory function which specifies a dependency on this service. AngularJS provides many inbuilt services for example, $http, $route, $window, $location etc. Inbuilt services are always prefixed with $ symbol.
There are two ways to create a service.

- factory
- Service

Syntax:
SERVICE:

```
module.service('MyService',function(){

        this.method1=function(){ //..}

        this.method1=function(){ //..}

    });
```

FACTORY:

```
module.factory('MyFactory',function(){
Var factory={};

factory .method1=function(){ //..}

factory.method2=function(){ //..}

return factory;
    });
```

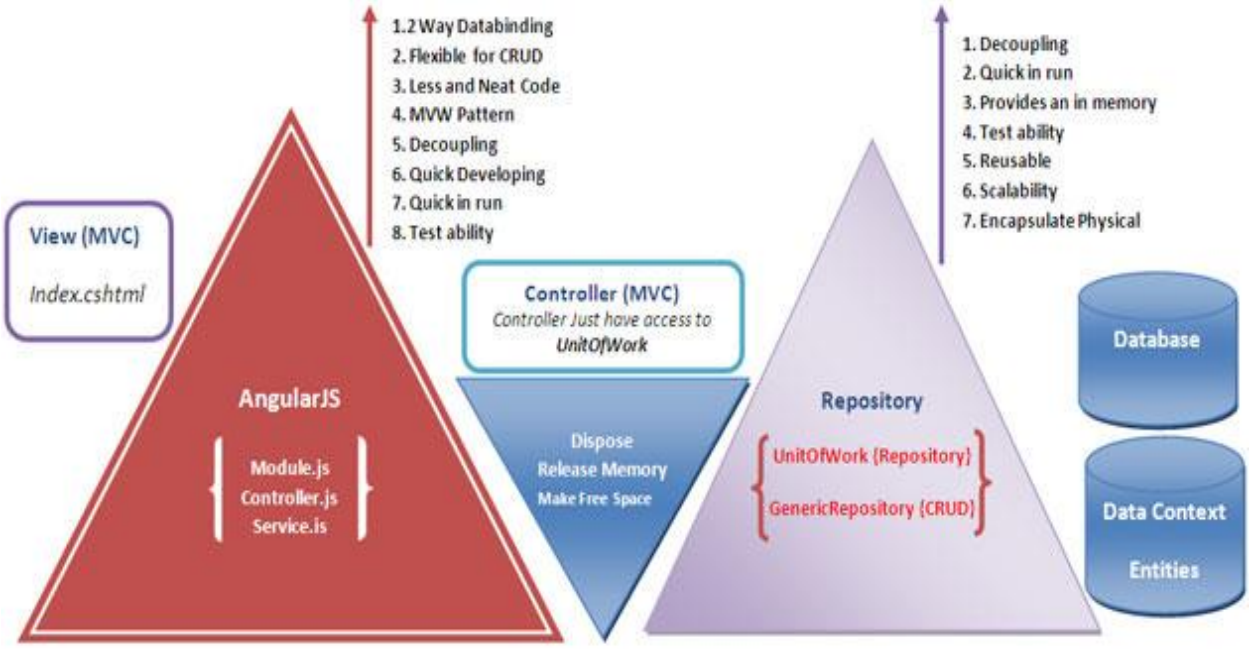
Example

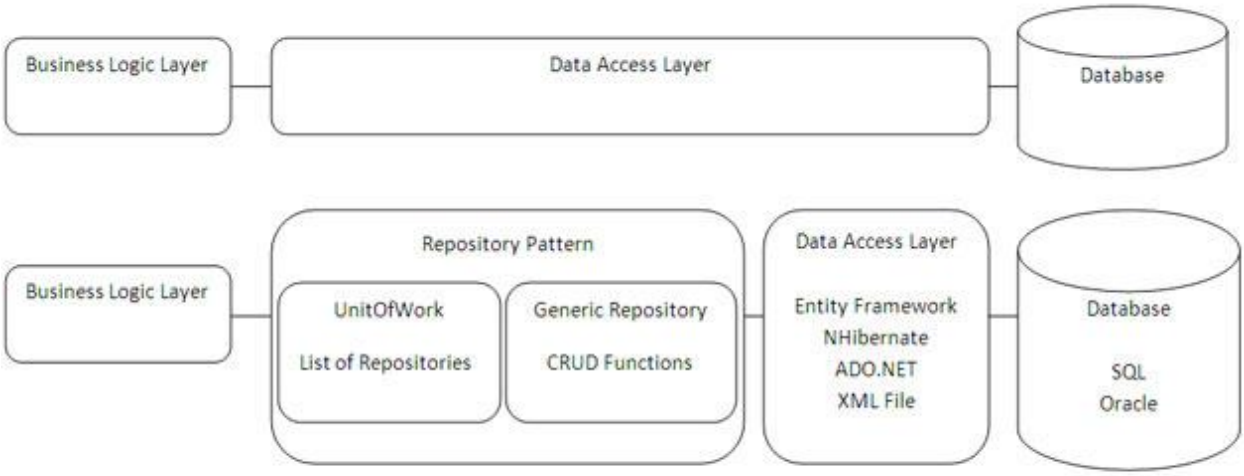
```
<!DOCTYPE html>
<html>
<body>
<div ng-app="myApp" ng-controller="myCtrl">

<p>The service method to find square(4) is:</p>

<h1>{{hex1}}</h1>
        <p>The factory method to find cube(4) is:</p>
<h1>{{hex2}}</h1>

</div>



<script>
var app = angular.module('myApp', []);

app.service('MyService', function() {
        this.myFunc = function (x) {
        return x*x;
        }
});

app.factory('MyFactory',function(){
return{

        myFunc:function(x){
                        return x*x*x;
                        }
        }

});

app.controller('myCtrl', function ($scope, MyService, MyFactory) //injecting module
 {
        $scope.hex1 = MyService.myFunc(4);
        $scope.hex2= MyFactory.myFunc(4);
```

```
});
</script>

</body>
</html>
```

Repository

Repository is one of the useful design pattern which hides physical data from the rest of the other sections of data access layers. professionally speaking Repository give us an instruction to how encapsulates our data context from other database queries and data access logic. It is such as a line which isolates actual database from other parts, eventually it makes our web application more convenient to test it layer by layer.





Example: Using Angular JS, we can load a list of customers by binding the data to customer variable.

