http://codewala.net/ http://fb.com/codewala https://twitter.com/code_wala

# AngularJS in 19 days

AngularJS v1.4, ASP.NET MVC and Visual Studio 2013

ABSTRACT

This book is a collection of 20 posts on AngularJS that got published on my blog (http://codewala.net). I refined a bit and added the samples at GIT for each day. This book is not just another book of AngularJS but it is a unique combination of Angular basics, detailed discussion on core Angular concepts, example driven and working it with ASP.NET MVC and Visual Studio. Although it is not mandatory but if you are a .NET Developer then you will enjoy this book more.

Brij Bhushan Mishra

# About the Author

## Brij Bhushan Mishra

**Microsoft MVP, Architect, Author Speaker**

Brij Bhushan Mishra is a 5 times Microsoft MVP, Author, Blogger, Programmer and Speaker. He is currently associated with an MNC as a Consultant/Architect in Microsoft technologies.
He regularly speaks in user group events and conferences on various technologies. He has written numerous articles for CodeProject, CSharpCorner, Infragistics and many other online platforms and regularly blogs at http://codewala.net. Many of his blog posts have been featured on the ASP.NET Microsoft official website and other various online communities. Other than his community activities, he loves exploring new places with his lovely wife and cute son.

### Awards

- Microsoft Most Valuable Professional (MVP) – 5 times in a row
- Former Code Project MVP
- C# Corner MVP
- Code Project Mentor/Insider and many more..

MVP Microsoft® Most Valuable Professional

MVP Most Valuable

MVP

Microsoft CERTIFIED Technology Specialist

.NET Framework 4, Web Applications
.NET Framework 4, Windows Communication Foundation Applications
.Net Framework 2.0, Web Applications

**Blog Address** - http://codewala.net

### Online Community Profiles

http://codewala.net/about/

http://www.codeproject.com/Members/BrijBhushanMishra

http://www.c-sharpcorner.com/authors/d551d3/brij-mishra.aspx

### Social Coordinates

https://twitter.com/code_wala http://fb.com/codewala

http://in.linkedin.com/in/brijbhushan https://goo.gl/Sd3nxe

# Table of Content

Day 1 .......... 5

AngularJS Getting Started .......... 5

Day 2 .......... 10

Modules and Directives .......... 10

Day 3 .......... 14

AngularJS with ASP.NET MVC .......... 14

Day 4 .......... 19

Services in AngularJS .......... 19

Day 5 .......... 23

Views and Routing .......... 23

Day 6 .......... 28

Views, Routing and Model .......... 28

Day 7 .......... 33

Data Binding in AngularJS .......... 33

Day 8 .......... 37

Data Binding – Under the Hood .......... 37

Day 9 .......... 43

Creating Custom Directive .......... 43

Day 10 .......... 48

Custom Directive with Isolate Scope .......... 48

Day 11 .......... 55

Passing Values function in Isolate Scope .......... 55

Day 12 .......... 59

Exploring Filters .......... 59

Day 13 .......... 64

Exploring Filters – Custom Filters .......... 64

Day 14 .......... 67

Using Filters in Controller, Factory, Service .......... 67

Day 15 .......... 71

Dependency Injection (DI) .......... 71

Day 16 .......... 77

Dependency Injection (DI) contd. ........ 77
Day 17 ........ 81
Dependency Injection Annotation Process ........ 81
Day 18 ........ 84
Getting Started with Unit Test ........ 84
Day 19 ........ 88
Writing Unit Test for Service, Custom Filter and Directives ........ 88

***Note – Download the complete code for all the days from GitHub.***

Web development is all about communication between client and server. The code we write for any web application executes either on client or server. The more code we execute on client machine, the faster web application will be. It also helps in providing rich UI and very smooth user experience. This is the key reason, our focus on Client side programming is taking over the server side programming and new tools and technologies are coming each day to help in writing better client side code.

JavaScript is main programming language to write client side code. We are seeing the explosion of JavaScript libraries. The reason for some JavaScript libraries got very popular and developers took them hand to hand because of its cool features with almost negligible performance cost. Now in our projects, count of JavaScript files (plugins and custom files) are increasing rapidly which is making it unmanageable and unmaintainable. AngularJS provides best of the both the worlds and this made it one of the most talked and used JavaScript frameworks in web applications.

In this book, we will be discussing the core concepts of AngularJS in detail and also see that. Each day we will start with a concept and discuss in details with an example. Some major concepts may be covered in multiple days as well based on the need.

***Note – Download the complete code for all the days from GitHub.***

# Day 1

# AngularJS Getting Started

Today is first day of the learning on AngularJS. We will focus on basics of AngularJS, initial constructs and create our first HelloWorld application and discuss it in details. Later, we will see that how any request processing takes place in it.

## What is AngularJS?

AngularJS is not just another JavaScript library but it is a complete framework that helps in writing a proper architectured, maintainable and testable client side code. Some of the key points are

- It follows MVC framework. If you don't have Idea MVC framework, I'll suggest you to have a look on MVC framework and then start learning AngularJS. You can refer this wiki page for MVC framework.
- AngularJS is primarily aimed to develop SPAs (Single Page Applications), it means your single HTML document turns into application. But it is also used a lot in ASP.NET and other web application technologies.
- It allows you to write Unit and integration tests for JavaScript code. Testability is one of the main points, which was kept in mind while writing this framework so it has great support of it.
- It provides its own and very rich list of attributes and properties for HTML controls which increases the usability of it exponentially. It is also called directives.
- Supports two-way binding that means your data model and control's data will be in sync.
- Angular supports Dependency injection. Read more about dependency injection on wiki.
- Angular library is also available on CDN so you just need to the URL of the CDN and it available for use.

## AngularJS is Open Source

AngularJS is an open source library which developed and supported by Google. Being an open source, you can go through the code itself and customize it if required. There is lot of support from JavaScript community and even you can contribute to it. Currently, more than 100 contributors have contributed and it is increasing day by day.

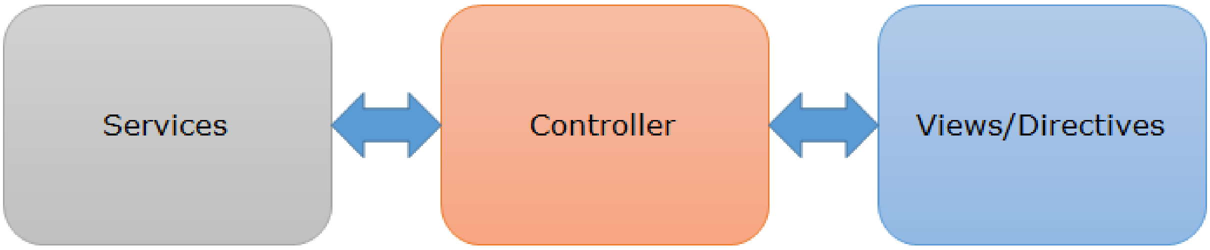
## Major Components

Controller – It is main component of AngularJS which contains the state and logic both. It acts as bridge between services and views.

Views/Directives – Here we generate the UI. Directives extends the HTML element and enables us to generate the complex html easily. Controllers talks to view in both directions.

Services – It contains the core logic and state of the application. In it, we can communicate to server to get and post the data.

Now let's see how all glued each other



Now you have got the basic explanation about the basics of AngularJS. So let's jump to the coding part. We need to learn few things mentioned below before writing the first application

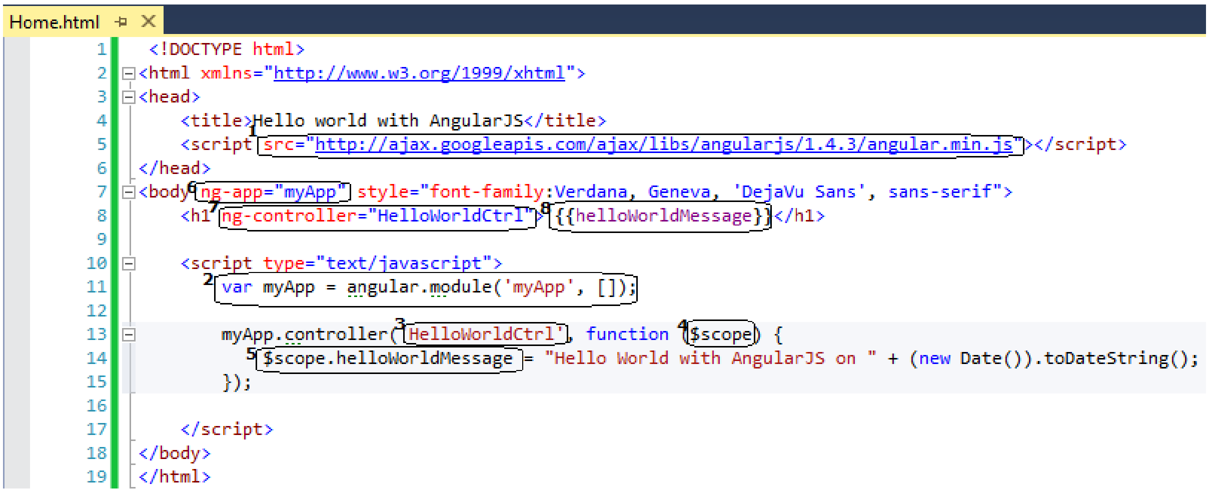
1- {{expression}} – It is called expressions and JavaScript like code can be written inside. It should be used for mainly small operations in HTML page.

2- $scope – This is one of very important variable used in AngularJS. It provides and link between the data and views. It holds all the required data for the views and used with in the HTML template.

3- ng-* – All the elements that are provided by angular starts from *ng-**. So if you see some attribute that ng- and angular library is also included in the page, then you can assume that this should be angular element only.

4- ng-app – This directive is used to bootstrap the application. Normally, it is put at in top level element like html or body tag. (That will be discussed in details in coming posts).

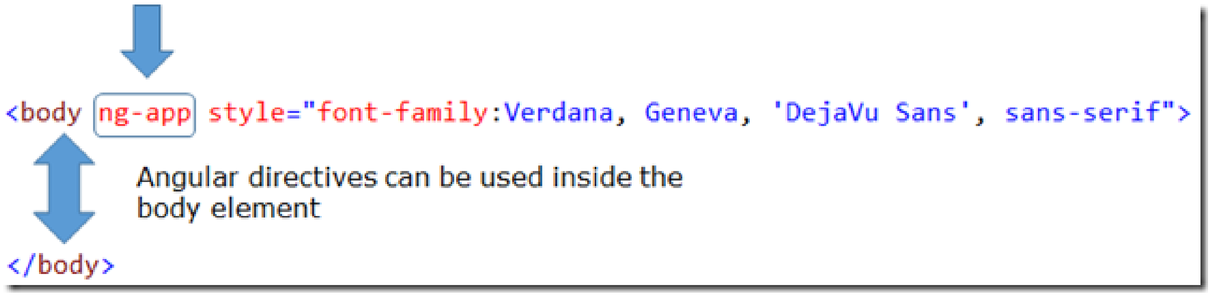
Now let's write our first *Hello World application* with AngularJS. In this example, I'll be using the basic construct of AngularJS. I have created Empty Project in Visual Studio and added an HTML page Home.html and written it as



```
  <!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Hello world with AngularJS</title>
    <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.4.3/angular.min.js"></script>
</head>
<body ng-app="myApp" style="font-family:Verdana, Geneva, 'DejaVu Sans', sans-serif">
    <h1 ng-controller="HelloWorldCtrl">{{helloWorldMessage}}</h1>

    <script type="text/javascript">
        var myApp = angular.module('myApp', []);

        myApp.controller('HelloWorldCtrl', function ($scope) {
            $scope.helloWorldMessage = "Hello World with AngularJS on " + (new Date()).toDateString();
        });

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

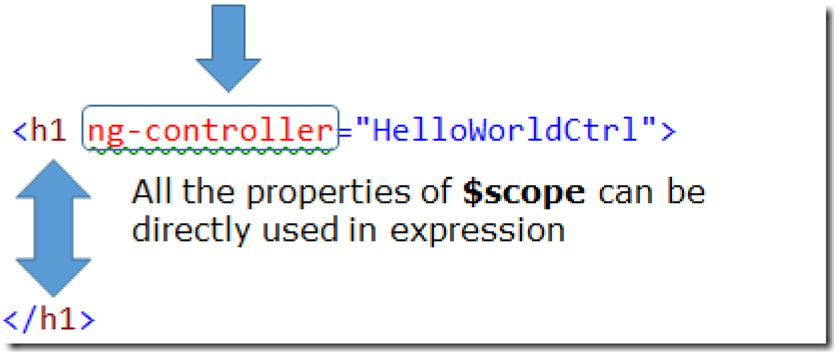
This is very simple page which uses AngularJS. I have encircled and numbered the specific part of the page that are relevant with AngularJS. Let's discuss it one by one.

1. I have included AngularJS library on the page using Google cdn.
2. Here we have created a module myApp that is assigned to ng-app attribute which works as container for an Angular application. We will discuss module in detail in coming posts.
3. Whenever we want to use the AngularJS, we need a controller. We defined our controller named HelloWorldCtrl here. Controller contains all the logic of an angular application. I appended Ctrl in the name just as a naming convention. Controller is later used in HTML element. Controller can be created in different file which we will see in coming posts.
4. We already discussed about $scope. This is a parameter in controller method and we can create dynamic properties and assign values to it, which can be later can be accessed in UI element. Here helloWorldMessage is used which is a dynamic property, created in controller.
5. Here we created a new property helloWorldMessage to $scope variable and assigns the message.
6. As discussed earlier that this attribute is required to bootstrap the AngularJS application so I have put it in body element. Putting it at body element allows us to use AngularJS inside this tag. In the above application, I could have put it at h1 element as well because I am using AngularJS inside this tag only. The scope can be displayed pictorially as



```
<body ng-app style="font-family:Verdana, Geneva, 'DejaVu Sans', sans-serif">
</body>
```

7. Here we assigned controller HelloWorldCtrl to ng-controller attribute and all functions/properties can be used inside the element. It can be pictorially depicted as

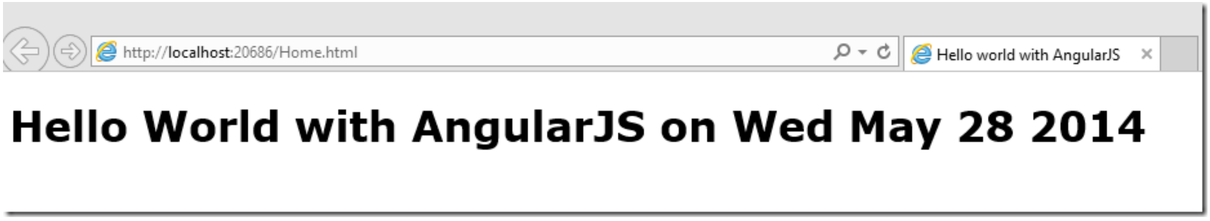


```
<h1 ng-controller="HelloWorldCtrl">
</h1>
```

All the properties of **$scope** can be directly used in expression

8. Now we see the expression here. As we provided the controller in this element and we can access the property `helloworldMessage` that we defined in the controller.

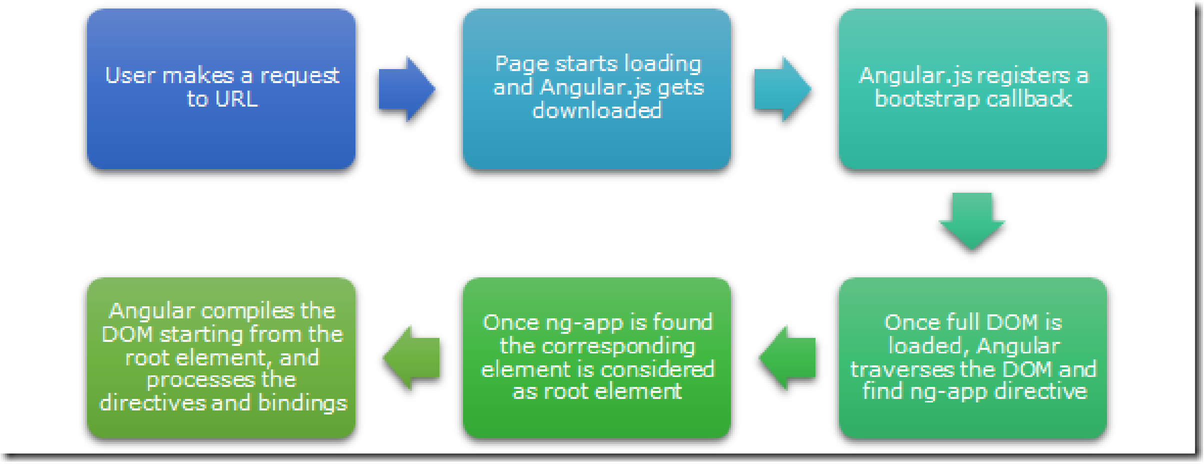
Now it's to run the application.



# Hello World with AngularJS on Wed May 28 2014

We can see the message with date and time that we have provided. Our page got rendered as per expectation

## Execution Flow of AngularJS Page Request



User makes a request to URL

Page starts loading and Angular.js gets downloaded

Angular.js registers a bootstrap callback

Once full DOM is loaded, Angular traverses the DOM and find ng-app directive

Once ng-app is found the corresponding element is considered as root element

Angular compiles the DOM starting from the root element, and processes the directives and bindings

So the above picture clearly illustrates that how does the flow of Angular page work at high level.

So In this post we created a very simple page which has all the code on the same HTML page. In the next post, we'll learn more concept and write a more detailed page and provide a better structure to the application.

## Conclusion

Today we discussed about AngularJS, its basic constructs and created our first Angular Application. Based on our example, I explained each angular elements and components then later discussed about the request processing of Angular Page.

# Day 2

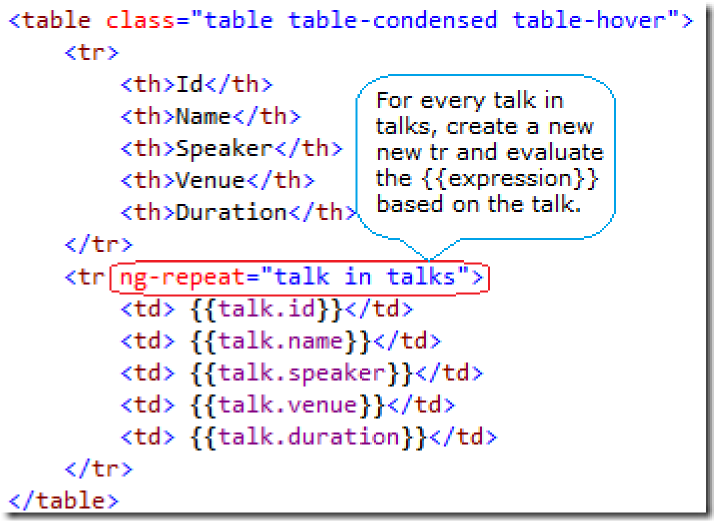
# Modules and Directives

In Day 1, we learnt about the basics of AngularJS that we used and created a simple hello world application. We also saw that how a request gets processed. As we discussed that AngularJS is a complete framework for writing client side code that allows us to use latest design principles, there are many important components.

Today, we'll take a step further and discuss some more concepts of AngularJS. In our example, we'll provide a structure to our application which is the beauty of AngularJS. In last post, we created a Hello World application, which used a controller and ng-app directive was used to bootstrap the application.

## Displaying Data in Grid format

One of the most common tasks in the web applications is, to show the data the tabular format and Angular did keep it in mind. If you've worked earlier on some client side template libraries then you will find it similar to that. Even if we don't have any idea about that then also you will find it very simple and easy to understand. Angular provides a way to repeat the some part of HTML (also called templates) based on the provided list of data. So we can have code like



```
<table class="table table-condensed table-hover">
    <tr>
        <th>Id</th>
        <th>Name</th>
        <th>Speaker</th>
        <th>Venue</th>
        <th>Duration</th>
    </tr>
    <tr ng-repeat="talk in talks">
        <td> {{talk.id}}</td>
        <td> {{talk.name}}</td>
        <td> {{talk.speaker}}</td>
        <td> {{talk.venue}}</td>
        <td> {{talk.duration}}</td>
    </tr>
</table>
```

For every talk in talks, create a new new tr and evaluate the {{expression}} based on the talk.

So if we see here the red encircled area then we see `ng-repeat` directive. It actually repeats the element on which it is used based on the provided list of data. Here *talks* is a JSON object which contains list of *talk*. It says for every talk in talks, repeat the `tr` and evaluate the provided expression based on the properties of talk. So if we have four items in the list then four rows will be created.

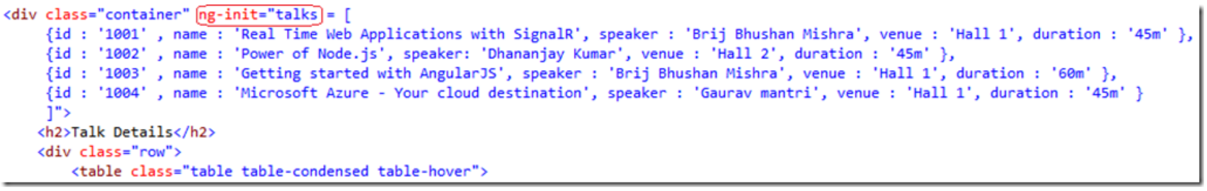
## How to Provided list of items?

In the above pic, `talks` is being accessed, it should be a global variable or a variable in the scope where it can be accessed. To fetch the data, there could be two ways

1. Initialize the value on the page
2. Or get the data from server/web services etc using some server side call preferably ajax call.

Let's take first scenario, initialize the value the on the page. AngularJS provides us a simple way to initialize using a directive that is called `ng-init`.

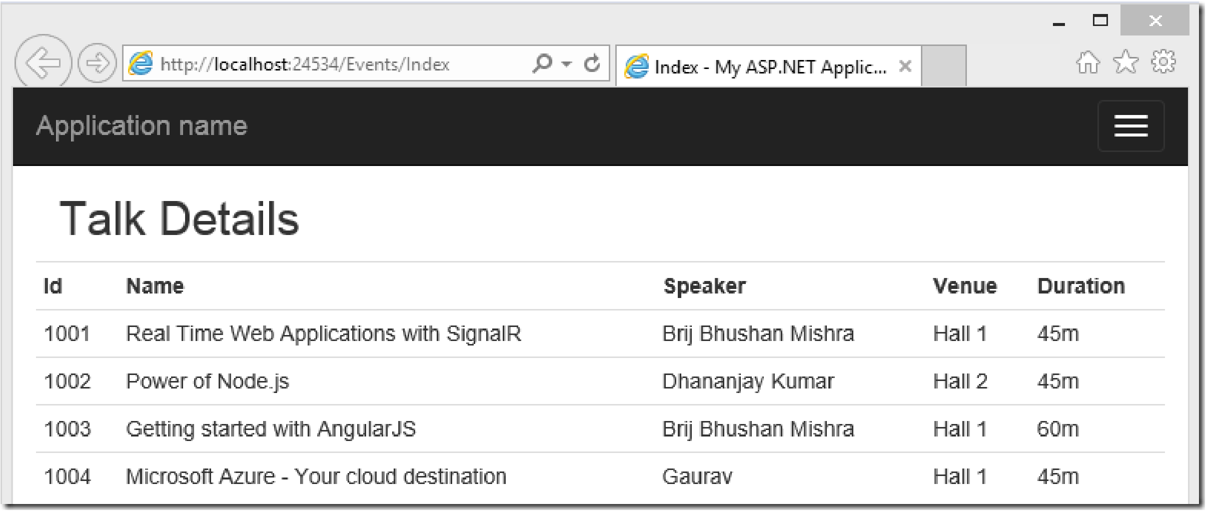
ng-init - This directive provides us a way to initialize a variable on the page. We can initialize the talks as



```
<div class="container" ng-init="talks = [
     {id : '1001' , name : 'Real Time Web Applications with SignalR', speaker : 'Brij Bhushan Mishra', venue : 'Hall 1', duration : '45m' },
     {id : '1002' , name : 'Power of Node.js', speaker: 'Dhananjay Kumar', venue : 'Hall 2', duration : '45m' },
     {id : '1003' , name : 'Getting started with AngularJS', speaker : 'Brij Bhushan Mishra', venue : 'Hall 1', duration : '60m' },
     {id : '1004' , name : 'Microsoft Azure - Your cloud destination', speaker : 'Gaurav mantri', venue : 'Hall 1', duration : '45m' }
     ]">
    <h2>Talk Details</h2>
    <div class="row">
        <table class="table table-condensed table-hover">
```

So here, put a div over the table and in div element, we initialized the variable talks using `ng-init` directive as above.

I added the angular library on the page. Also I have used `bootstrap.css` for rendering the not so bad looking UI. Let's run the application.


http://localhost:24534/Events/Index
Index - My ASP.NET Applic...
Application name

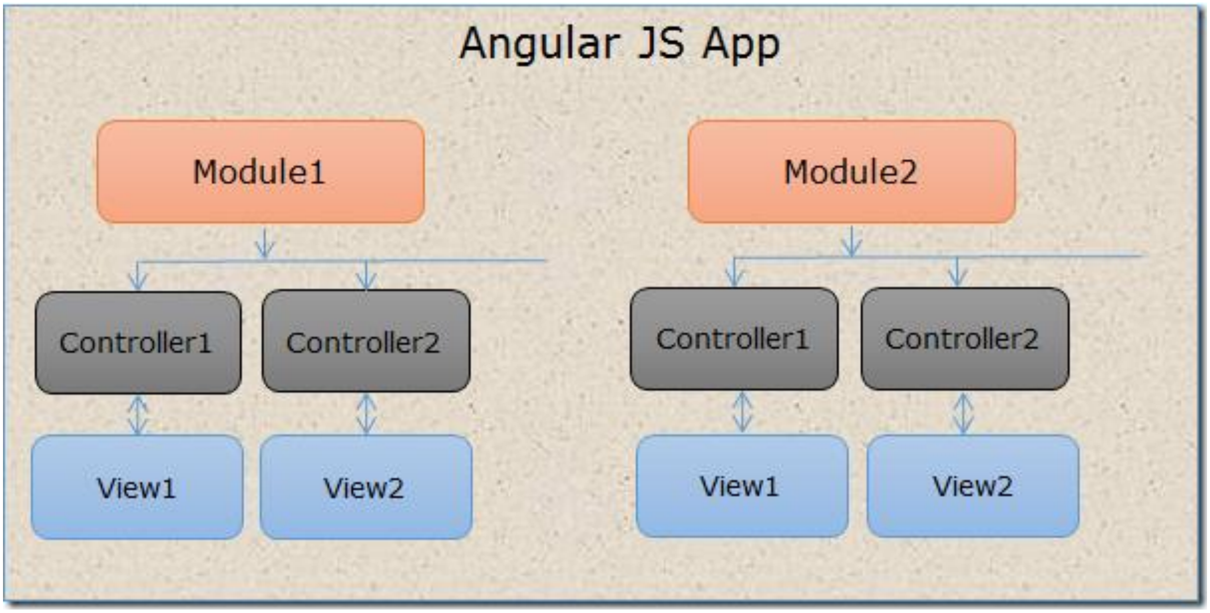
Talk Details

| Id | Name | Speaker | Venue | Duration |
|---|---|---|---|---|
| 1001 | Real Time Web Applications with SignalR | Brij Bhushan Mishra | Hall 1 | 45m |
| 1002 | Power of Node.js | Dhananjay Kumar | Hall 2 | 45m |
| 1003 | Getting started with AngularJS | Brij Bhushan Mishra | Hall 1 | 60m |
| 1004 | Microsoft Azure - Your cloud destination | Gaurav | Hall 1 | 45m |

So we have created an angular app. Now before providing it's a structure, let's discuss one very important component of Angular that we'll use in today that is called Module.

## What is Module?

Modules are like containers. It is a top level element and contains all the part of AngularJS that are used on the page. So let's see how it fits into an application



Above image is for the basic understanding. It shows an AngularJS application can have multiple modules and every module contains controllers, views etc.

There are many other components which we'll introduce in coming posts. As we discussed last day that the main beauty of an AngularJS application is its architecture so let's restructure the code that we have written and we'll use another component module in this example.

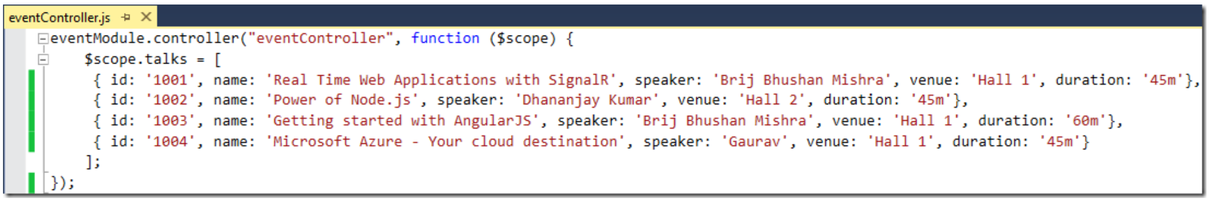
We are going to have controller and module in our application. These should be put in different JavaScript file. If you are working on some enterprise level application which has lots of pages then one need to decide how is (s)he going manage or structure that. There are different views on that I'll not discuss that in detail. In my code, I normally prefer having folders functionality wise, which makes very easy to find any file, add/remove any functionality at any point of time.

In this example, I have created a folder named as events. Now I have put all the angular JavaScript files associated to this feature in the same folder. As we discussed that the top level item is module. So let's create a module named as `eventModule`.

So I have created a file `eventModule.js` and here I registered this module with angular.

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

Now let's create a controller named *eventController* as



```
eventModule.controller("eventController", function ($scope) {
    $scope.talks = [
     { id: '1001', name: 'Real Time Web Applications with SignalR', speaker: 'Brij Bhushan Mishra', venue: 'Hall 1', duration: '45m'},
     { id: '1002', name: 'Power of Node.js', speaker: 'Dhananjay Kumar', venue: 'Hall 2', duration: '45m'},
     { id: '1003', name: 'Getting started with AngularJS', speaker: 'Brij Bhushan Mishra', venue: 'Hall 1', duration: '60m'},
     { id: '1004', name: 'Microsoft Azure - Your cloud destination', speaker: 'Gaurav', venue: 'Hall 1', duration: '45m'}
    ];
});
```

So here controller also got registered with Module. In this method we initialized the talks in *$scope* variable which is a default parameter to the controller.

Now let's move our HTML page and provide the controller name to the element.

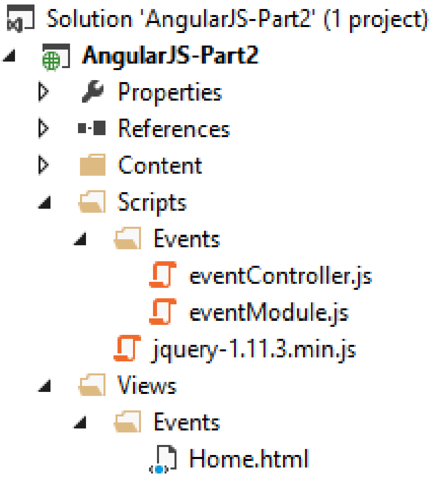
```
<div class="row" ng-controller="eventController">
```

Now we have created a module and a controller. Controller is also registered with module and controller is assigned to the UI element as well. Also we mapped module with angular. What else is left?

How the module that we created will get loaded. In last post, we discussed that when AngularJS loads it looks for the ng-app directive and this is the place where we need to provide our module as

```
<html ng-app="eventModule">
```

We need to include all the controller and module JavaScript files to our page that we created. Now, we don't have any JavaScript code on our page. The folder structure of our applications look like



```
Solution 'AngularJS-Part2' (1 project)
  AngularJS-Part2
    Properties
    References
    Content
    Scripts
      Events
        eventController.js
        eventModule.js
      jquery-1.11.3.min.js
    Views
      Events
        Home.html
```

If I want to add some more features, I can easily create another folder with feature name under script folder and similarly under Views as well. And at later point of time, if I want to remove, I can easily do that as well.

Now when we run the page, we get the similar output which we got initially. Refer third figure from Top.

## Conclusion

Today, we discussed about some new directives `ng-init, ng-repeat` and displayed the data in grid format at our page. Then we discussed about new component module and gave our solution proper structure by placing the JavaScript code in right file and right folder.

# Day 3

# AngularJS with ASP.NET MVC

Day 1, we created a simple Hello World application using AngularJS and learn the basics of AngularJS. On Day 2, we created a page which displays data in tabular format and used few directives to generate that. We also discussed one of the very important components, Module and provided a proper structure to our application.

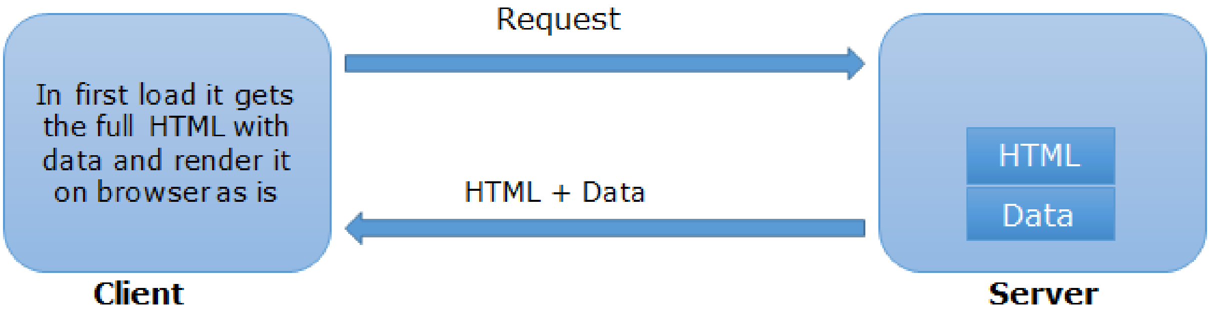
## How to use Angular in ASP.NET MVC

Today we are going to see a marriage between AngularJS and ASP.NET MVC and leverage the power of AngularJS with one of the used Web Application frameworks - ASP.NET MVC. Last day, we created a HTML page and initialized the data using `ng-init` directive. Now let's think that how can we have the similar application in ASP.NET MVC. In that example, we provided the data at html page itself. But while working with ASP.NET application, we may want to provide the initial data from server while initial load.

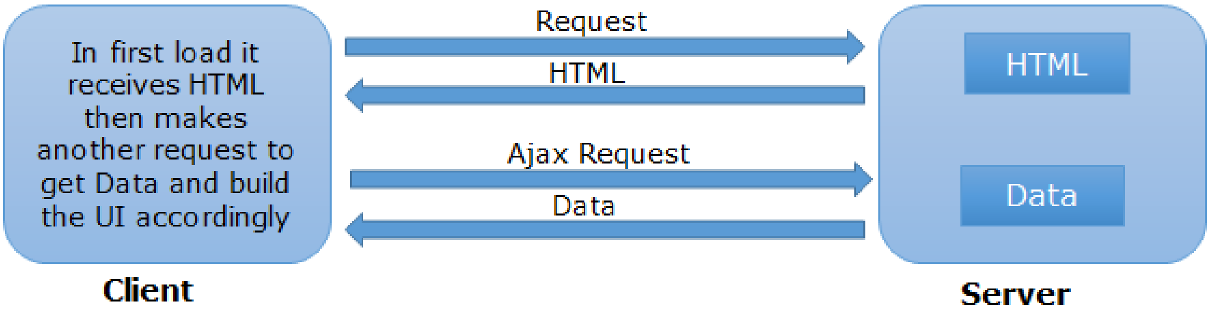
## Approach to get data from Server

To achieve this, there are two ways.

1. Get the HTML and data from the server while the page loads itself


Request
In first load it gets the full HTML with data and render it on browser as is
HTML + Data
HTML
Data
Client
Server

2. Load the page first, then get the data via an AJAX call and update the page accordingly.


In first load it receives HTML then makes another request to get Data and build the UI accordingly
Request
HTML
Ajax Request
Data
HTML
Data
Client
Server

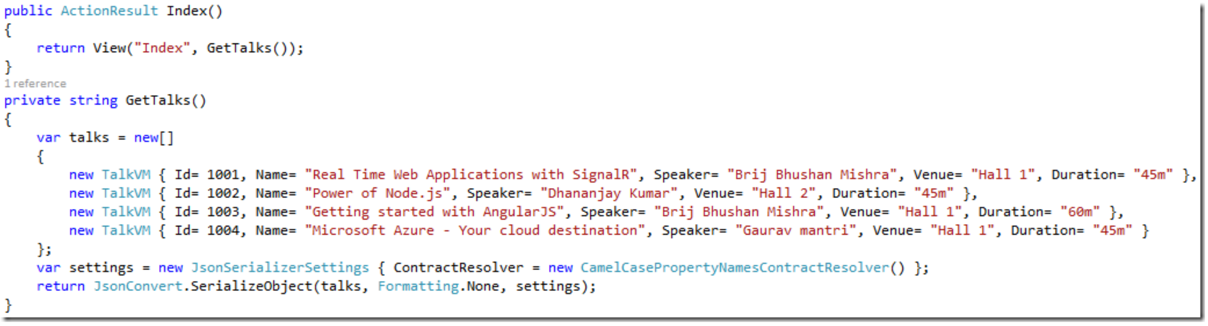
## Using first approach

We'll take the first approach, because in a typical ASP.NET MVC application, we provide data to the view via controller and use that model in the cshtml.

### *Creating ASP.NET MVC Application*

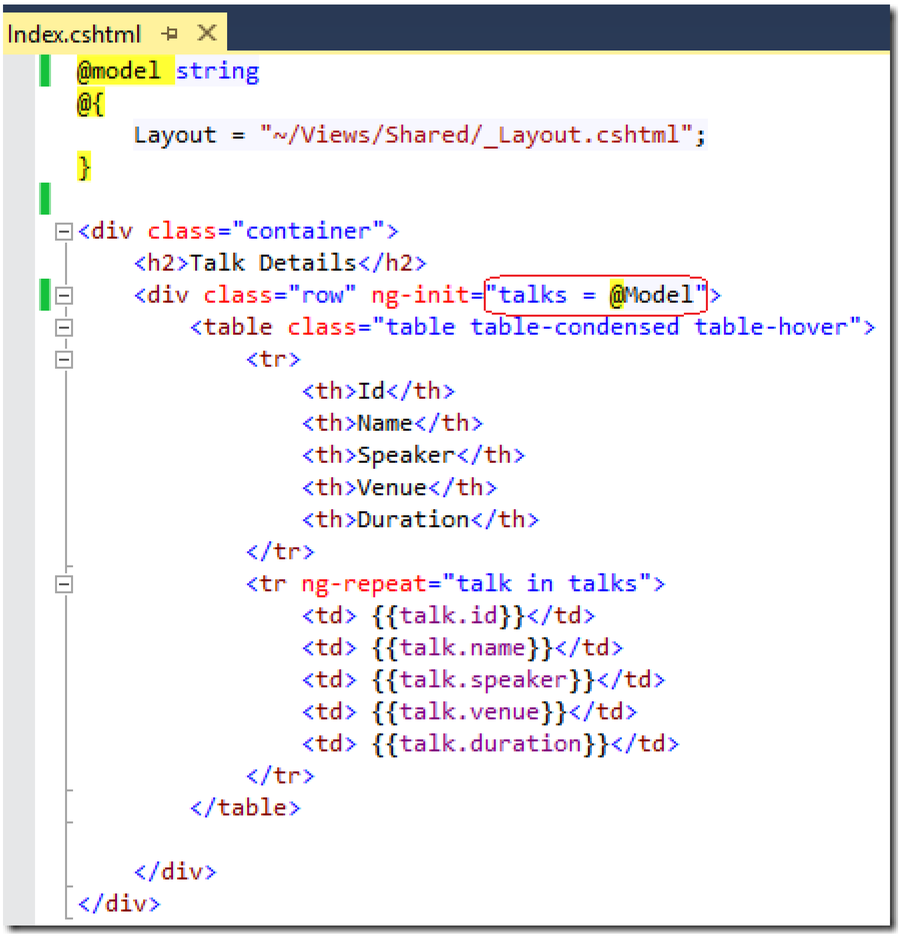
For that I created an ASP.NET MVC empty application and created a MVC Controller named `EventsController.cs` and put only index method. Then I have created an empty view for that.

In the view, copied the html that we created last day where we assigned data in `ng-init`. As, we didn't want to hard code the data in our view itself and want this to be fetched from server. Here we require the data in JSON format at client side so returned the data after serializing in JSON format. The controller looks as



```
public ActionResult Index()
{
    return View("Index", GetTalks());
}
1 reference
private string GetTalks()
{
    var talks = new[]
    {
        new TalkVM { Id= 1001, Name= "Real Time Web Applications with SignalR", Speaker= "Brij Bhushan Mishra", Venue= "Hall 1", Duration= "45m" },
        new TalkVM { Id= 1002, Name= "Power of Node.js", Speaker= "Dhananjay Kumar", Venue= "Hall 2", Duration= "45m" },
        new TalkVM { Id= 1003, Name= "Getting started with AngularJS", Speaker= "Brij Bhushan Mishra", Venue= "Hall 1", Duration= "60m" },
        new TalkVM { Id= 1004, Name= "Microsoft Azure - Your cloud destination", Speaker= "Gaurav mantri", Venue= "Hall 1", Duration= "45m" }
    };
    var settings = new JsonSerializerSettings { ContractResolver = new CamelCasePropertyNamesContractResolver() };
    return JsonConvert.SerializeObject(talks, Formatting.None, settings);
}
```

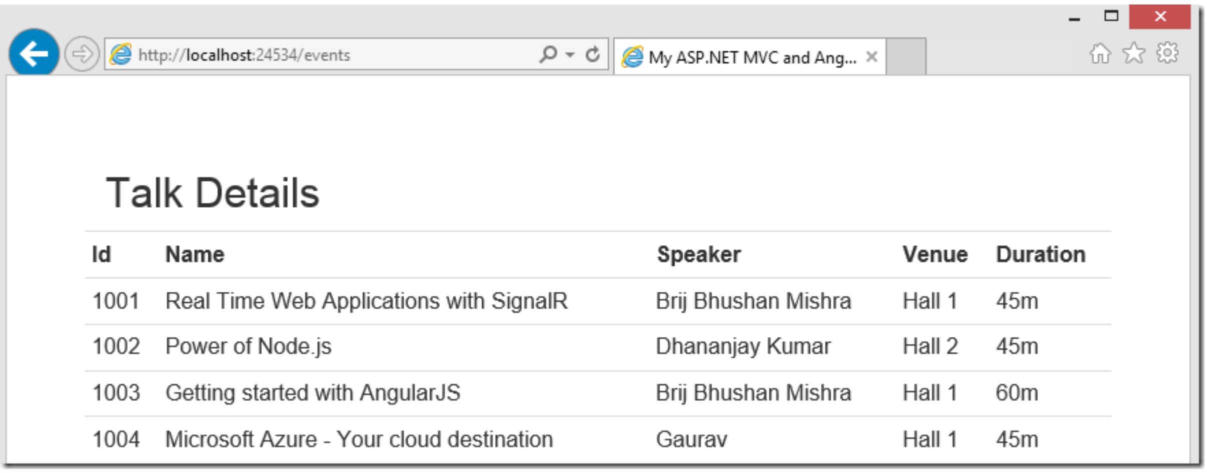
Here, I have created a list of talks and converted it in JSON format using the .NET library and returned. For this sample, I have hard-coded the data but in real implementation, it can be fetched from database or some web services or anywhere else. Anyways now we can use this data in our view and provide this to `ng-init` directive. Now my `Index.cshtml` looks like