JS Code for Controller

```
angular.module ('CMApplication.Controllers')
      .controller ('CustomerController', ['$scope', 'CustomerRepository',
           function ($scope, CustomerRepository) {
                CustomerRepository.getCustomers (function (data) {
                     $scope.customers = data;
         });
}]);
```

This controller code initiates the call to fetch data when the script files is included in the page which is rendered. getCustomers function has the function definition in the factory a.k.a repository. The code for repository depicted below.

JS Code for Factory (A) Repository

```
angular.module ('CMApplication.Repositories')
          .factory ('CustomerRepository', ['$http', function ($http) {
           return {
```

```
            getCustomers: function (callback) {
                $http.get ('Database/Customers.json').success (function (data) {
                callback (data);
                });
        }
        };
}]);
```

This factory method initiates a http service call to the JSON file which has the customer details. When the request is successful, the function gets acknowledged and receives the JSON data. This response is stored in the scope variable 'customers'. And further the scope variable can be used in the HTML to display the data contained in the variable.

JSON file that contains Customer Details

```
[
   { "id": 1, "name": "Abcd" },
   { "id": 2, "name": "Xyz" },
   { "id": 3, "name": "Customer Name 3" }
]
```

The above is the example JSON data. This JSON consists of Customer ID and Customer Name.

HTML Code

```
<h3>Customers</h3>

<div ng-repeat="cust in customers">
   <div>ID: {{cust.id}} - Name: {{cust.name}}</div>
</div>
```

In the above HTML code, ng-repeat is used to iterate through the variable customers using another variable cust. For every data that is available in customers, ng-repeat keeps assigning values to cust and keeps displaying the values.