```
@model string
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<div class="container">
    <h2>Talk Details</h2>
    <div class="row" ng-init="talks = @Model">
        <table class="table table-condensed table-hover">
            <tr>
                <th>Id</th>
                <th>Name</th>
                <th>Speaker</th>
                <th>Venue</th>
                <th>Duration</th>
            </tr>
            <tr ng-repeat="talk in talks">
                <td> {{talk.id}}</td>
                <td> {{talk.name}}</td>
                <td> {{talk.speaker}}</td>
                <td> {{talk.venue}}</td>
                <td> {{talk.duration}}</td>
            </tr>
        </table>

    </div>
</div>
```

Here talks is initialized with the model (refer the Red encircled area) and provided in *ng-init* directive.

I also provided the *ng-app* directive in `_Layout.cshtml` in body tag. Now let's run it.



Talk Details

| Id | Name | Speaker | Venue | Duration |
|---|---|---|---|---|
| 1001 | Real Time Web Applications with SignalR | Brij Bhushan Mishra | Hall 1 | 45m |
| 1002 | Power of Node.js | Dhananjay Kumar | Hall 2 | 45m |
| 1003 | Getting started with AngularJS | Brij Bhushan Mishra | Hall 1 | 60m |
| 1004 | Microsoft Azure - Your cloud destination | Gaurav | Hall 1 | 45m |

It displays the data as expected. This is a very simple ASP.NET MVC application with AngularJS.

Now let's provide the proper structure. For that, I have copied the module and controller javascript files. I also put `ng-app="eventModule" in _Layout.cshtml` as

```
<html ng-app="eventModule">
```

In last post second example, we hard-coded the data in the angular controller. Currently we have data at View so the next question is, how to pass it to the there. So there could be multiple ways.

1 Create a JavaScript global variable in view and use that variable in controller. But that is not a good design.
2 Another way, we can add a service that returns the data.

As we have data at our view (`index.cshtml`) so we need to read it from here and send it to controller in one or other way. For that purpose, we can use services and put it on the view only. But before using services, let's discuss what AngularJS Service is?

## Angular Services

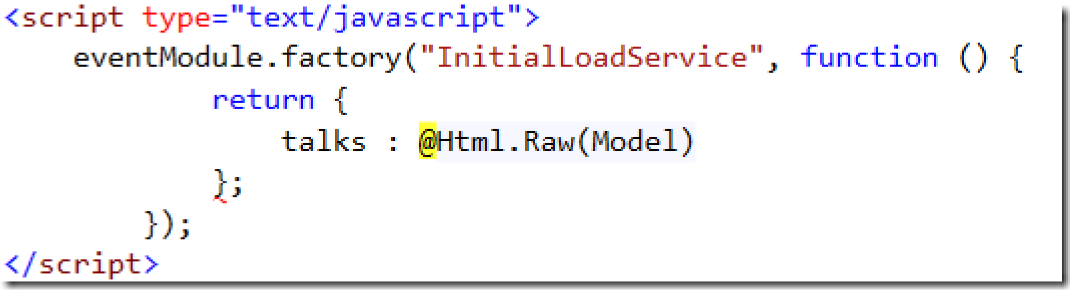
Angular Services are not like any other services which travels over the wire. But it is simple object which does some work for you. In specific words, Angular Services are like a singleton object which does some task for you and can be reused. The main benefits of the services is single responsibility, re-usability and. testability.

### Using Angular Services

AngularJS has many in built services but it also provides us capability to write our own custom services. Once we create a service we need to inject in some controller.

```
eventModule.factory(“<ServiceName>”, function() {

// Do some task and return data

});
```

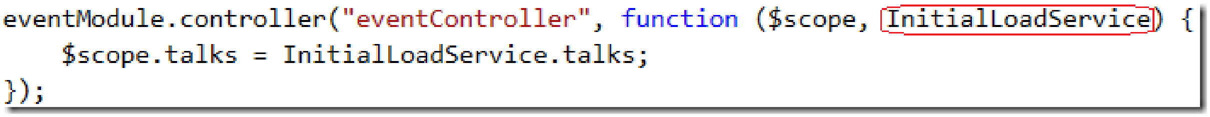
In our example, we have created a service named `InitialLoadService` as



```
<script type="text/javascript">
    eventModule.factory("InitialLoadService", function () {
            return {
                talks : @Html.Raw(Model)
            };
        });
</script>
```

In the above code, we have created a service name `InitialLoadService` that has been registered with module using factory method. This just simply returns the data that is read from the model.

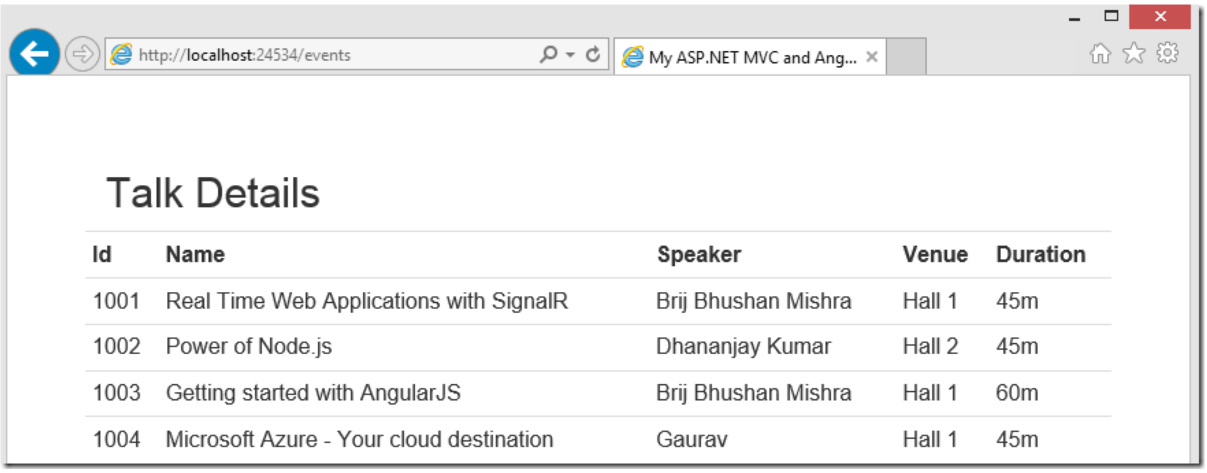
Now we need to use this service. For that we can inject it in our controller (`eventController.js`) as



```
eventModule.controller("eventController", function ($scope, InitialLoadService) {
    $scope.talks = InitialLoadService.talks;
});
```

Here in our controller, we have injected the services (refer Red encircled Area) and read the data from the service.

Now we have created a custom service and injected the service in controller. Let's run the application



Talk Details

| Id | Name | Speaker | Venue | Duration |
|---|---|---|---|---|
| 1001 | Real Time Web Applications with SignalR | Brij Bhushan Mishra | Hall 1 | 45m |
| 1002 | Power of Node.js | Dhananjay Kumar | Hall 2 | 45m |
| 1003 | Getting started with AngularJS | Brij Bhushan Mishra | Hall 1 | 60m |
| 1004 | Microsoft Azure - Your cloud destination | Gaurav | Hall 1 | 45m |

Great!! Now we have successfully created an ASP.NET MVC and AngularJS application. It is same as earlier one in this post but here we used different components of AngularJS.

## Conclusion

Today we saw that how we can leverage the power of AngularJS in an ASP.NET MVC application. We used two approaches to load the initial data on client side. Later we discussed another component Services in Angular and used in second approach.

# Day 4

# Services in AngularJS

It is Day 4 and till date we got basic understanding of AngularJS and also gave an initial look on the use of AngularJS with ASP.NET MVC. Till date we discussed four major components `Controllers`, `Directives`, `Modules` and `Factories`. Today we are going to discuss `Services`.

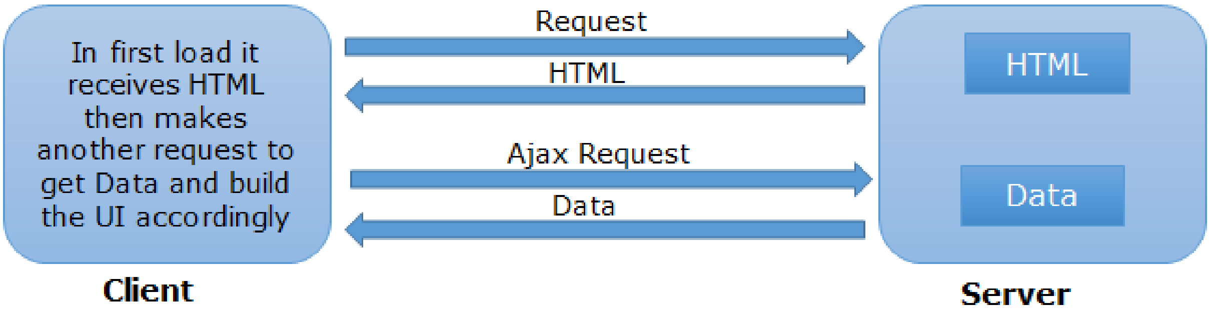
Angular services is vast topic but I am going to cover the important concepts. One of the important technologies that made the web glittering is AJAX. So today, our focus would that how AngularJS enables us to use AJAX.

On Day 3, we first created a simple ASP.NET MVC project and displayed data in tabular format. We provided our Angular application, a proper structure and used controller, module and Factory services. In that example, we fetched the data while page load. We also discussed that there are two ways to load the data on UI and these are

1. Get the HTML and data from the server while the page loads itself.
2. Load the page first, then get the data via an AJAX call and update the page accordingly.

## How to Initiate AJAX request

Today we will be using second approach and see that how we can initiate the AJAX request via AngularJS.


In first load it receives HTML then makes another request to get Data and build the UI accordingly
Request
HTML
Ajax Request
Data
HTML
Data
Client
Server

For our example, we'll use the sample that we created previous day. In last sample, the data is passed with the view itself which won't be the case here. So let's make few changes in `EventController.cs`.

1- Don't send the data with view. So Index action would look like as

```
public ActionResult Index()
{
        return View();
}
```

2- Let's create a repository class (say `EventRepository`) that returns the data as required. It'll help us further while scaling up this application. This repository has `GetTalks` that returns array of talks as

```
public TalkVM[] GetTalks()
{
        var talks = new[]
        {
                new TalkVM { Id= "T001", Name= "Real Time Web 
Applications with SignalR", Speaker= "Brij Bhushan Mishra", Venue= 
"Hall 1", Duration= "45m" },
                new TalkVM { Id= "T002", Name= "Power of Node.js", 
Speaker= "Dhananjay Kumar", Venue= "Hall 2", Duration= "45m" },
                new TalkVM { Id= "T003", Name= "Getting started with 
AngularJS", Speaker= "Brij Bhushan Mishra", Venue= "Hall 1", 
Duration= "60m" },
                new TalkVM { Id= "T004", Name= "Microsoft Azure - 
Your cloud destination", Speaker= "Gaurav", Venue= "Hall 1", 
Duration= "45m" }
         };
         return talks;
}
```

3- Let's change the `GetTalks` method of controller to `GetTalkDetails` and mark it as public because we'll call it using AJAX. Also change the return type as `ActionResult`. For that, we created an instance of type `ContentResult` and return that. It internally calls `EventRepository` to get the talks and convert it to the type `ContentResult`. It looks like

```
public ActionResult GetTalkDetails()
{
        var settings = new JsonSerializerSettings { ContractResolver = 
new CamelCasePropertyNamesContractResolver() };

        var jsonResult = new ContentResult
        {
                Content = 
JsonConvert.SerializeObject(eventsRepository.GetTalks(), settings),
                ContentType = "application/json"
        };

        return jsonResult;
}
```

Now we have made changes at server side. We have a method `GetTalkDetails` that is ready to be called via AJAX.
So let's move to `Index.cshtml`. As here we created an Angular Service that just read the data from the server using AJAX. Now we don't want it in View that so let's delete it. So let's create a new file named EventsService.js to maintain the proper structure.

Here we are going to use two inbuilt angular services. Let's discuss these.

## What is $http services

As the name suggest, it enable to initiate the AJAX request to the server and get the response. This is a core angular service and uses XMLHTTPRequest. This API can be paired with $q service to handle the request asynchronously. Its syntax is similar to jQuery AJAX.

## What is $q services

This service exposes deferred and promise APIs. Defer API is used to expose the Promise instance. Promise returns the result when the asynchronous request successfully completes. It has also APIs that tells us whether the request got successful or unsuccessful.

We'll use both in our example. Now it's time to write the code in Angular service file.

```
eventModule.factory("EventsService", function ($http, $q) {
    return {
        getTalks: function () {
            // Get the deferred object
            var deferred = $q.defer();
            // Initiates the AJAX call
            $http({ method: 'GET', url: '/events/GetTalkDetails'
}).success(deferred.resolve).error(deferred.reject);
            // Returns the promise - Contains result once request
completes
            return deferred.promise;
        }
    }
});
```

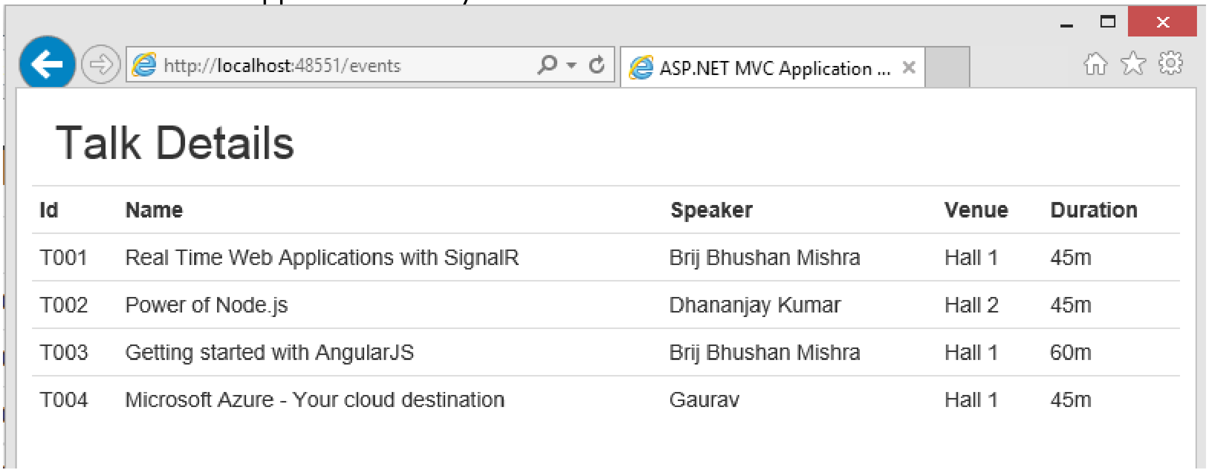
In the above code, we are using $http and $q services both. In getTalks, we have created a deferred object using $q. Then used $http services to initiate the asynchronous request to the server to get the data. Based on the request status success and error callbacks are called. In success callback, we provided deferred.resolve and in case of error, deferred.reject. deferred.resolve resolves the derived promise with its value.

We also need to make changes in our controller. But do we need to change the controller? If Yes Why?

Because the service method returns the Promise, not the actual value. While accessing the promise, response may not be received. So we cannot directly use it. To handle that, we need to use *then*, which gets fired once promise is resolved or rejected. It takes two functions as parameter, first gets called in case of success and second is called in case of error. Based on status of promise, it calls success or error one asynchronously. Let's see the code

```
eventModule.controller("eventController", function ($scope, EventsService)
{
    EventsService.getTalks().then(function (talks) { $scope.talks = talks
}, function ()
    { alert('error while fetching talks from server') })
});
```

Now we have our application ready. So let's run it.



Talk Details

| Id | Name | Speaker | Venue | Duration |
|---|---|---|---|---|
| T001 | Real Time Web Applications with SignalR | Brij Bhushan Mishra | Hall 1 | 45m |
| T002 | Power of Node.js | Dhananjay Kumar | Hall 2 | 45m |
| T003 | Getting started with AngularJS | Brij Bhushan Mishra | Hall 1 | 60m |
| T004 | Microsoft Azure - Your cloud destination | Gaurav | Hall 1 | 45m |

Great!! This is what we have expected.

*Note – `$http` is a very powerful service and provide lots of features. I have just discussed the basic concept and feature around this.*

## Conclusion

Today I introduced two very important Angular Services $http and $q and saw that how we can initiate AJAX call using these serveries. And leveraged in our ASP.NET MVC project. There are many more inbuilt services in AngularJS and some of them we will use in coming posts.

# Day 5

# Views and Routing

This is Day 5 and we had discussed many basic components and learnt those by examples. We also saw how can we leverage it in ASP.NET 5 and saw the way to structure an Angular Application. Today we are going to discuss two very important components Views and Routing. Although you might have seen about Views earlier but there are more it and we will explore it today

Earlier whenever we required to display multiple views on a single webpage then we normally put multiple `divs` on the page and show or hide div using JavaScript or jQuery. One of the main issues with this approach is that the URL does not change as view changes so bookmarking any particular view is not possible. It also becomes tough to maintain these pages as it contains lots of html and JavaScript code as well.

AngularJS provides a robust solution to handle these scenarios. First it enables us to help in maintaining multiple views of the page in separate files that is used to load the particular view and also allows us to have different URL based on each view. Let's understand Views and Routing.

## What is View?

Angular allows us to have multiple templates for views that can be used to provide multiple different view. To implement this, it provides an attribute `ng-view` which can be changed based on the defined routes. For each views it allows us to have different controller to provide all the required logic. We will it in coming examples. Let's discuss routes now.

## What is Routing?

Routing plays a key role in scenarios where we have multiple views on the same page. It enables us to provide different templates (View) and controller based on URL. On clicking particular URL, the corresponding template is loaded with the assigned controller. To define the routes Angular provides another service named `$routeprovider`.

There is major breaking changes took place in `1.2.*`. In this version, routing (`$routeprovider`) is not part of main angular library and to use it we need to include another plugin (`angular-route.js`) and while creating the module we need to pass `ngModule`. We'll see in coming example.

*Note – These Views are similar to ASP.NET MVC views. For each view there is a Controller associated, which uses the View and add required data to render it on the UI.*

## Example using Views and Routing

So let's discuss the example. We'll take the last day's sample and scale it up. In that example, we displayed Talk details on the page. Now we will display speaker details on that page as well. So we'll have two tabs: Talk Details and Speaker Details. And by clicking on the tabs, corresponding details will be displayed.

### Server side code changes

1- Added a new method `GetSpeakers` in `EventRepository` that returns an array of speakers. The code is as

```
public SpeakerVM[] GetSpeakers()
{
    var speakers = new[]
    {
        new SpeakerVM { Id= "S001", Name= "Brij Bhushan Mishra",
Expertise= "Client Script, ASP.NET", TalksDelivered= 28 },
        new SpeakerVM { Id= "S002", Name= "Dhananjay Kumar", Expertise=
"Node.js, WCF", TalksDelivered= 54 },
        new SpeakerVM { Id= "S003", Name= "Gaurav", Expertise=
"Microsoft Azure", TalksDelivered= 68 }
    };
    return speakers;
}
```

2- Added a new method in `GetSpeakerDetails` (similar to `GetTalkDetails`) in `EventsController` that gets the data from Repository and returns the `JSONResult`. This method will be called via Angular Service. The method is as

```
public ActionResult GetSpeakerDetails()
{
    var settings = new JsonSerializerSettings { ContractResolver = new
CamelCasePropertyNamesContractResolver() };

    var jsonResult = new ContentResult
    {
        Content =
JsonConvert.SerializeObject(eventsRepository.GetSpeakers(), settings),
        ContentType = "application/json"
    };

    return jsonResult;
}
```

### Updating Client side Code:

Now we have added server side code. Let's change Angular code at Client side

1- Added a new service `getSpeakers` in Event Service as

```
getSpeakers: function () {
    // Get the deferred object
    var deferred = $q.defer();
    // Initiates the AJAX call
    $http({ method: 'GET', url: '/events/GetSpeakerDetails'
}).success(deferred.resolve).error(deferred.reject);
```

```
    // Returns the promise - Contains result once request completes
    return deferred.promise;
}
```

It is similar to `getTalks` that we discussed in Day 4.

2- As every view has one controller so let’s create another controller for speaker details named as `speakerController` (`speakerController.js`) and put the code similar to `eventController` as

```
eventModule.controller("speakerController", function ($scope,
EventsService) {
    EventsService.getSpeakers().then(function (speakers) {
$scope.speakers = speakers }, function ()
    { alert('error while fetching speakers from server') })
});
```

3- Now it’s time to create to templates that will be rendered on the UI. For that, I have created `Templates` folder. In this folder we’ll create templates for talk details and speaker details. So let’s create an HTML file named as *Talk.html* and copy the html from `Index,html` and it looks like as

```
<div class="container">
    <h2>Talk Details</h2>
    <div class="row">
        <table class="table table-condensed table-hover">
            <tr>
                <th>Id</th>
                <th>Name</th>
                <th>Speaker</th>
                <th>Venue</th>
                <th>Duration</th>
            </tr>
            <tr ng-repeat="talk in talks">
                <td> {{talk.id}}</td>
                <td> {{talk.name}}</td>
                <td> {{talk.speaker}}</td>
                <td> {{talk.venue}}</td>
                <td> {{talk.duration}}</td>
            </tr>
        </table>
    </div>
</div>
```

Let’s create similar view for Speaker View as well. It is as

```
<div class="container">
    <h2>Speaker Details</h2>
    <div class="row">
        <table class="table table-condensed table-hover">
            <tr>
                <th>Id</th>
                <th>Name</th>
                <th>Expertise</th>
                <th>Talks Delivered</th>
            </tr>
            <tr ng-repeat="speaker in speakers">
                <td> {{speaker.id}}</td>
                <td> {{speaker.name}}</td>
                <td> {{speaker.expertise}}</td>
                <td> {{speaker.talksDelivered}}</td>
            </tr>
        </table>
    </div>
</div>
```

4- Let’s go to `index.cshtml`. Now in this file we’ll have only tabs and one more new item `ng-view` as

```
<div class="container">
    <div class="navbar navbar-default">
        <div class="navbar-header">
            <ul class="nav navbar-nav">
                <li class="navbar-brand"><a
href="#Talks">Talks</a></li>
                <li class="navbar-brand"><a
href="#Speakers">Speakers</a></li>
            </ul>

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

    </div>
</div>
```

Here I have used nav bar to create the navigation links. Please refer the `href` links, one is for Talks and another for Speakers. And in the links we provided the url with `#`. Angular `routeProvider` paths are added with # in the base url. As discussed, based on the route the required templates will be rendered inside ng-view.

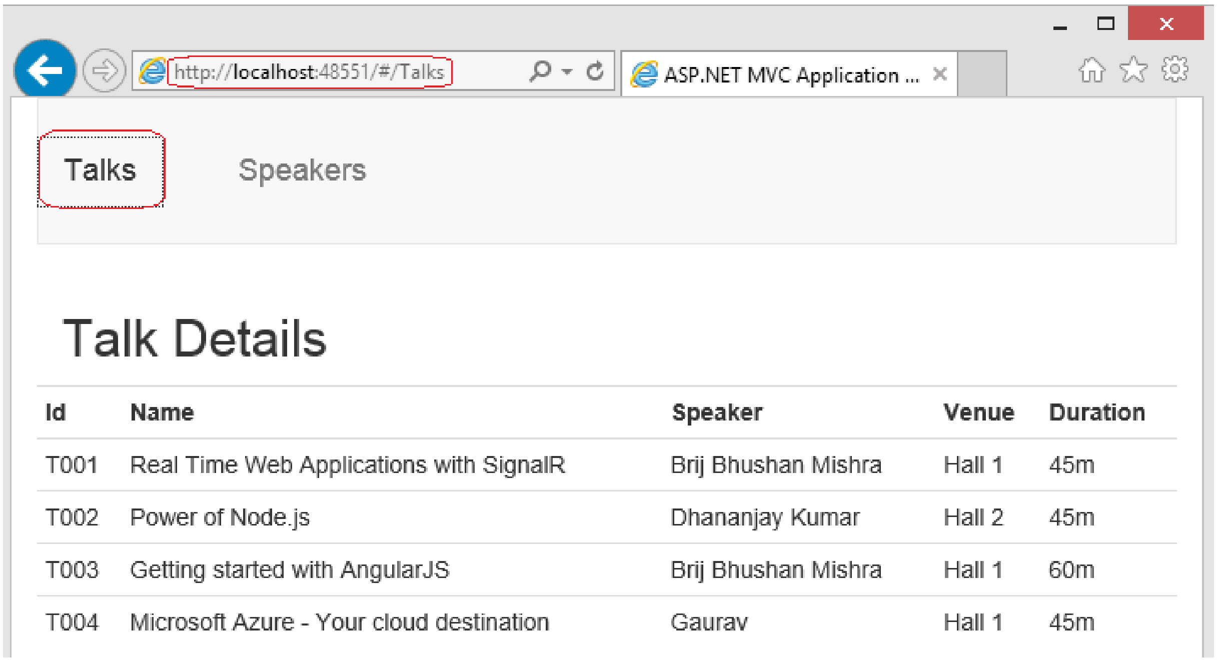
5- Now we have only one item that is left, to initialize the route that is defined in the module. As mentioned earlier, that to use routing we need to use another plugin (`angular-route.js`) and initialize in module

```
var eventModule = angular.module("eventModule",
['ngRoute']).config(function ($routeProvider) {
                         //Path - it should be same as href link
    $routeProvider.when('/Talks', { templateUrl:
'/Templates/Talk.html', controller: 'eventController' });
    $routeProvider.when('/Speakers', { templateUrl:
'/Templates/Speaker.html', controller: 'speakerController' });
});
```

As we earlier discussed that module is like container to all the features and services. Route is configured via `config` method in module. Path of the link should be same as the path used in the navigation links. For each path, we provides the path of the corresponding View and its associated controller.

## Running the example

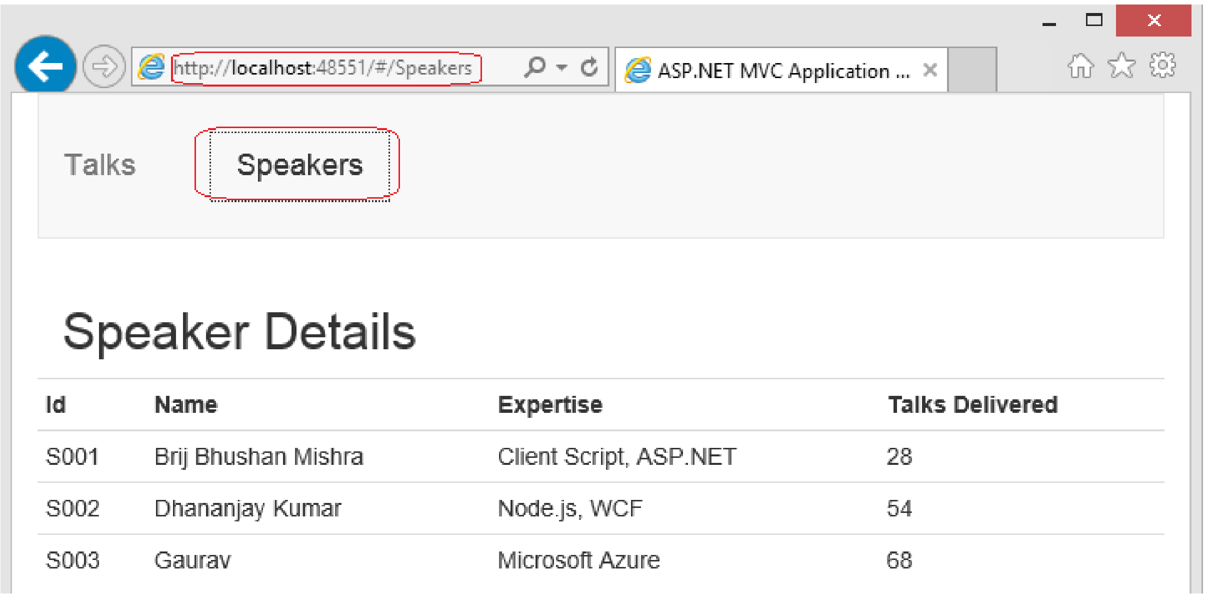
Now our application is ready and let’s run this



## Talk Details

| Id | Name | Speaker | Venue | Duration |
|---|---|---|---|---|
| T001 | Real Time Web Applications with SignalR | Brij Bhushan Mishra | Hall 1 | 45m |
| T002 | Power of Node.js | Dhananjay Kumar | Hall 2 | 45m |
| T003 | Getting started with AngularJS | Brij Bhushan Mishra | Hall 1 | 60m |
| T004 | Microsoft Azure - Your cloud destination | Gaurav | Hall 1 | 45m |

Here I ran the application and clicked on the Talks link (red encircled). After the url got changed (refer url) and talks got populated. Similarly let's click on Speakers


http://localhost:48551/#/Speakers
ASP.NET MVC Application ...
Talks
Speakers

## Speaker Details

| Id | Name | Expertise | Talks Delivered |
|---|---|---|---|
| S001 | Brij Bhushan Mishra | Client Script, ASP.NET | 28 |
| S002 | Dhananjay Kumar | Node.js, WCF | 54 |
| S003 | Gaurav | Microsoft Azure | 68 |

When I clicked the speaker the URL also got changed which enables us to bookmark the URL based on the view.

In both the URL, one thing we notice that route that we defined got added with # which also helps in conflicting the other routing like ASP.NET MVC routing.

## Conclusion

Today we discussed about Views and Routes. These two are very important concept if we want to have multiple Views on a same web page. Routes are defined for each View where we provide the path of the View and Controller. When the URL is opened, corresponding View and Controller is loaded.

# Day 6

# Views, Routing and Model

Last Day, we created a sample where we had a page that comprised two views: Talk details and Speaker details. We fetched two sets of data (Talks and Speakers) from server via AJAX call and rendered on the page using templates with the help of Angular rendering engine. Today, we are going to use again the same application and use Angular Model.

Till today, we read the data and displayed on UI, now we will post the data on server and see how it reflects on UI. As we are using an ASP.NET MVC application we will be posting data to an MVC Action using Angular AJAX services. For that we will create an additional view which allows us to add new Talk and we will see that how easily the data will be available on client side and posted on server.

## Posting Data to Server using Angular AJAX Services

Till today, we read the data and displayed on UI, now we will post the data on server and see how it reflects on UI. As we are using an ASP.NET MVC application we will be posting data to an MVC Action using Angular AJAX services. For that we will create an additional view which allows us to add new Talk and we will see that how easily the data will be available on client side and posted on server.

First we will see what all changes that we need to do in our pervious example.

1- Create new template for Add Talk.
2- Create a new Angular controller (say talkController.js) for the Add Talk view.
3- Add an ajax method in `EventsService` to post the data on server (Calls a MVC action method)
4- Create a new Action method say `AddTalk` in `EventsController.cs` at server side.
5- Define the route for `AddTalk` and add the link for the same.
6- Handle the AJAX response appropriately.

## Creating new template

So let’s start with first step. I have created the talk template (`AddTalk.html`) based on earlier model and it is as

```
<table>
    <tr>
```

```
        <td>Id</td>
        <td><input type="text" ng-model="talk.Id" class="form-control"
/></td>
    </tr>
    <tr>
        <td>Name</td>
        <td><input type="text" ng-model="talk.Name" class="form-control"
/></td>
    </tr>
    <tr>
        <td>Speaker</td>
        <td><input type="text" ng-model="talk.Speaker" class="form-
control" /></td>
    </tr>
    <tr>
        <td>Venue</td>
        <td><input type="text" ng-model="talk.Venue" class="form-control"
/></td>
    </tr>
    <tr>
        <td>Duration</td>
        <td><input type="text" ng-model="talk.Duration" class="form-
control" /></td>
    </tr>
    <tr>
        <td colspan="2" align="center"> <input type="button" ng-
click="add(talk)" value="Add New Talk"/></td>
    </tr>
</table>
```

I have put the above file in the template folder as earlier. Here I have used another directive `ng-model`. Let's discuss it briefly and another associated concept `ViewModel`.

## What is ViewModel

In earlier days, we used `$scope` various times and discussed it in brief that it acts as a glue between views and controllers. Let's describe it bit more.

`$scope` is kind of dynamic object which acts as an application model and it is accessible to views and controller. All the data/expression/code sharing is done via `$scope` between view and controller. It provides some more features like `watch` and `apply` APIs that we'll discuss in coming days.

On every input control, I have added an `ng-model` attribute, and provided the model name.

## What is ng-model

It is a directive that is used to bind the input (`select` and `textarea` as well) control to a property that on the current scope. If it finds the property already available then it uses the same else creates a new one dynamically. Binding with model is handled using `NgModelController`. It provides many other feature like validation, other various states of the data etc that we'll be discussing in coming days.

*Note : If you are adding the new properties and want to serialize it at server with server side model, make sure the property names used in the model, should match with server side model else it will not be populated.*

Above template also contains a button control that has ng-click attribute. This allows us put the function/expression that would be evaluated on button click. So here we have a provided a add method which takes *talk* as parameter. Now let's move to our angular controller.

## Creating an angular controller

In our earlier controllers, we added a method that gets fired on load and which further gets the data from server using a service and assign it to $scope. But here, we have different scenario, here we have to call a method on the button click that further sends the data to server with the help of other various components. So our controller could be written as (I have created a new JavaScript file named talkController.js)

```
eventModule.controller("talkController", function ($scope, EventsService)
{
    $scope.add = function (talk) {
        EventsService.addTalk(talk);
    };
});
```

So here we added a new property *add* to *$scope* which points to the method that handles Add Talk functionality. As *$scope* is available to the view so it can be accessed via ng-click directive. If we see the controller then we find that we call a method addTalk of the EventsService that takes an input parameter talk as well. So let's add this new method to EventsService.

## Adding method to Service

Here we will add a new method addTalk that will initiate an ajax request and post the data at server. So let's see the code

```
addTalk: function (talk) {
           // Initiates the AJAX call to the server
           $http({ method: 'POST', url:'/events/AddTalk', data: talk });
       }
```

As we can see that $http provides a simple way to Post the data to any URL. Here we are initiating an AJAX call and posting data (talk) AddTalk method on server.

## Adding MVC Action method to the server

From the above AJAX call, we can see that it calls a method AddTalk at the server. Let's add it in EventsController.cs accordingly.

```
[HttpPost]
public ActionResult AddTalk(TalkVM talk)
   {
        eventsRepository.AddTalk(talk);
        return new HttpStatusCodeResult(HttpStatusCode.OK, "Item added");
   }
```

Here we added an Action AddTalk and added HttpPost attribute on top of that so that it can be called via Post method only. Also corresponding method got added in repository.

## Define the route for Add Talk

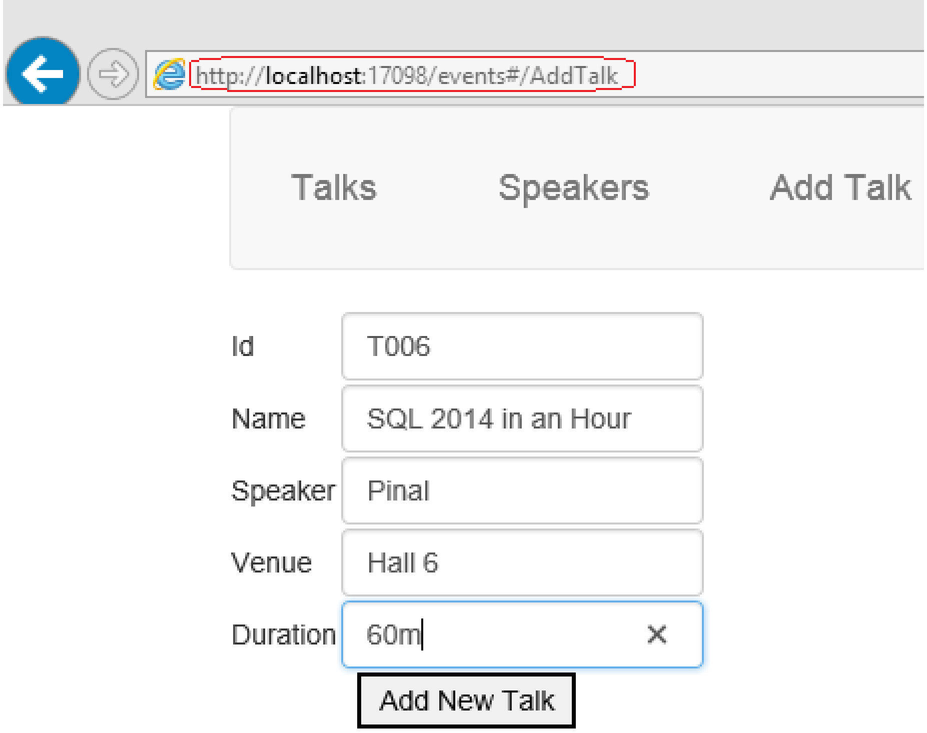
Now we need to define the route for Add Talk and provides the path of template and associated controller.

```
$routeProvider.when('/AddTalk', { templateUrl: '/Templates/AddTalk.html',
controller: 'talkController' });
```

Now let's add the link for AddTalk at our MVC view as

```
<li class="navbar-brand"><a href="#AddTalk">Add Talk</a></li>
```

Here I added a *li* with the link of Add Talk that points to the route that we defined for Add Talk in the module. Also make sure that new angular controller file talkController.js reference got added. Now we are done. Let's run our application



Id: T006

Name: SQL 2014 in an Hour

Speaker: Pinal

Venue: Hall 6

Duration: 60m

So when I clicked on *Add New Talk*, it successfully posted on the server. But it remained on the page without any message. Normally, if it gets successfully added then it should move to listing page and in case of failure, it should display a user friendly message. It can be achieved easily if we modify our code a bit. From our MVC Action, we are already sending a HTTP response code to the client. We just need to capture it and act accordingly.

So let's change our angular service as

```
addTalk: function (talk) {
    // Get the deferred object
    var deferred = $q.defer();
    // Initiates the AJAX call
    $http({ method: 'POST', url:'/events/AddTalk', data: talk
}).success(deferred.resolve).error(deferred.reject);
    // Returns the promise - Contains result once request completes
    return deferred.promise;
}
```

Here we are returning the promise object as used in earlier posts. Now we need to handle it accordingly in our controller. So let's see the updated controller