Restangular

Restangular is an AngularJS service which makes GET/POST/PUT/DELETE requests simpler and easier. The configuration is easier and can be done in several ways with different initiation settings.

Note: Restangular depends on either lodash or underscore, so in order support restangular, we'll need to make sure we include one of the two.

```
<script type="text/javascript"
src="//cdn.jsdelivr.net/lodash/2.1.0/lodash.compat.min.js">
</script>
```

Also, just like any other AngularJS library, we'll need to include the restangular resource as a dependency for our app module object:

```
angular.module('myApp', ['restangular']);
```

Once we've done that, we'll be able to inject the Restangular service in our Angular objects:

```
angular.module('myApp')
.controller('MainController', ['$scope', 'Restangular',
  function($scope, Restangular) {
 }]);
```

To use Restangular, there are two ways we can create an object to fetch services. We can either set the base route to fetch objects from:

```
var User = Restangular.all('users');
```

This will set all HTTP requests that come from the Restangular service to pull from /users. For instance, calling getList() on the above object will fetch from /users:

```
var allUsers = User.getList(); // GET /users
```

It's also possible to fetch nested requests for a single object. Instead of passing only a route, we can pass a unique ID to pull from as well:

```
var oneUser = Restangular.one('users', 'abc123');
```

This will generate the request /users/abc123 when calling getList() on it.

```
oneUser.get().then(function(user) {
  // GET /users/abc123/inboxes
  user.getList('inboxes');
});
```

We can also create messages using the Restangular object. To create an object, we'll use the post() method. The post method requires a single object as a parameter and will send a POST request to the URL we specified. We can also add queryParameters and headers to this request.

```
// POST to /messages
var newMessage = {
  body: "Hello world"
};
messages.post(newMessage);
// OR we can call this on an element to create a nested
// resource for the element
var message = Restangular.one('messages', 'abc123');
message.post('replies', newMessage);
```

Because Restangular returns promises, we can then call methods on the returned data on promises so that we can run a function after the promise has completed. For instance, after we update a collection, we can then refresh the collection on our scope:

```
messages.post(newMessage).then(function(newMsg) {
  $scope.messages = messages.getList();
}, function error(reason) {
  // An error has occurred
});
```

We can also remove an object from the collection. Using the remove() method, we can send a DELETE HTTP request to our back end. To send a delete request, we can call the remove() method on an object inside the collection (an element).

```
var message = messages.get(123);
message.remove(); // Send a DELETE to /messages
```

Setting the baseUrl

To set the baseUrl for all calls to our backendAPI, we can use the setBaseUrl() method. For instance, if our API is located at /api/v1, rather than at the root of our server.

```
angular.module('myApp')
  .config(function(RestangularProvider) {
    RestangularProvider.setBaseUrl('/api/v1');
  });
```

Custom Restangular services

Finally, we highly recommend using Restangular from inside a custom service object. This is particularly useful, as we can configure Restangular on a per-service level using a service as well as disconnecting the logic to talk to our back end from within our controllers/directives and enabling our services to handle talking to them directly.

To create a service that wraps Restangular, we simply need to inject the Restangular service in our factory and call the methods like normal. Inside this factory, we can create custom configurations by using the withConfig() function.

```
angular.module('myApp', ['restangular'])
.factory('MessageService', [
  'Restangular', function(Restangular) {
    var restAngular =
      Restangular.withConfig(function(Configurer) {
        Configurer.setBaseUrl('/api/v2/messages');
    });
    var _messageService = restAngular.all('messages');
     return {
```

```
    getMessages: function() {
      return _messageService.getList();
    }
  }
}]);
```

Routing

If you want to navigate to different pages in your application, but you also want the application to be a SPA (Single Page Application), with no page reloading, you can use the ngRoute module.