```
eventModule.controller("talkController", function ($scope, $location,
EventsService) {
    $scope.add = function (talk) {
        EventsService.addTalk(talk).then(function () {
$location.url('/Talks'); }, function ()
        { alert('error while adding talk at server') });
    };
});
```

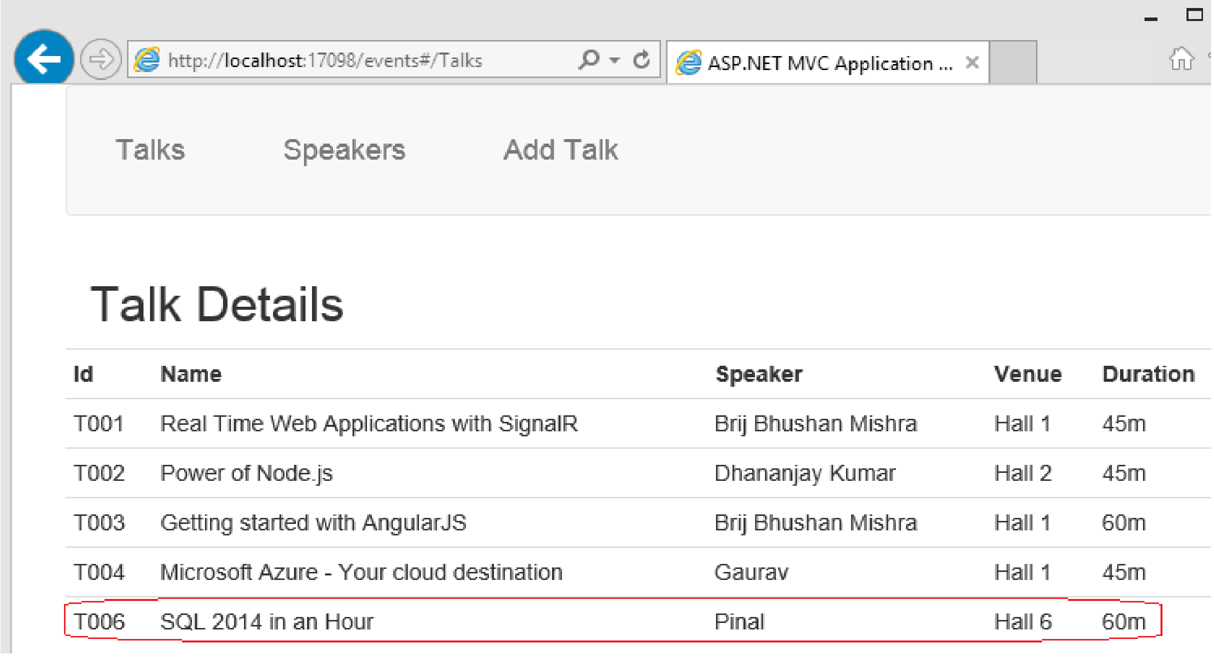
Here I have used one more inbuilt angular service called location service. Let's see what it is

### What is $location service

$location service is another angular inbuilt services, that parses the url in the browser and make it available for use. It provides mainly two features: change the url in the browser or do something when the url changes in the browser. WE can also understand similar like `window.location`.

So here we are loading the talk details in case of success and showing an alert message in case of failure.

Now when we click on Add New Talk on add talk form, it redirect to talk details as



http://localhost:17098/events#/Talks
ASP.NET MVC Application ...
Talks
Speakers
Add Talk

Talk Details

| Id | Name | Speaker | Venue | Duration |
|---|---|---|---|---|
| T001 | Real Time Web Applications with SignalR | Brij Bhushan Mishra | Hall 1 | 45m |
| T002 | Power of Node.js | Dhananjay Kumar | Hall 2 | 45m |
| T003 | Getting started with AngularJS | Brij Bhushan Mishra | Hall 1 | 60m |
| T004 | Microsoft Azure - Your cloud destination | Gaurav | Hall 1 | 45m |
| T006 | SQL 2014 in an Hour | Pinal | Hall 6 | 60m |

So here we can see the talk got added successfully

### Conclusion

Today we took the example of earlier post and added another feature to Post data to MVC action. We discussed about `ViewModel`, `ng-Model` and `$location` which we used in our example. At last, we saw that how to handle the response from AJAX and take the appropriate action.

# Day 7

# Data Binding in AngularJS

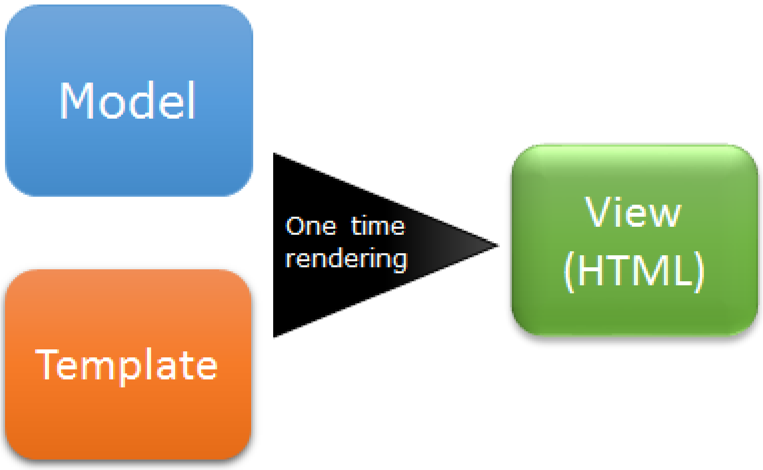
Today we understand Databinding with examples. Databinding is very powerful in Angular and saves lots of time else we might require to track each UI element and model at client side.

Broadly, Data binding can be divided in the two parts.

1. One way binding
2. Two way binding

## What is one way data binding?

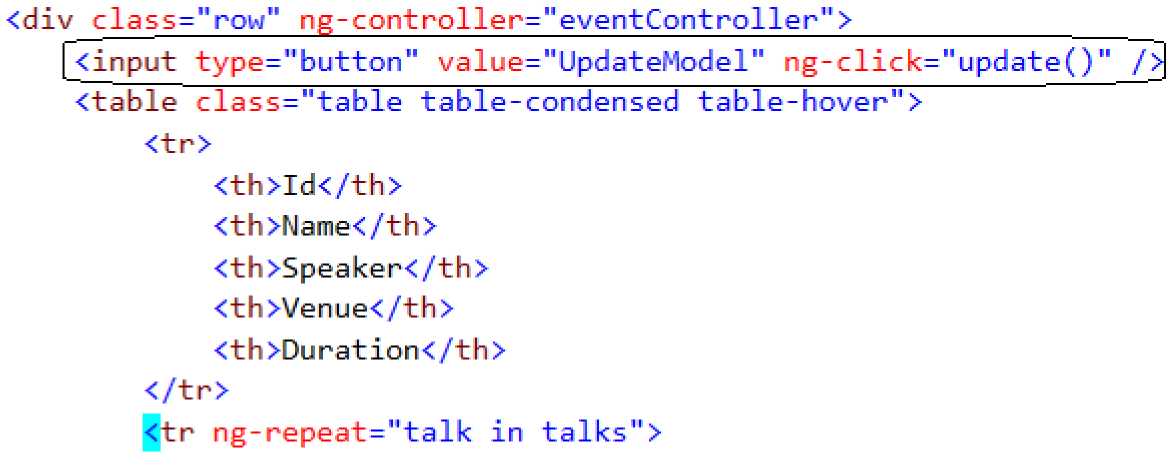
You must have seen/used *One way binding* at several times while using different templating mechanism. In a normal scenario, model (can be retrieved some services/database or generated on the fly) and template, is used to render to view. Let's see it pictorially



From the above image, we can understand that the model and template is using by the rendering engine and corresponding html is generated. This works as one time merging. But what would happen if the underlying data gets changed? Will that reflect on UI in the above scenario? Off course not because it is just one time rendering.

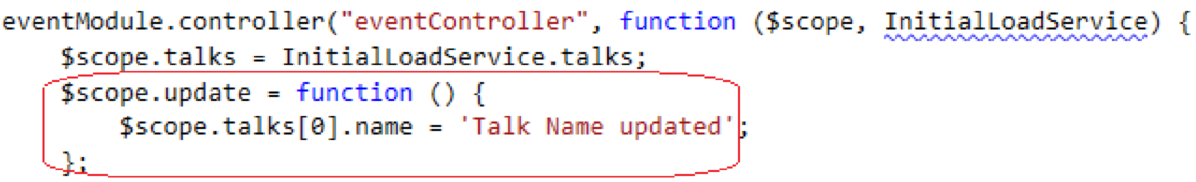
## Working with one way binding

For this demo, I am going to use the same example application that we created on 3rd day. There we had a list of Talks that we displayed using `ng-repeat`. It was an example of one way binding. Here I am going to show you another flavor of one way binding. Let's see the view (`Index.cshtml`)



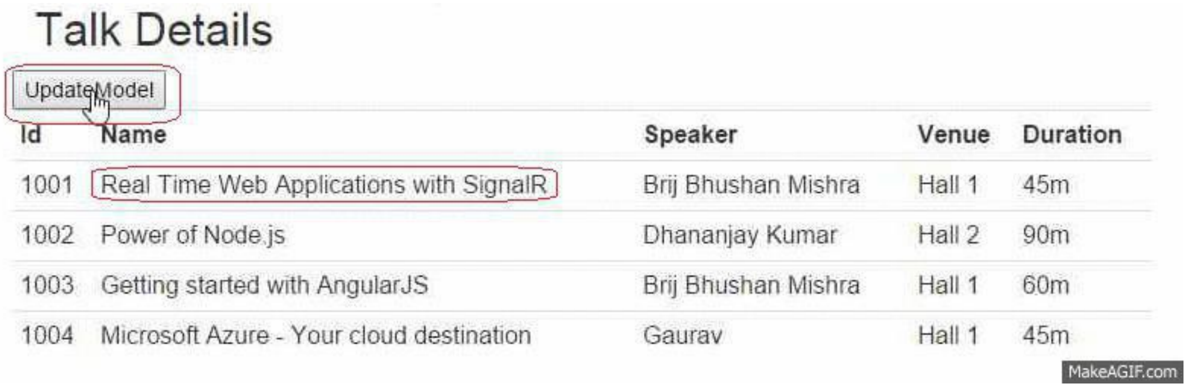
```
<div class="row" ng-controller="eventController">
    <input type="button" value="UpdateModel" ng-click="update()" />
    <table class="table table-condensed table-hover">
        <tr>
            <th>Id</th>
            <th>Name</th>
            <th>Speaker</th>
            <th>Venue</th>
            <th>Duration</th>
        </tr>
        <tr ng-repeat="talk in talks">
```

In the above screenshot, I have encircled that I added a button that calls an update method. Now let's see the definition. It is in `EventController.js` as



```
eventModule.controller("eventController", function ($scope, InitialLoadService) {
    $scope.talks = InitialLoadService.talks;
    $scope.update = function () {
        $scope.talks[0].name = 'Talk Name updated';
    };
```

Here I have provided the definition of the update method and just updating name of the first object in this list for this demo. Now let's run the application



Talk Details

| Id | Name | Speaker | Venue | Duration |
|---|---|---|---|---|
| 1001 | Real Time Web Applications with SignalR | Brij Bhushan Mishra | Hall 1 | 45m |
| 1002 | Power of Node.js | Dhananjay Kumar | Hall 2 | 90m |
| 1003 | Getting started with AngularJS | Brij Bhushan Mishra | Hall 1 | 60m |
| 1004 | Microsoft Azure - Your cloud destination | Gaurav | Hall 1 | 45m |

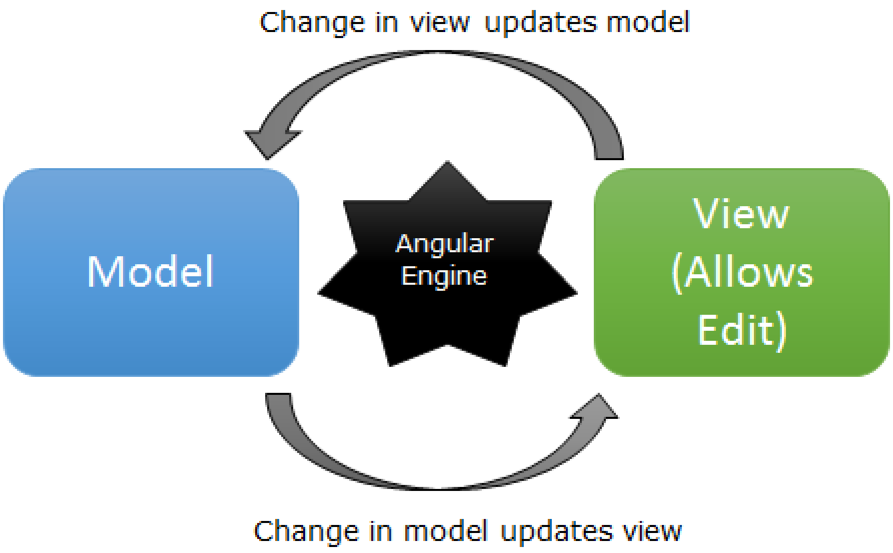
The list of talks contains four objects and rendered on browser using one way binding. When we click on `UpdateModel` button, name in the first row got updated. And I just wrote the code to update the model in the method (not the UI element). So if your model gets updated any how then Angular makes sure that it reflects the same on UI. This is the power of AngularJS.

## What is two way data binding?

Let's understand the two way binding in simple words

1- If the underlying data gets changes then corresponding UI elements gets changed.
2- When user enters/updates the UI elements then underlying model gets updated.

So first one, itself called as one way data binding that we discussed in my first section of the post. So how does two way binding works



From the above picture, we can understand that Angular makes sure that whenever a model changes, view gets updated and vice-versa. In this flow, controller is not involved at all.

*Note- In the above picture, I have not shown that how initially view gets rendered using template. It is same as one way binding*

To provide the above features, AngularJS provides another directive call ng-model that should be applied to each element on UI for which we want to use two way binding feature.

## Working with Two Way Binding

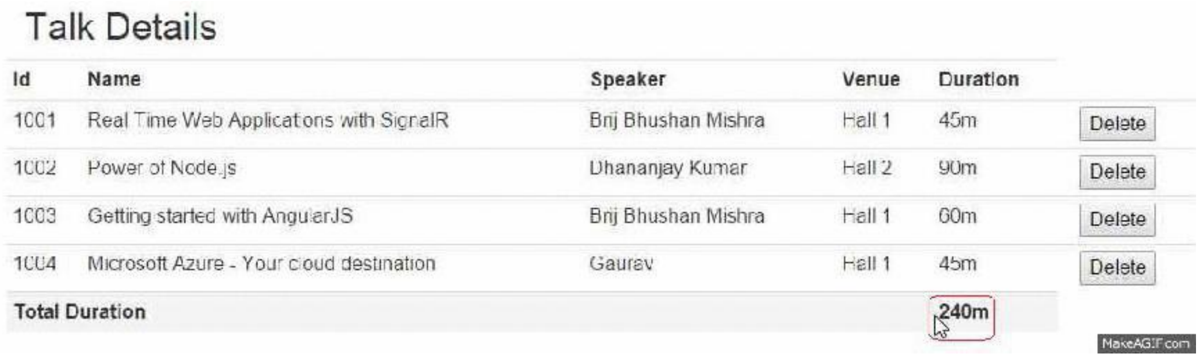
As I mentioned in above post that to implement two way data binding, a directive ng-model is provided, it makes sure the underlying object gets updated as soon as it gets updated from UI.

```
<tr ng-repeat="talk in talks">
    <td> {{talk.id}}</td>
    <td> {{talk.name}}</td>
    <td> {{talk.speaker}}</td>
    <td> {{talk.venue}}</td>
    <td ng-click="ShowEdit()" ng-hide="editDuration">
{{talk.duration}}m</td>
    <td> <input type="text" ng-show="editDuration" ng-
model="talk.duration"
    ng-mouseleave="HideEdit()" /></td>
    <td><input type="button" value="Delete" ng-click="deleteItem($index)"
/></td>
</tr>
<tr><td colspan="4" style="font-weight: bolder"> Total Duration</td>
    <td style="font-weight: bolder"> {{TotalDuration()}}m</td></tr>
```

In above code, for duration I have added two columns (td), one contains the value (view only) and another contains an input control but showing only one td at a time. One new directive, I used in input control – `ng-model`, it enables two way binding. I added another column which has a delete button which deletes a row. On click of delete button, `deleteItem` is called which deletes an item from the model. Also I added another row (`tr`) which show the sum of values of duration column. These new methods are added in the controller. Let's see it

```
$scope.editDuration = false;

  $scope.ShowEdit = function () {
      $scope.editDuration = true;
  };

  $scope.HideEdit = function () {
      $scope.editDuration = false;
  };

  $scope.deleteItem = function (index) {
      $scope.talks.splice(index, 1);
  }
  $scope.TotalDuration = function () {
      var sumofDuration = 0;
      for (var i in $scope.talks) {
          sumofDuration += parseInt($scope.talks[i].duration);
      }
      return sumofDuration;
  }
```

Here `ShowEdit` and `HideEdit` are simple, I am just setting a variable and based on that showing the View only or editable cell in the table. In `deleteItem`, index of the item is passed and the item is deleted from underlying model. In `TotalDuration`, I am just adding all the duration from the underlying model and returning it that is shown on `TotalDuration` row.



Talk Details

| Id | Name | Speaker | Venue | Duration | |
|---|---|---|---|---|---|
| 1001 | Real Time Web Applications with SignalR | Brij Bhushan Mishra | Hall 1 | 45m | Delete |
| 1002 | Power of Node.js | Dhananjay Kumar | Hall 2 | 90m | Delete |
| 1003 | Getting started with AngularJS | Brij Bhushan Mishra | Hall 1 | 60m | Delete |
| 1004 | Microsoft Azure - Your cloud destination | Gaurav | Hall 1 | 45m | Delete |
| Total Duration | | | | 240m | |

In the above example, we can see when we click the delete button that deletes the row which also updates total row as it depends on the underlying model. Similarly, when we edit a row and change the value to 190, Total Duration again gets changed.

## Conclusion

Today we have discussed the Databinding in details. Angular supports two types: One way binding and two way binding. Both can be easily coded and very powerful. Next day we will discuss that how all work behind the scene.

# Day 8

# Data Binding – Under the Hood

Last day we discussed about the Databinding capabilities in AngularJS. Today we are going to discuss more about this and see that how actually it works behind the scene. This concept is very important as it will help in learning other topics in future.

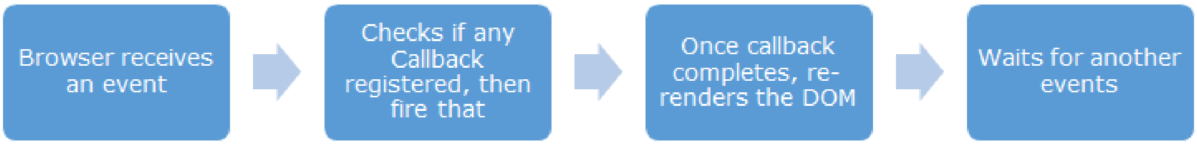
Before moving directly to the concepts behind the scene, let's discuss prerequisites that will help in understanding the whole concept. So let's start from the scope.

## What is Scope?

Scope is the core of an Angular application. It refers to an application model and execution context. An application has single `$rootscope` and multiple $scope objects associated to it. `$scope` is accessible between view and controller and it is the only medium of communication between both. It is like a container and holds all the other required information that is used by Angular for providing other features. You'll see some in this post.

## How Event handling works

First, let's understand that how browser identifies an event and takes the necessary actions. The flow of an event handling works as


Browser receives an event
Checks if any Callback registered, then fire that
Once callback completes, re-renders the DOM
Waits for another events

Events could be fired in many scenarios like page load, some response comes from server, mouse move, button click, timer events etc.

When we use Angular in an application, it modifies the normal JavaScript event flow and provides its own event handling model. This event handling model comes in flow when event is processed through Angular Execution context. It also provide many other features like exception handling, property watching etc. All the events that we define using angular constructs, is part of Angular Execution context. But you must have seen that when we try to use normal JavaScript or jQuery event then it does not fire. Hold On!! We will discuss it and its workaround at latter part of the day.

So we have understood the basics, now let's come to the main topic and discuss how data-binding works in Angular. There are two steps involved in it.

1. Angular creates a `$watch list` that is available under scope `$$watchers`. Every element that access any data in UI using some directive, a watch is created for that and got a place in this list.
2. Whenever an event (already discussed earlier in this post) occurs, all the watchers get iterated (also known as digest loop) and it is checked if any of the data got changed (called dirty-checking). If yes then it updates the UI or the underlying model.

Let's understand the above points in detail. The key concepts are

1. $watch and $watchlist
2. $digest loop
3. $apply (Looking new? Got discussed in detail later section)
4. Dirty checking etc

## What is $Watch list?

It is the key of data binding and as discussed above that when an item is accessed in UI via angular directive, an entry gets created in the watch list for that item. Now can you imagine the number of watchers in an application? It could be way high than what you think. We'll see that in coming section. Let's see some examples

### Example – 1:

`$watch` gets created for elements that are bound to UI. Say I have created some properties as

*JS (controller)-*

```
appModule.controller('myController', function ($scope) {
    $scope.message1 = 'message1';
    $scope.message2 = 'message2';
    $scope.helloMessage = 'Hello';
});
```

*HTML (View)-*

```
<div ng-controller="myController">
    Message - {{helloMessage}}
</div>
```

So here only one entry will be created in the watch-list as only `helloMessage` is accessed in UI. Other properties like *message1, message2* that are part of *$scope* but watchers would not be created for those as there are not accessed in UI.

### Example – 2:

For each `ng-model` attribute, one item gets created in watch-list. As

```
<input type="text" ng-model="talkName" />
<input type="text" ng-model="speakerName" />
```

Here, we have two input controls with *ng-model* as `talkName` and `speakerName`. For both, there will be an entry get created in watch list. `$scope.talkName` and `$scope.speakerName` are bound to first and second input respectively.

### Example – 3:

We have used ng-repeat in our earlier post, which renders the elements in UI based on the available list of items. How many watchers will be created in that case? Let's take example from the last day, where I have shown the list of talks on UI by using ng-repeat.

*JS (controller)-*

```
eventModule.controller("eventController ", function ($scope,
EventsService) {
    EventsService.getTalks().then(function (talks) { $scope.talks = talks
}, function ()
    {alert('error while fetching talks from server') })
});
```

*HTML (View)-*

```
<div class="row">
    <table class="table table-condensed table-hover">
        <tr ng-repeat="talk in talks">
            <td> {{talk.id}}</td>
            <td> {{talk.name}}</td>
            <td> {{talk.speaker}}</td>
            <td> {{talk.venue}}</td>
            <td> {{talk.duration}}</td>
        </tr>
    </table>
</div>
```

How many watchers get created for the above?
It would be 5*6 = 30. (Properties bound in UI * number of items in the list).

As I mentioned in my post earlier, number of watchers could be very high because if you have used many ng-repeat in your page then it would grow exponentially.

### Example – 4:

We can also create the watch programmatically. Let's see the example

*HTML (View)-*

```
<div ng-controller="myController">
    <input type="button" value="UpdateHelloMessage" ng-
click="updateHelloMessage()" />
    {{message1}}
</div>
```

*JS (controller)-*

```
appModule.controller('myController', function ($scope) {
    $scope.message1 = 'Test Message';
    $scope.helloMessage = &quot;Hello&quot;;

    $scope.updateHelloMessage= function () {
        $scope.helloMessage = &quot;Hellow World!!&quot;;
        console.log($scope.helloMessage);
    };

    $scope.$watch('helloMessage', function (newVal, oldVal) {
       if(newVal != oldVal)
            $scope.message1 = &quot;Hello message updated&quot;;
    });
});
```

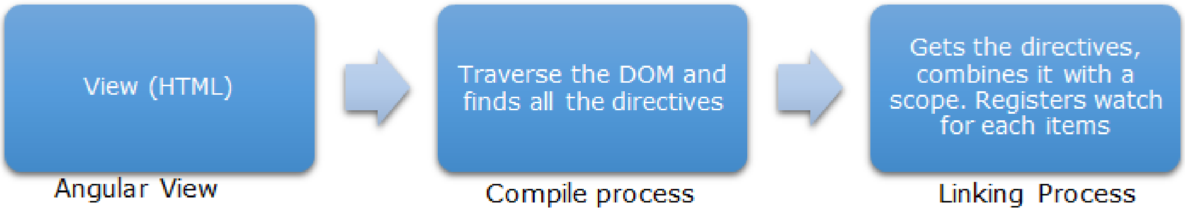
If we see the HTML then we'll say that there are only one watch (for *message1*) in this app. But it has two as I had added a watch on $scope.helloMessage programmatically.

When we click on button `UpdateHelloMessage`, `$scope.message1` also get updated due to new watch created and gets reflected on UI.

Before moving to next topic, let's understand when does an entry get created and added in watch list.

## When does an entry get added in the watch list?

Angular uses its own compiler which traverses the UI to get all the angular directives used. This is how angular directives used in the HTML, are understood and accordingly rendered in UI. Angular process can be depicted as


View (HTML)
Angular View
Traverse the DOM and finds all the directives
Compile process
Gets the directives, combines it with a scope. Registers watch for each items
Linking Process

As we can see there are two main steps involved: Compile and Linking.

### Compile process

It traverses the DOM and collects all the directives and passes it to the linking function.

### Linking Process

It actually combines all the directives and assigns to a scope. It sets up `$watch` expression for each directive. `$watch` actually responsible to notify the directive in case of a change in property of $scope, then it get reflected in UI. Similarly, if any changes occurs in UI as by some user interaction, it get reflected in model. Here one point to be noted that both controller and directive has the ability to access the scope but not each other.

So here we can understand the entries in watch list, get created in linking phase. And this is where it knows the number of watch need to be created in case of *ng-repeat*. We'll see some examples later.

## Understanding $digest loop

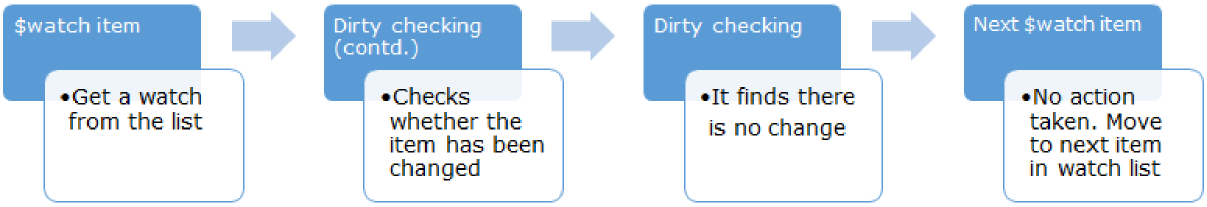
Earlier in this post, I mentioned that whenever an event occurs on browser, digest loop gets fired. It contains two smaller loops, one processes $evalAsync queue and other iterates the watch list for each item, to detect the changes. Then, it finds all the items which got changed and UI gets updated accordingly.

*Note – The $evalAsync queue is used to schedule work which needs to occur outside of current stack frame, but before the browser's view render. It provides the similar features provided by setTimeout().*

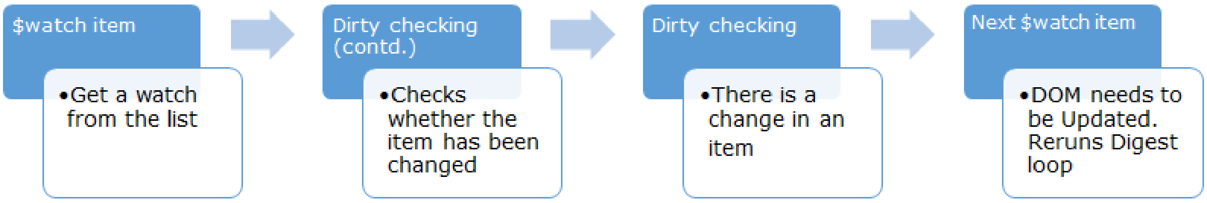
## What is dirty checking?

The process of checking every watch to detect the changes, is called dirty checking. There could be two scenarios

First


$watch item
•Get a watch from the list
Dirty checking (contd.)
•Checks whether the item has been changed
Dirty checking
•It finds there is no change
Next $watch item
•No action taken. Move to next item in watch list

Second


$watch item
•Get a watch from the list
Dirty checking (contd.)
•Checks whether the item has been changed
Dirty checking
•There is a change in an item
Next $watch item
•DOM needs to be Updated. Reruns Digest loop

So till now you must have understood that magic of data binding occurs in AngularJS. We had one more key item left - apply that we didn't discuss in detail. What's the use of that?

## Understanding $apply

It actually plays pivotal role in the whole process. Angular does not trigger the digest loop directly, it has to be triggered via $apply call and this method is responsible to enter the execution in Angular Execution context. Only model modifications which execute inside the $apply method is properly accounted for by Angular. So now question arises that even we did not call the $apply in any earlier examples still it is working as expected. Actually Angular does it for us. Angular wraps every event with apply so that digest loop gets fired.

## What other uses are of apply method?

As we have already seen that angular wraps every event with apply that fires digest loop. It means if we want to run the digest loop in some scenarios then we need to execute the apply method. You must have seen many times that whenever we make some changes using JavaScript or jQuery methods it is not reflected in Angular. In those scenarios apply method provides entry in Angular Execution context. Let's see it with an example.

*HTML (View)-*

```
<div ng-controller="myController">
    <input type="button" value="UpdateHelloMessage" ng-
click="updateHelloMessage()" />
    {{helloMessage}}
</div>
```

*JS (controller)-*

```
appModule.controller('myController', function ($scope) {
    $scope.helloMessage = &quot;Hello&quot;;

    $scope.updateHelloMessage = function () {
        setTimeout(function() {
            $scope.helloMessage = &quot;Hello World!!&quot;;
        }, 0);
    };
});
```

So what do you think about the above code? Would it work as expected? I mean when you click on the UpdateHelloMessage button, updated $scope.helloMessage (Hello World!!) would reflect on UI.
No!! It won't work.

Although the model $scope.helloMessage will get updated but it won't appear on UI because digest loop won't run in this case. But why digest loop won't run? Because the value is updated in setTimeout method which is JavaScript method and it won't run in Angular context. So to get it running as expected we need to call $scope.$apply() and that would trigger digest loop and updated value will be shown on UI. So we can update the JS as

```
appModule.controller('myController', function ($scope) {
    $scope.helloMessage = "Hello";

    $scope.updateHelloMessage = function () {
        setTimeout(function() {
            $scope.helloMessage = "Hello World!!"
            $scope.$apply();
        }, 0);
    };
});
```

Now it will work as expected. Now you must have understand that there is no magic but a simple logic is written to do that.

## Conclusion

Today we discussed many important concept that are very important. We got to know that how the Data Binding actually works and discussed about $watch, $watchlist, $digest loop, $apply, dirty checking etc. with examples. Do try by yourself to understand it better and read one more time if needed.

# Day 9

# Creating Custom Directive

AS we know the directive is the core of AngularJS and being Angular very flexible and open, it allows us to write our own custom directive. We can easily our own project specific custom directive and use it as per out need. Also custom directive is a vast topic which cannot be covered in just day, I will span it in couple of days. So we'll start from basics of it.

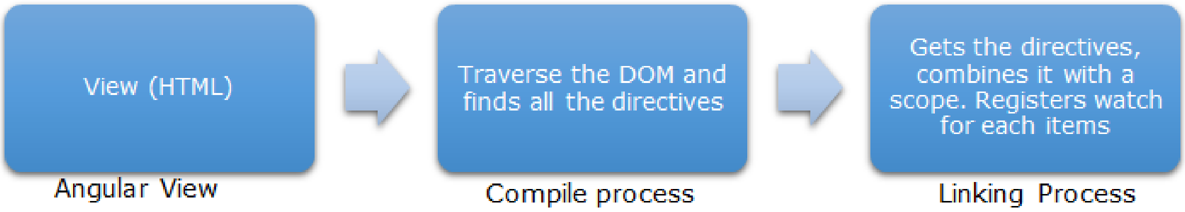
## What are Directives?

From Angular's documentation "*At a high level, directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell AngularJS's* **HTML compiler** *(*$compile*) to attach a specified behavior to that DOM element or even transform the DOM element and its children."*

So anything that we write in HTML that is handled by Angular are Directives like `{{}}`, `ng-click`, `ng-repeat` etc.

## How Directives are handled?

Directives provides us a capability to have more control and flexible way to rendering the html. As we know these are not standard HTML attribute so browser doesn't understand it. Angular chips in between and reads all the directives and process them to emit the browser readable HTML. So what is the process of handling directives? I already discussed that previous day, but I am including it here briefly.


View (HTML)
Angular View
Traverse the DOM and finds all the directives
Compile process
Gets the directives, combines it with a scope. Registers watch for each items
Linking Process

So there are two steps involved here: `Compiling` and `Linking`.

## Types of Directives

There are four types of directives. These are

1. Element directives
2. Attribute directives
3. CSS class directives
4. Comment directives

### Element Directives

Element directives are like HTML elements as

```
<myElementDirective></myElementDirective>
```

### Attribute Directive

Attribute directives are which can be added an attribute over an element as

```
<div myAttrDirective></div>
```

### CSS class Directive

These directive can be added as a CSS class

```
<div class="myAttrDirective: expression;"></div>
```

### Comment Directives

Comment directives is represented as

```
<!-- directive: myCustomAttrDirective expression -->
```

## How to create custom Directive

One of the key benefits of AngularJS is that apart from its built-in directives, it allows us to write our own custom directives so that we can render the HTML on browsers based on our specific requirement. Angular provides us a simple syntax to create our own custom directive.

```
var myElementDirective = function () {
var myDirective = {};

myDirective.restrict: 'E', //E = element, A = attribute, C = class, M =
comment
myDirective.scope: {
// To provide scope specific for that directive  },
myDirective.template: '&lt;mdiv&gt;This is custom directive&lt;/div&gt;',
myDirective.templateUrl: '/myElementTemplate.html',
// use either template or templateUrl to provide the html
myDirective.controller: controllerFunction, //custom controller for that
directive
myDirective.link: function ($scope, element, attrs) { } //DOM manipulation
}
```

The brief description of each property is given below

| Property Name | Description |
|---|---|
| `restrict` | Defines the type of directive. Possible values are E (element), A (Attribute), C (Class), M (Comment) or any combination of these four. |
| `scope` | Allows us to provide a specific scope for the element or child scopes. |
| `template` | This contains the HTML content that would be replaced in place of directive. It also could contain any other angular directive. |

| | |
|---|---|
| `templateUrl` | Allows us to provide the URL of the HTML template that contains the actual html content similar to above. Set any one of the two. |
| `controller` | Controller function for the directive. |

So we got the details about creating a custom directive. But the next question is – How to register it? As we discussed on Day 2 that Module works as container in AngularJS so the same applies here as well. It means we need to register it with module as directive. Let's see an example

### Example 1

I am going to create a simple element directive as

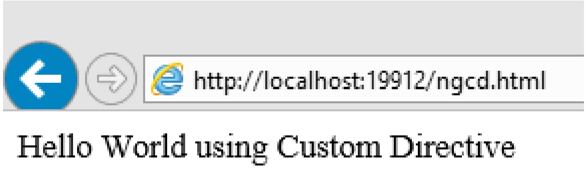
*JS -*

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

myangularapp.directive('myelementdirective', function () {

    var directive = {};

    directive.restrict = 'E'; //restrict this directive to elements

    directive.template = "Hello World using Custom Directive";

    return directive;

});
```

*HTML (View)-*

```
<body ng-app="myangularapp">
    <myelementdirective></myelementdirective>
</body>
```

Now when we run this page, it will appear as



http://localhost:19912/ngcd.html

Hello World using Custom Directive

Here we can see the element `myelementdirective` is replaced with the text. In this example, I have used only few properties that we discussed.

We can see in the above example that this is an element directive as *restrict* is set to `E` (Element). `restrict` property can be also set as any combinations of `E,A,C,M` like EA or EAC. If we set it as EA then compiler will find `myelementdirective` as an element or attribute and replace it accordingly.

We can also use `templateUrl` instead of template. `templateUrl` will pick the content of the file and use it as template.

### Processing of the Custom Directive

Let's see step by step

1- When the application loads, Angular Directive is called which registers the directive as defined.

2- Compiler find out any element with the name `myelementdirective` appears in the HTML. Here, it finds at one place.
3- Replaces that with the template.

So looks simple. Right!!

### Example 2

Let's see another example

*JS-*

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


myangularapp.directive('myelementdirective', function () {
    var directive = {};
    directive.restrict = 'E'; //restrict this directive to elements
    directive.template = "Hello {{name}} !! Welcome to this Angular App";
    return directive;
});
```

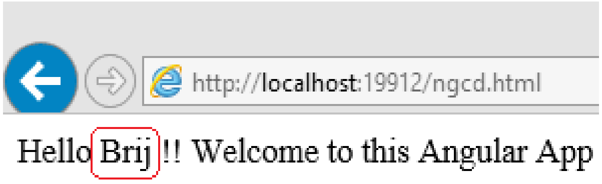
So here we used an interpolate directive `{{}}` in the template and used name. So we need to provide the value for this. So let's add a controller to handle that

```
myangularapp.controller("mycontroller", function ($scope) {
    $scope.name = 'Brij';
});
```

*HTML (View)-*

```
<body ng-app="myangularapp" ng-controller="mycontroller">
    <myelementdirective></myelementdirective>
</body>
```

Now when we run the application, would it work?



Hello Brij !! Welcome to this Angular App

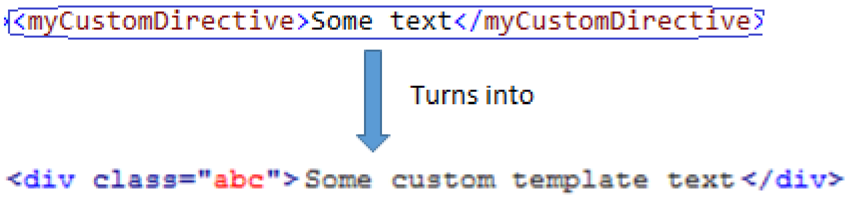
So it did work. But in this case, what would be the flow?

As I mentioned earlier that there are steps involved while rendering a page. `Compiling` and `linking`. All the data population takes place in linking phase. So during compilation phase it replaced the directive and in linking phase the data got populated from the scope. So earlier steps would be there, just one more steps would be inserted in between 2nd and 3rd.

## Using Transclude

This word may look complex but it provides a very useful feature. Till now we have seen that custom directive are just replaced by the template that we provide while creating custom directive but there could be some scenario where we do not want to replace the whole html but only some part of it. Like say I want that the text that is wrapped by the custom directive should not get replaced but only element part. Didn't get? Let's see pictorially

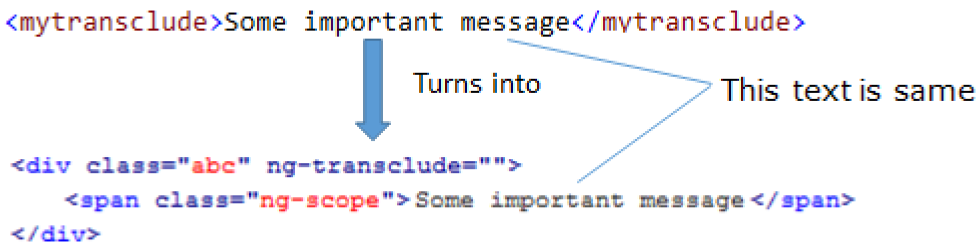
Say I created a custom element directive as `myCustomDirective` and the template provided for it as *<div>Some custom template text<div>*



```
<myCustomDirective>Some text</myCustomDirective>
```

```
<div class="abc">Some custom template text</div>
```

In case of `transclude`



```
<mytransclude>Some important message</mytransclude>
```

```
<div class="abc" ng-transclude="">
    <span class="ng-scope">Some important message</span>
</div>
```

*Note – Here I changed the template as <div ng-transclude>Some custom template text</div> to get the* `transclusion` *working.*

So in the above pic, we can see that the text is not replaced while in case normal directive the whole template replaces the directive. Now we understood the `transclusion` so let us see what all we need to do to use it. There are two steps

1- Set the `transclude` property to true while creating directive.
2- We need to inform Angular that which part contains the transcluded html that's why for transclusion example, I changed the template a bit (Refer Notes above)

## Conclusion

Today we discussed about creating custom directive. We started from the basics of directives, and then custom directives and it types. There are four types of custom directives and we created couple of example with different flavor. Last, we discussed about transclusion which is provides us more power to customize more the directives and can be very helpful in many scenarios.

# Day 10

# Custom Directive with Isolate Scope

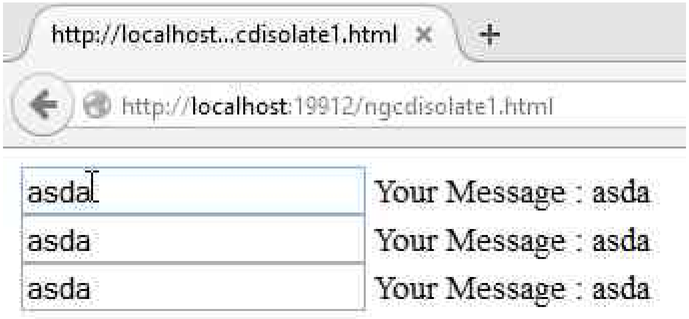
Today we will continue our discussion on custom directive and learn some more features associated with it. We have discussed the basics of custom directive but the real value of a custom directive, if it is reusable and can be independently used at many places. If you are using the parent scope directly in your directive then it won't be reusable. One more side effect, if the parent scope gets updated then your custom directive will also be affected even if you don't want. Not clear? Let's see an example

```
var myangularapp = angular.module(&amp;quot;myangularapp&amp;quot;, []);
myangularapp.directive('customdirective', function () {
    var directive = {
    restrict : 'E', // restrict this directive to elements
    template: &amp;quot;&amp;lt;input type='text' placeholder='Enter
message...' ng-model='message' /&amp;gt;&amp;quot; +
        &amp;quot; Your Message : {{message}}&amp;quot;
    };
    return directive;
});
```

I have created a directive where user is allowed to enter some message as it and created multiple instances of same directive as

```
<customdirective> </customdirective>
<customdirective> </customdirective>
<customdirective> </customdirective>
```

Now let's run it



http://localhost...cdisolate1.html

http://localhost:19912/ngcdisolate1.html

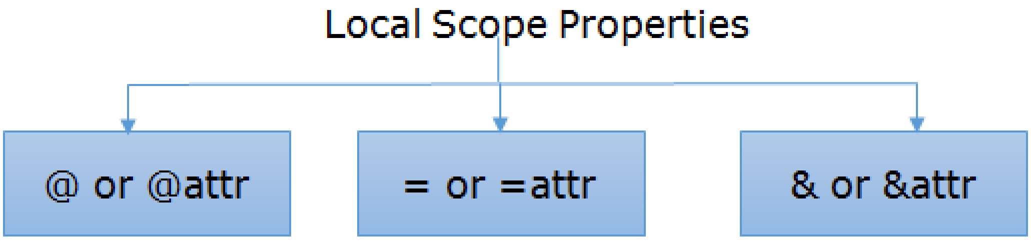
| | |
|---|---|
| asda | Your Message : asda |
| asda | Your Message : asda |
| asda | Your Message : asda |

Here we can see that even we are changing the values in one directive but it's getting reflected in all the directives. It is because all are accessing the same scope. Certainly we don't want this. This problem can be resolved using isolate scope.

## What is isolate scope?

The problems that we discussed above, can be resolved by using `isolate` scope. We can isolate the scope easily that is used in the custom directive. To add isolate scope, we need to add the scope property while defining the directive. It also makes sure that parent scope is not available to the custom directive.

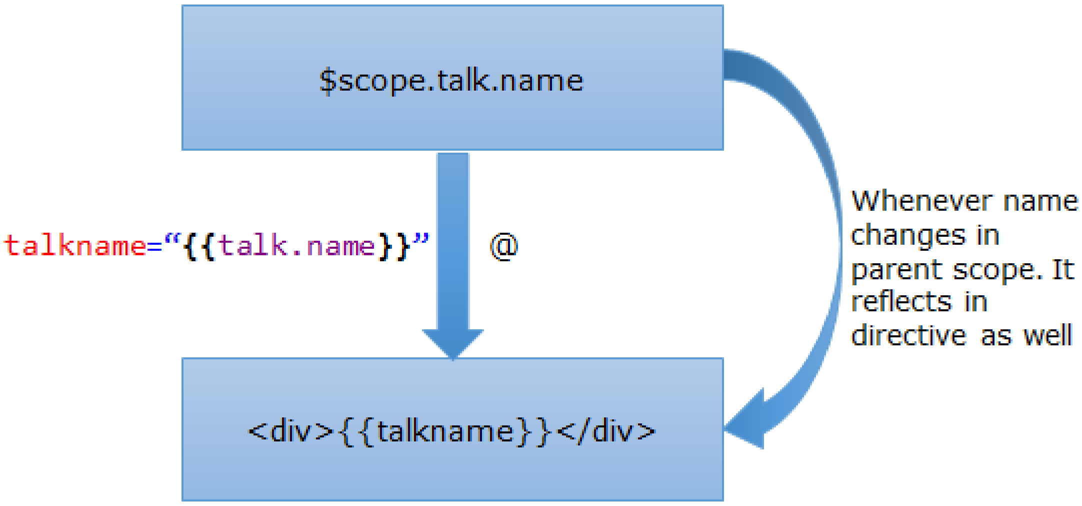
But now the question arises how will the directive interact with the outside world? Because the directive cannot be useful if it works in complete isolation. To handle it, Angular provides us three options to communicate to isolate directive that is also known as `Local Scope properties`. These are



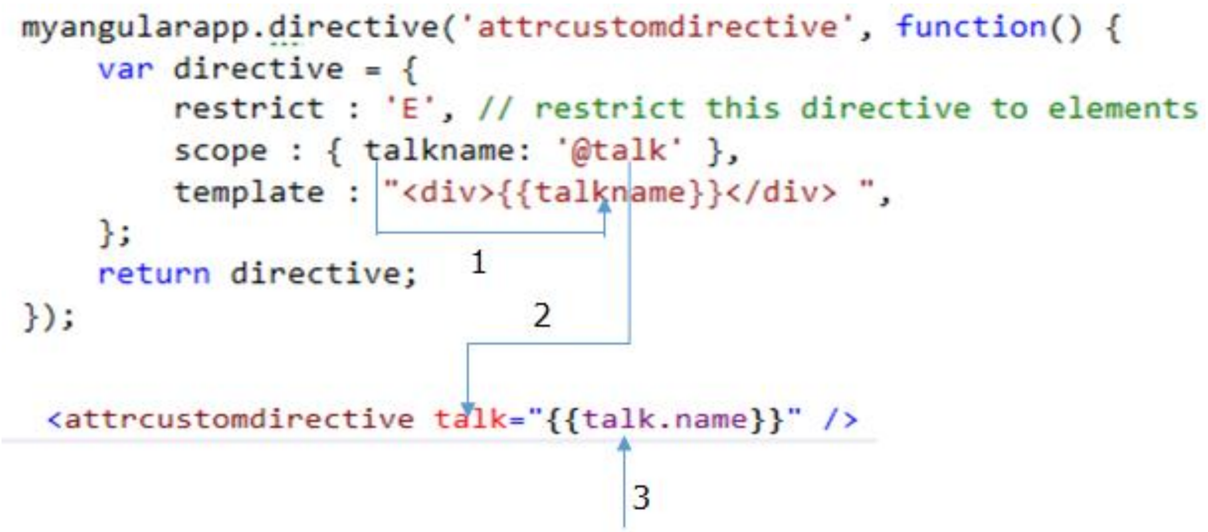
Let's discuss each in details.

### @ or @attr

@ provides us capability to pass a value to custom directive in `string format`. I have made the string format as bold because I wanted to highlight it, here we cannot pass an object. If you pass an object it will be treated as string format only and accordingly displayed. Also, it works like one way binding it means if the data changes in parent scope, then it reflects in the directive as well.


$scope.talk.name
talkname="{{talk.name}}"
@
<div>{{talkname}}</div>
Whenever name changes in parent scope. It reflects in directive as well

So here we find that a change takes place in parent scope, it reflects in directive itself but vice versa is not true. If it gets changed inside directive, parent scope does not get affected. How to create the directive?