The ngRoute module routes your application to different pages without reloading the entire application. The ngRoute module helps your application to become a Single Page Application.

Example
Navigate to "red.html", "green.html", and "blue.html":

```
<body ng-app="myApp">

<p><a href="#/">Main</a></p>

<a href="#red">Red</a>
<a href="#green">Green</a>
<a href="#blue">Blue</a>

<div ng-view></div>

<script>
var app = angular.module("myApp", ["ngRoute"]);
app.config(function($routeProvider) {
  $routeProvider
  .when("/", {
    templateUrl : "main.html"
  })
  .when("/red", {
    templateUrl : "red.html"
  })
```

```
    .when("/green", {
        templateUrl : "green.html"
    })
    .when("/blue", {
        templateUrl : "blue.html"
    });
});
</script>
```

To make your applications ready for routing, you must include the AngularJS Route module:

```
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular-route.js"></script>
```

Then you must add the ngRoute as a dependency in the application module:

```
var app = angular.module("myApp", ["ngRoute"]);
```

Now your application has access to the route module, which provides the $routeProvider. Use the $routeProvider to configure different routes in your application:

```
app.config(function($routeProvider) {
  $routeProvider
  .when("/", {
    templateUrl : "main.html"
  })
  .when("/red", {
    templateUrl : "red.html"
  })
  .when("/green", {
    templateUrl : "green.html"
  })
  .when("/blue", {
    templateUrl : "blue.html"
  });
});
```

Your application needs a container to put the content provided by the routing.

This container is the ng-view directive.

There are three different ways to include the ng-view directive in your application:

```
<div ng-view></div>
```

```
<ng-view></ng-view>
```

```
<div class="ng-view"></div>
```

Applications can only have one ng-view directive, and this will be the placeholder for all views provided by the route.

With the $routeProvider you can also define a controller for each "view".

```
var app = angular.module("myApp", ["ngRoute"]);
app.config(function($routeProvider) {
    $routeProvider
    .when("/", {
        templateUrl : "main.htm"
    })
    .when("/london", {
        templateUrl : "london.htm",
        controller : "londonCtrl"
    })
    .when("/paris", {
        templateUrl : "paris.htm",
        controller : "parisCtrl"
    });
});
app.controller("londonCtrl", function ($scope) {
    $scope.msg = "I love London";
});
app.controller("parisCtrl", function ($scope) {
    $scope.msg = "I love Paris";
});
```

Template
In the previous examples we have used the templateUrl property in the $routeProvider.when method.

You can also use the template property, which allows you to write HTML directly in the property value, and not refer to a page.

```
var app = angular.module("myApp", ["ngRoute"]);
app.config(function($routeProvider) {
    $routeProvider
    .when("/", {
        template : "<h1>Main</h1><p>Click on the links to change this content</p>"
    })
    .when("/banana", {
        template : "<h1>Banana</h1><p>Bananas contain around 75% water.</p>"
    })
    .when("/tomato", {
        template : "<h1>Tomato</h1><p>Tomatoes contain around 95% water.</p>"
    });
});
```

Since there is a disadvantage of having only one view in ngRoute, uiRoute is prefered.

UI-Router

The ui-router library is one of the most useful that the AngularUI library gives us. It's a routing framework that allows us to organize our interface by a state machine, rather than a simple URL route.

This library provides for a lot of extra control in our views. We can created nested views, use multiple views on the same page, have multiple views that control a single view, and more. For finer grain control and more complex applications, the ui-router is a great library to harness.

We need to link the library to our view:

```
<script type="text/javascript" src="app/bower_components/angular-ui-router/release/angular-ui-router.js"></script>
```

And we need to inject the ui.router as a dependency in our app:

```
angular.module('myApp', ['ui.router'])
```

Now, unlike the built-in ngRoute service, the ui-router can nest views, as it works based off states, not just a URL.

Instead of using the ng-view directive as we do with the ngRoute service, we'll use the ui-view directive with ngRoute.