```
myangularapp.directive('attrcustomdirective', function() {
    var directive = {
        restrict : 'E', // restrict this directive to elements
        scope : { talkname: '@talk' },
        template : "<div>{{talkname}}</div> ",
    };
    return directive;
});

  <attrcustomdirective talk="{{talk.name}}" />
```

So here we have created a directive and used it. The details of three points (in pic) are

1- `talkname` is a property in the isolate scope and it is only accessible in directive itself
2- `@talk` means that this would be the attribute name in the custom directive that will be used to communicate. We can also write `scope: { talkname: '@' }`, in this case, `talk` and `talkname` both would be same. In this example, I have used different name for better understanding.
3- This is passed from parent scope. As I mentioned in earlier section, that if parent changes then it would reflect in directive as well.

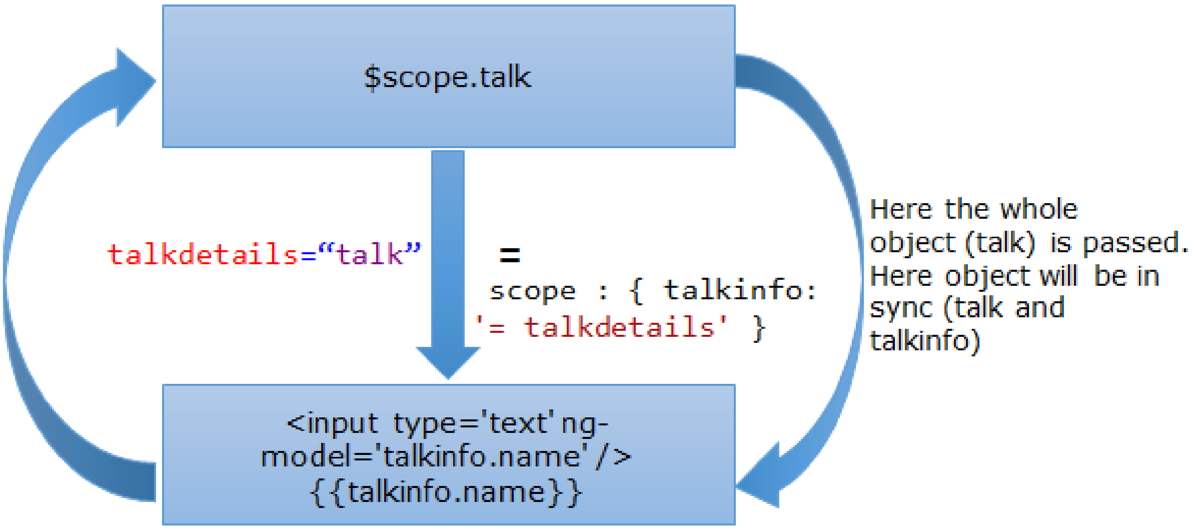
Let's see the complete code

```
<script language="javascript" type="text/javascript">
    var myangularapp = angular.module("myangularapp", []);
    myangularapp.controller("mycontroller", function ($scope) {
        $scope.talk = { name : 'Building modern web apps with ASP.NET 5',
duration : '60m'}
    });
    myangularapp.directive('attrcustomdirective', function() {
        var directive = {
            restrict : 'E', // restrict this directive to elements
            scope : { talkname: '@talk' },
            template : "<div>{{talkname}}</div> ",
        };
        return directive;
    });
</script>
</head>
<body ng-app="myangularapp" ng-controller="mycontroller">
    <attrcustomdirective talk="{{talk.name}}" />
</body>
```

Now let’s move to another type.

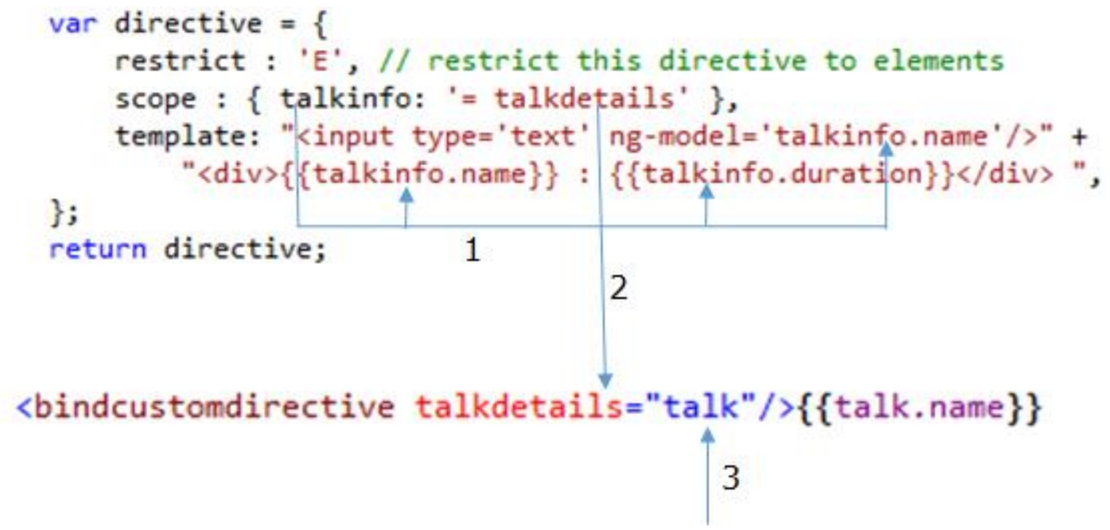
= or =attr

Unlike @ property which allows us to pass only string value, it allows us to pass the object itself to directive. And the data also gets synced in parent and child scope like two data binding. If the data changes from inside the directive it reflects in parent scope.



Here the whole object (talk) is passed. Here object will be in sync (talk and talkinfo)

Here `talk` (whole object) is passed in directive and assigned to `talkinfo`. Now whether the talk or `talkinfo` gets updated both remains always in sync



```
var directive = {
    restrict : 'E', // restrict this directive to elements
    scope : { talkinfo: '= talkdetails' },
    template: "<input type='text' ng-model='talkinfo.name'/>" +
        "<div>{{talkinfo.name}} : {{talkinfo.duration}}</div> ",
};
return directive;
```

```
<bindcustomdirective talkdetails="talk"/>{{talk.name}}
```

We can see from above that how the directive got created with = and its uses. The details of three points (in pic) are

1- `talkinfo` here is the object that that got received via `talkdetails`. You can see, I have accessed the value of `talkinfo` three times via its properties.
2- `talkdetails` is attribute name that is used to pass the object via directive. Similar as earlier if we don't provide the attr as `scope : { talkinfo: '=' }` then the attribute name will be `talkinfo` only.
3- `talk` is the scope object that is assigned to `talkdetails`.

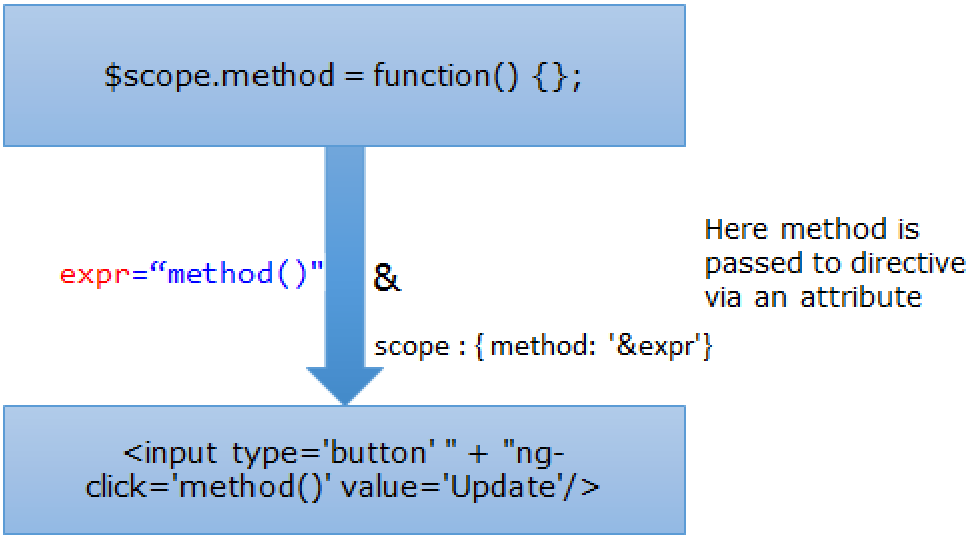
The complete example will be as

```
<script language="javascript" type="text/javascript">
    var myangularapp = angular.module("myangularapp", []);
    myangularapp.controller("mycontroller", function ($scope) {
        $scope.talk = { name : 'Building modern web apps with ASP.NET5',
duration : '60m'}
    });
    myangularapp.directive('bindcustomdirective', function() {
        var directive = {
            restrict : 'E', // restrict this directive to elements
            scope : { talkinfo: '= talkdetails' },
            template: "<input type='text' ng-model='talkinfo.name'/>" +
                "<div>{{talkinfo.name}} : {{talkinfo.duration}}</div> ",
        };
        return directive;
    });
</script>
</head>
<body ng-app="myangularapp" ng-controller="mycontroller">
    <bindcustomdirective talkdetails="talk"/>{{talk.name}}
</body>
```

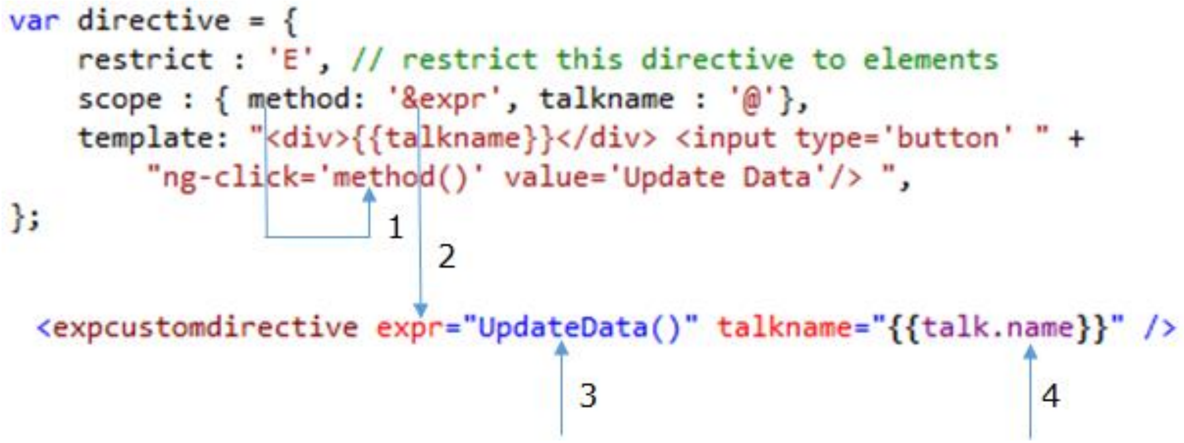
Let's move to last Local scope property.

### & or &attr

This is the third and last isolate local scope property. It allows us to wire up external expression to the isolate directive. It could be very useful at certain scenario, where we don't have details about the expression while defining the directive like we want to call some external function when some event occurs inside the directive based on requirement.


$scope.method = function() {};
expr="method()"
&
scope : { method: '&expr'}
<input type='button' " + "ng-click='method()' value='Update'/>
Here method is passed to directive via an attribute

In the above pic, we see that we have function with name method that is passed to directive as via the attribute named expr. Let's see how we create the directive and how different property and attributes are co-related.



```
var directive = {
    restrict : 'E', // restrict this directive to elements
    scope : { method: '&expr', talkname : '@'},
    template: "<div>{{talkname}}</div> <input type='button' " +
        "ng-click='method()' value='Update Data'/> ",
};

  <expcustomdirective expr="UpdateData()" talkname="{{talk.name}}" />
```

In the example above, I have used two directives & and @. @ is just used to support this example. Although you must have understand the three points that I have used in the above pic as it is similar to earlier but let me explain it once more in this context. In this example, we are updating an object that gets updated and reflected in the directive as well because of one way binding behavior of @ local scope property.

1- method is the property of inner scope here so it is used to access the passed method.
2- expr is the attribute name that is used in the directive to pass the expression or defined method. Behavior would be same as earlier local scope property if we just write scope : { method: '& '}.
3- UpdateData() is the method name that we want to pass in the directive and it is defined as part of parent scope.
4- This value gets updated when we click on *Update Data* button that calls UpdateData() method which updated the object Talk.

Let's see the complete example.

```
<script language="javascript" type="text/javascript">
    var myangularapp = angular.module("myangularapp", []);
    myangularapp.controller("mycontroller", function ($scope) {
        $scope.talk = { name: 'Building modern web apps with ASP.NET5',
duration: '60m' }

        $scope.UpdateData = function () {
            $scope.talk = {
                name: 'Working with AngularJS',
                duration: '45m'
            };
        };
    });
    myangularapp.directive('expcustomdirective', function() {
        var directive = {
            restrict : 'E', // restrict this directive to elements
            scope : { method: '&expr', talkname : '@'},
            template: "<div>{{talkname}}</div> <input type='button' " +
                "ng-click='method()' value='Update Data'/> ",
        };
        return directive;
    });

</script>
```

```
</head>
<body ng-app="myangularapp" ng-controller="mycontroller">
    <expcustomdirective expr="UpdateData()" talkname="{{talk.name}}" />
</body>
```

Now, you must have got good idea about local scope properties in isolate scope and will be able to decide easily that what to use.

## Conclusion

Today we took a step further and discussed about Isolate Scope in Custom Directives. Angular provides three local scope properties to isolate the scope based on different requirement. We discussed with example.

# Day 11

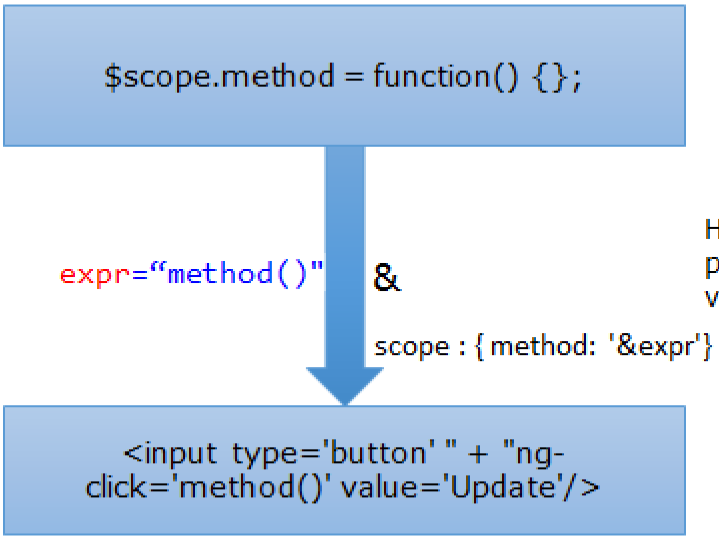
# Passing Values function in Isolate Scope

Today we are in eleventh day and third continuous day on Custom Directives. As I already mentioned earlier that it is pretty vast topic and a small book can be written on this, but I am trying to cover all the aspects that can be useful in our daily tasks. Previous day, we discussed about isolate scope which allows us to write the reusable custom directives. As we discussed that Angular provides three local properties to leverage isolate scope and last one allows us use expression in isolate scope, but what if we need to pass some parameters which sometimes we need. This is bit tricky. Today we will discuss it and see the options that we can use when required.

In Day 10, we discussed about various options available while creating isolate scope in custom directives and three different types of Local Scope properties. These are

1. `@` or `@attr`
2. = or `=attr`
3. & or `&attr`

As mentioned, let's continue with third option. It allows us to wire up some external expression to the custom directive as discussed in the last post. Let's see the revisit the pic about the third option in isolate scope


$scope.method = function() {};
expr="method()"
&
scope : { method: '&expr'}
<input type='button' " + "ng-click='method()' value='Update'/>
Here method is passed to directive via an attribute

## Passing parameter to expression

In above image, we are passing `UpdateData` expression in the isolate directive. Now let's see what we should do if we want to pass some values in this method. There could be multiple ways, I am going to discuss a few.

### First Option

In this scenario, let's say we have some data that is available in the custom directive itself that we need to pass to in external expression as parameter. For this, I added an input text box in template that I will be using the entered value as parameter for the external function. Let's first see the directive

```
<expcustomdirective expr="UpdateData(updatedName)"
talkname="{{talk.name}}" />
```

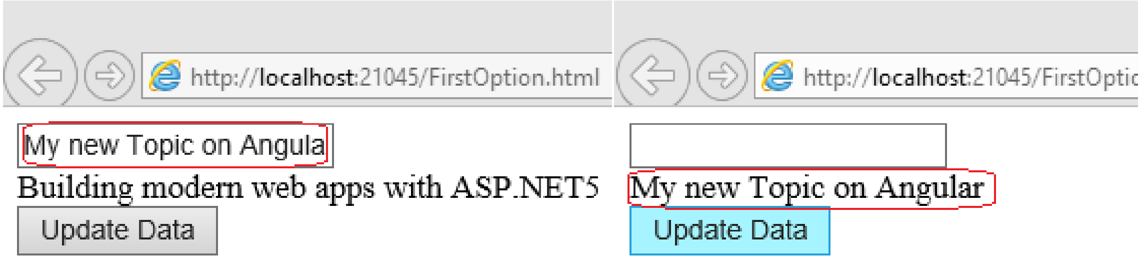
Here I am passing a parameter named `updatedName` in `UpdateData`. So let's change the `UpdateData` method accordingly

```
$scope.UpdateData = function (modifiedName) {
    $scope.talk = {
        name: modifiedName,
        duration: '45m'
    };
};
```

Now let's see the custom directive

```
myangularapp.directive('expcustomdirective', function () {
    var directive = {
        restrict: 'E', // restrict this directive to elements
        scope: { method: '&expr', talkname: '@' },
        template: "<input type='text' ng-
model='mytalkName'/><div>{{talkname}}</div> <input type='button' " +
            "ng-click='method({updatedName : mytalkName})' value='Update
Data'/> "
    }

    return directive;
});
```

In the template above, I added an input text box and provided the model (*ng-model*) name as `myTalkName`. Another important point (*ng-click*) to see here, how the value is passed. Here we need to provide an object map so that the value gets bound correctly as `{updatedName : mytalkName}`. It is very important else corresponding value won't be available in expression. Now let's run it



http://localhost:21045/FirstOption.html

My new Topic on Angula

Building modern web apps with ASP.NET5

Update Data

http://localhost:21045/FirstOptic

My new Topic on Angular

Update Data

We can see here that we can easily get a value that is part of custom directive itself and use it in external expression.

### Second Option

We discussed in first option that how can we pass some value which is generated in the directive itself. Let's take another scenario, where you need to pass the values that is available outside.

We know that the compilation process involves two steps in Angular – One is `compiling` and another is `linking`. `Linking` has the responsibility to assign the scope to directive. We can take the help of linking function to pass some values in the external function.

While writing to custom directive, we can attach a function to link as

```
link: function (scope, element, attrs) {

}
```

This gets called once for the custom directive. Here it has three parameters.

1- Scope - It is the isolated scope that is available for the directive.
2- Element - It is a custom directive element.
3- Attrs - List of attributes available in custom directive.

So let's update our custom directive.

```
myangularapp.directive('expcustomdirective', function () {
    var directive = {
        restrict: 'E', // restrict this directive to elements
        scope: { method: '&expr', talkname: '@' },
        template: "<div>{{talkname}}</div> <input type='button' " +
            "ng-click='method(updatedName)' value='Update Data'/> ",
        link: function (scope, element, attrs) {
            scope.method({ updatedName: "Updated topic name" });
        }
    }
    return directive;
});
```

See the `link` part in above code, it is calling the external expression that got assigned to method property of isolate scope and passing the parameter as you can see above. Here also the parameter is passed using object map. But here if we run the page, then this method will be called by default called as link will be executed while load.

### Third Option

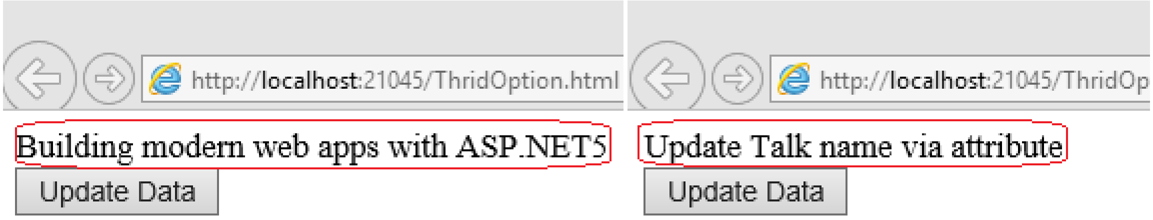
This is similar approach with above but we will further enhance it. As I discussed above that `attrs` contains all the attributes of the custom directive. So we can set some value in attribute that we can read later. Let's see with an example

```
<expcustomdirective expr="UpdateData(updatedName)"
talkname="{{talk.name}}" talknewname ="Update Talk name via attribute"/>
```

Here I added an attribute as `TalkNewName` that we will use and pass it in the expression. Let's see How?