When dealing with routes and states inside of ui-router, we're mainly concerned with which state the application is in as well as at which route the web app currently stands.

```
<div ng-controller="DemoController">
  <div ui-view></div>
</div>
```

Just like ngRoute, the templates we define at any given state will be placed inside of the <div ui-view> element. Each of these templates can include their own ui-view as well, which is how we can have nested views inside our routes.

To define a route, we use the .config method, just like normal, but instead of setting our routes on $routeProvider, we set our states on the $stateProvider.

```
.config(function($stateProvider, $urlRouterProvider) {
  $stateProvider
    .state('start', {
      url: '/start',
      templateUrl: 'partials/start.html'
    })
});
```

This step assigns the state named start to the state configuration object. The state configuration object, or the stateConfig, has similar options to the route config object, which is how we can configure our states.

We can set up templates on each of our views using one of the three following options:

- template – A string of HTML content or a function that returns HTML
- templateUrl – A string URL path to a template or a function that returns a URL path string
- templateProvider – A function that returns an HTML content string

Just like in ngRoute, we can either associate an already registered controller with a URL (via a string) or we can create a controller function that operates as the controller for the state.

If there is no template defined (in one of the previous options), then the controller will **not** be created.

The url option will assign a URL that the application is at to a specific state inside our app. That way, we can get the same features of deep linking while navigating around the app by state, rather than simply by URL.

This option is similar to the ngRoute URL, but can be considered a major upgrade, as we'll see in a moment.

We can specify the basic route like so:

```
$stateProvider
  .state('inbox', {
    url: '/inbox',
    template: '<h1>Welcome to your inbox</h1>'
  });
```

When the user navigates to /inbox, then the app will transition into the inbox state and fill the main ui-view directive with the contents of the template (<h1>Welcome to your inbox</h1>). The URL can take several different options, which makes it incredibly powerful. We can set the basic parameters in the URL like we do in ngRoute:

```
$stateProvider
  .state('inbox', {
    url: '/inbox/:inboxId',
    template: '<h1>Welcome to your inbox</h1>',
    controller: function($scope, $stateParams) {
      $scope.inboxId = $stateParams.inboxId;
    }
  });
```

Params
The params option is an array of parameter names or regexes. This option cannot be combined with the url option. When the state becomes active, the app will populate the $stateParams service with these parameters.

Views

We can set multiple named views inside of a state. This feature is a particularly powerful one in ui-router: Inside of a single view, we can define multiple views that we can reference inside of a single template.

```
<div>
  <div ui-view="filters"></div>
  <div ui-view="mailbox"></div>
  <div ui-view="priority"></div>
</div>
```

We can now create named views that fill each of these individual templates. Each of the subviews can contain their own templates, controllers, and resolve data.

```
$stateProvider
  .state('inbox', {
    views: {
      'filters': {
        template: '<h4>Filter inbox</h4>',
        controller: function($scope) {}
      },
      'mailbox': {
        templateUrl: 'partials/mailbox.html'
      },
      'priority': {
        template: '<h4>Priority inbox</h4>',
        resolve: {
          facebook: function() {
            return FB.messages();
          }
        }
      }
    }
  });
```

External Plugin in Angularjs

AngularJS allow the developer to add external plugin as injectables component,dependency through directive,controller or service. Modal dialogs is one of the external plugin for the angularjs .Modal dialogs and popups provider for Angularjs applications. ngDialog minimalistic API, highly customizable through themes and has only Angular.js as dependency.

You need only to include ngDialog.js, ngDialog.css and ngDialog-theme-default.css (as minimal setup) to your project and then you can start using the ngDialog provider in your directives, controllers and services. Define the class Name to be the ngDialog-theme-default. For example in controllers:

```
var app = angular.module('exampleApp', ['ngDialog']);
app.controller('MainCtrl', function ($scope, ngDialog) {
 $scope.clickToOpen = function () {
      ngDialog.open({ template: 'popupTmpl.html',
       className:'ngdialog-theme-default' });
       };
});
```