```
myangularapp.directive('expcustomdirective', function () {
    var directive = {
        restrict: 'E', // restrict this directive to elements
        scope: { method: '&expr', talkname: '@' },
        template: "<div>{{talkname}}</div> <input type='button' " +
            "ng-click='method({updatedName : NewTalkName})' value='Update
Data'/> ",
        link: function (scope, element, attrs) {
            scope.NewTalkName =  attrs.talknewname;
        }
    }
    return directive;
});
```

In above code, if we see the link function then we find, we are creating a new property (named `NewTalkName`) in isolated scope and assign value from the attribute (See inside `link`). Now again, if we see `ng-click` in the code, I am mapping the argument name with new property added in the scope. Let's run it.


http://localhost:21045/ThridOption.html
http://localhost:21045/ThridOp

Building modern web apps with ASP.NET5

Update Talk name via attribute

*Note – AngularJS converts camel-case to snake-case when moving from JavaScript to HTML or vice-versa. Like if you added an attribute with name `talkNewName`, it will be accessed id JavaScript by `talknewname` only. I spent lots time to figure it out.*

## Conclusion

So in this post, we discussed different work around of passing values in the external expression. It can be helpful in many different scenarios. You yourself can explore some other options if it does not serve your need but there is no direct way in AngularJS. Above code may also not look very intuitive but it does solve the requirement.

# Day 12
# Exploring Filters

Today we are going to going another awesome feature – `Filter`. Displaying data and providing filters one of the common tasks of web applications. We have already discussed about populating data using ng-repeat, it's time to discuss Filters. We'll start from basics then dig deep and create our own custom filter at the end.

## What is Filter

`Filter` as the name suggests, filters the data that gets passed through it. In a simple sentence, it takes an array of items as input and filter it which results to another array of data with same or less number of items and in same or other transformed format.

## Filter in JavaScript

JavaScript also provides us capability to filter the data so let's understand that before moving to Angular Filter. If we want to filter the data what should be required.

1. An array of data
2. A comparator which return true of false. It takes each element if comparator returns true then add in resultant array else leave that element.

In JavaScript, comparator takes three input parameters – Value of the element, index of the element, the array being traversed and returns true or false. We can also pass some more parameters based on our requirement and have our own custom logic.

So let's have a look on JavaScript filter. In this example, I have just provided comparator which checks whether the passed value is even or odd. The syntax of JavaScript filter is as

```
arr.filter(callback[, thisArg])
```

Here callback takes three arguments.

1- value of the element
2- index of the element
3- Array object being traversed

## Example

Here I am providing a comparator which returns true in case of even numbers.

```
function IsEven(element, index, array) {
    // Not using input parameters like index and array
    return (element %2 == 0);
}

var filtered = [530, 3, 8, 112, 11, 240, 43].filter(IsEven);

document.write("Filtered Value : " + filtered);
```

The above code is self-explanatory. Here I created a comparator which returns true or false based on the condition and provided that in the filter. It returns – 530, 8,112,240.

*Note – This feature is of ECMA-262 standard and won't work which supports prior version.*

Now we have understood the basics of filter and similar feature available with JavaScript. Let's move to AngularJS.

## Filters in AngularJS

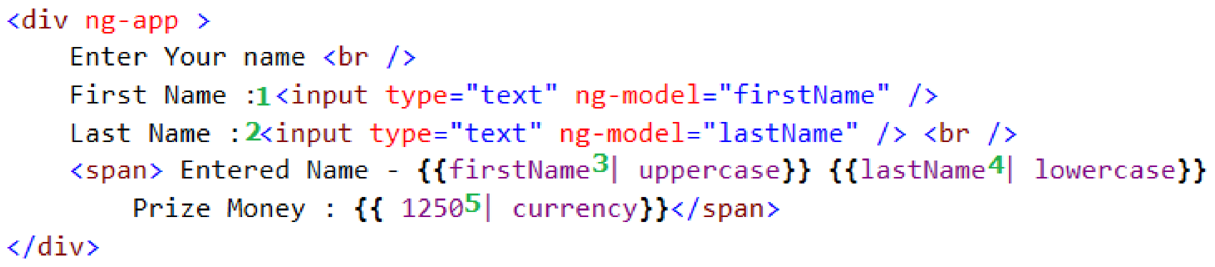
As AngularJS provides us more power to do thing with less and simpler code, the same applies here as well. Let's start exploring the filters and its uses.

Angular provides a very simple way to use a Filter. We need to provide a filter expression separated by pipe (|) and we can add many filters in one go, just by separating with pipe. Filters can be used as two ways in AngularJS

1- Take an item and transform on another format. Ex – uppercase, lowercase, currency, date.
2- Filters an array which results another array with same or transformed data as discussed earlier.

## Filter of type 1 (Single item)

We will see both in this post via an example and start from first one. Let's look at the code



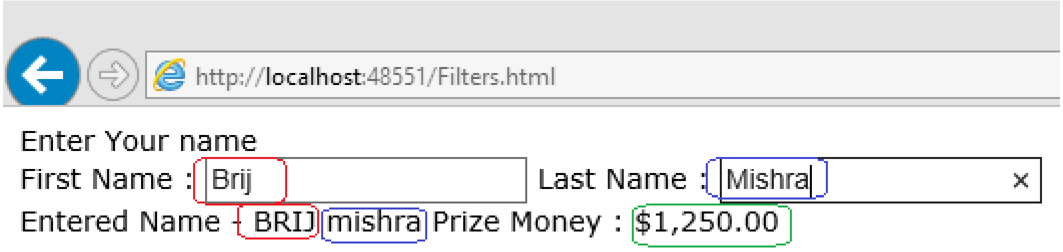
```
<div ng-app >
    Enter Your name <br />
    First Name :1<input type="text" ng-model="firstName" />
    Last Name :2<input type="text" ng-model="lastName" /> <br />
    <span> Entered Name - {{firstName3| uppercase}} {{lastName4| lowercase}}
        Prize Money : {{ 12505| currency}}</span>
</div>
```

I have put some numbers in green and let's discuss one by one.

1- An input box to enter first name with Angular model `firstName`.
2- An input box to enter first name with Angular model `lastName`.
3- Here `firstName` that is entered in textbox is displayed with `Filter` uppercase which converts the text in uppercase, no matter in which case it is entered by user.
4- Here `lastName` that is entered in textbox is displayed with `Filter` lowercase which converts the text in lowercase, no matter in which case it entered by user.

5- Prize money is hard-coded here and currency `Filter` is applied.

Now let's see it running



http://localhost:48551/Filters.html

Enter Your name
First Name : Brij Last Name : Mishra
Entered Name : BRIJ mishra Prize Money : $1,250.00

Here we can see that it is running as expected. First name is in upper case while last name is in lower case and Prize money transformed in currency format.

## Filter of type 2 (An array of items)-

Let's look the filter that applies while displaying list of data. So the code looks like

```
    <div ng-app>
        <div class="container"1ng-init="courses = [
{number : '1001' , name : 'ASPNET 5 from Scratch', trainer : 'Brij Bhushan Mishra'},
{number : '1002' , name : 'Node js', trainer : 'Dhananjay Kumar'},
{number : '1003' , name : 'ASPNET WebAPI', trainer : 'Suchit Khanna'},
{number : '1004' , name : 'WCF Exception handling', trainer : 'Anil Kumar'} ]">

            <div class="row">
                <table class="table table-condensed table-hover">
                    <tr>
                        <th>Course</th> <th>Name</th> <th>Trainer</th>
                    </tr>
                    <tr2ng-repeat="course in courses3| orderBy : 'name' : true">
                        <td> {{course.number}}</td>
                        <td> {{course.name}}</td>
                        <td> {{course.trainer}}</td>
                    </tr>
                </table>
            </div>
```

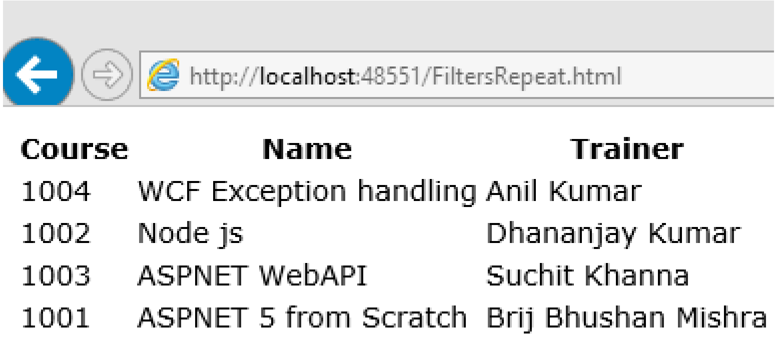
Here I have initialized some data then used that later for the demo. Let's discuss each number mentioned in above code as earlier.

1- Initializing an array named courses which a list of course via `ng-init`.
2- Using `ng-repeat` to display all the items in array in tabular format.
3- Applied an order by attribute on the property name. Here name is provided in single commas as a value because Angular search that string in the items in provided array. Also there is one more value provided *true,* what is the use of that. Let's see the syntax of this filter.

`{{ orderBy_expression | orderBy : expression : reverse}}`

So here the last item is *reverse* so if it *true* the items would be sorted in descending order else ascending order. And here we have set it is as true so it should be in descending order.

Now let's run that and see the output



http://localhost:48551/FiltersRepeat.html

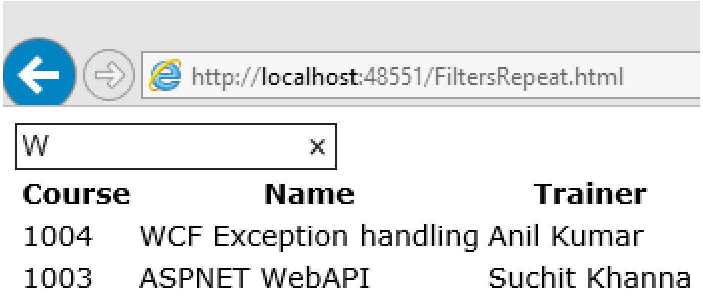
| Course | Name | Trainer |
|---|---|---|
| 1004 | WCF Exception handling | Anil Kumar |
| 1002 | Node js | Dhananjay Kumar |
| 1003 | ASPNET WebAPI | Suchit Khanna |
| 1001 | ASPNET 5 from Scratch | Brij Bhushan Mishra |

So here we can see the items is sorted in descending order based on the course name as expected. Let's see one more example

```
<input type="text"1ng-model="searchText"/>
<table class="table table-condensed table-hover">
    <tr>
        <th>Course</th> <th>Name</th> <th>Trainer</th>
    </tr>
    <tr ng-repeat="course in courses2|filter : searchText3| orderBy : 'name' : true">
        <td> {{course.number}}</td>
        <td> {{course.name}}</td>
        <td> {{course.trainer}}</td>
    </tr>
</table>
```

Here I added search input and applied filter using that

1- Textbox to enter the search text.
2- Here I added a filter (named as filter) on the entered value in text box with model name `searchText`.
3- One more filter `orderBy` added as earlier example. It means here we have applied two filters and that is totally correct. We can apply as many filter as we want by separating via pipe (`|`). let's run that



http://localhost:48551/FiltersRepeat.html

W ×

| Course | Name | Trainer |
|---|---|---|
| 1004 | WCF Exception handling | Anil Kumar |
| 1003 | ASPNET WebAPI | Suchit Khanna |

Here `W` is entered in textbox and two rows shown which contained this characters. And you can observe that ordering is also working based on the filters provided.

## Conclusion

Today we started discussing basics of Filters and plain JavaScript supports it. Then started discussing about the Filters available in JavaScript. We have seen that in Angular, Filters can be broadly divided in two types – First which applies on a single item and other which applies on array of items. We have seen that how easy to use and leverage the power of these features.

# Day 13

# Exploring Filters – Custom Filters

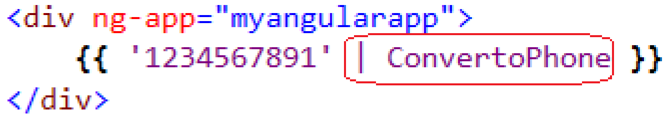
Customization, extendibility, openness sits in the core of Angular. Most of the feature provides us simple way to customize the feature and create our own based on the need. In Day 12, we discussed about the basics of Filter and created various examples. Today we will move further and discuss how to create custom Filters. Today we will be discussing about custom Filter, its uses and create a few custom ones. Also we'll try to explore some more flavors of the filters. We discussed two main types of Filters in previous post, one that transforms an item in another format. Another one that can be applied on array of items which produces another array after applying Filter. As we discussed in earlier days that Modules is Angular are like containers. So every custom items must be registered with it. Same is true here as well.

## Custom Filter of Type 1 (Single Item)

Let's start creating filter of first type, we already saw some Filters like uppercase, lowercase, currency etc. Now we are going to create a Filter which currently does not exist. It takes a string of numbers and convert into phone number format. Excited? Let's start

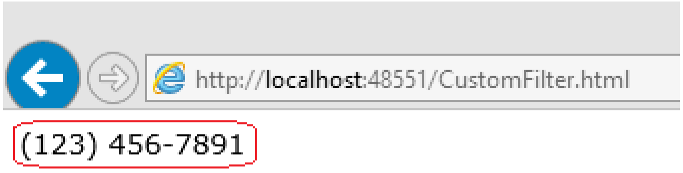
```
var myangularapp = angular.module('myangularapp', []);
    myangularapp.filter('ConvertoPhone', function () {
        return function (item) {
            var temp = ("" + item).replace(/\D/g, '');
            var temparr = temp.match(/^(\d{3})(\d{3})(\d{4})$/);
            return (!temparr) ? null : "(" + temparr[1] + ") " +
temparr[2] + "-" + temparr[3];
        };
    });
```

Here first we created an Angular module `myangularapp`. Then added a filter in that module – `ConverttoPhone` which takes an input parameter and converts that in Phone Number format. Inside the function, normal JavaScript code is written to transform the data. And we have added a custom filter with module. Now it's time to use it. Let's see the HTML first



```
<div ng-app="myangularapp">
    {{ '1234567891' | ConvertoPhone }}
</div>
```

Here we see that ConverttoPhone (red encircled) is used as similar to predefined filters and currently it is applied on a hard-coded value. Let's run that



(123) 456-7891

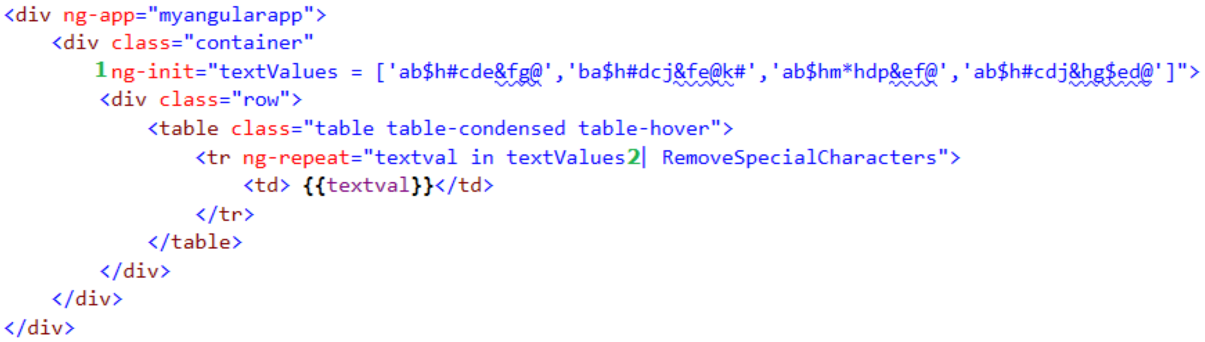
So it just converted the number in US phone number format as we wrote in JavaScript function.

## Custom Filter of Type 2 (Array of Items)

Not to forget that the above filter cannot be applied on arrays or list of data. So let's create the filter that applies on array of items and we will also try to find the reason that why the above filter cannot be applied. We are going to create a filter that takes a list of items and removes special character from each item. Let's see the JavaScript code

```
myangularapp.filter('RemoveSpecialCharacters', function () {
    return function (items) {
        var filtered = [];
        for (var i = 0; i < items.length; i++) {
            var item = items[i];
            filtered.push(item.replace(/[^\w\s]/gi, ''));
        }
        return filtered;
    };
});
```

Here if we see then the only difference from earlier one is that here the function takes in input, an array of items that internally iterated and each value is processed accordingly. And the filtered value is returned. Now let's see the HTML code

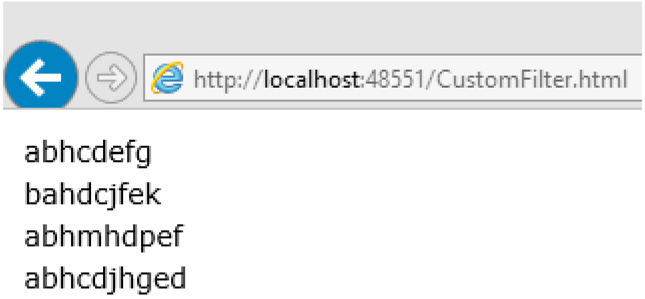


```
<div ng-app="myangularapp">
    <div class="container"
       1ng-init="textValues = ['ab$h#cde&fg@','ba$h#dcj&fe@k#','ab$hm*hdp&ef@','ab$h#cdj&hg$ed@']">
        <div class="row">
            <table class="table table-condensed table-hover">
                <tr ng-repeat="textval in textValues2| RemoveSpecialCharacters">
                    <td> {{textval}}</td>
                </tr>
            </table>
        </div>
    </div>
</div>
```

Refer the two numbers (1 and 2) above and let's see what is happening there?

1- We initialized an array textValues with a list of four values which contains special characters.
2- We applied the custom filter RemoveSpecialCharacters as Angular provided filter.

Now let's run the code

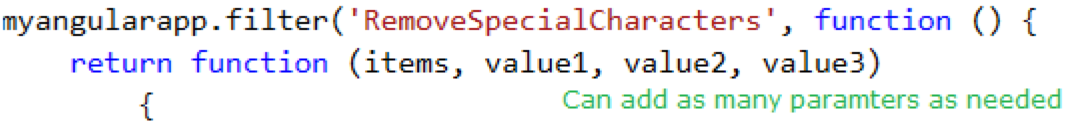


```
abhcdefg
bahdcjfek
abhmhdpef
abhcdjhged
```

Here we can see that the filtered items returned without any special character.

## Passing additional Values to Filter

We have seen in Angular filters that it provides the ability to provide some more values to a Filer separated by colon (:), how to do that in custom filter. It is very simple, have your custom filter created as



```
myangularapp.filter('RemoveSpecialCharacters', function () {
    return function (items, value1, value2, value3)
        {
```

So we can add as many parameters as we need and these can be passed via colon (:) similar to pre-defined filters.

## Conclusion

Today we continued with previous day and discuss further about the Angular Filter and the main focus was on Custom Filters. We created custom filters of both types and saw that they can be used similar to predefined Filters. We also saw that how we can pass additional parameters to the custom Filters.

# Day 14

# Using Filters in Controller, Factory, Service

We have discussed about Filters in detail. But you must have seen here and many other places that Filters are used in Views (HTML) but there could be scenarios where we require to use filters at some other places like Controller, Factory, and Services etc. Then how would we able to access it there. Fundamentally as we know every item is contained by Module so one can be used in another

## Accessing Filters in Controller

Filters can be accessed at other different components in Angular and we will first see it in Controller then at other places. There are two ways to achieve it. Let's discuss it in details.

### First Option

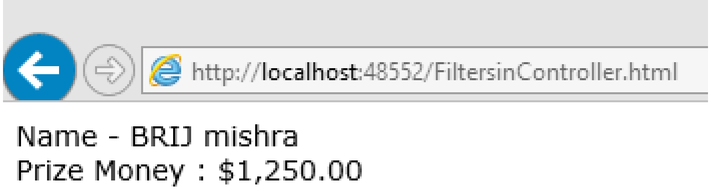
We can use a filter in a Controller by injecting $filter. The syntax looks like

```
function myController($scope, $filter)
{
...
$filter('<fliterName>')(<comma separated filter arguments>);
...
};
```

The above syntax is self-explanatory. Let's see some examples

```
myangularapp.controller("mycontroller", function ($scope, $filter) {
    $scope.filteredvalueUpperCase = $filter('uppercase')('brij');
    $scope.filteredvaluelowercase = $filter('lowercase')('Mishra');
    $scope.filteredvaluecurrency = $filter('currency')('1250');
});

// HTML
<div ng-app="myangularapp" ng-controller="mycontroller">
    <span>
        Name - {{filteredvalueUpperCase}} {{filteredvaluelowercase}} <br
/>
        Prize Money : {{ filteredvaluecurrency}}
    </span>
</div>
```

Above we can see that I have used three predefined filters (`uppercase`, `lowercase`, currency) and assigned the filtered values in new properties of `$scope` and later used it in view. Let's see it running



Name - BRIJ mishra
Prize Money : $1,250.00

The Filters worked as expected.

### Using Custom Filter

Similarly we can use custom filters in Controller as well. Let's see

```
myangularapp.filter('ConvertoPhone', function () {
    return function (item) {
        var temp = ("" + item).replace(/\D/g, '');
        var temparr = temp.match(/^(\d{3})(\d{3})(\d{4})$/);
        return (!temparr) ? null : "(" + temparr[1] + ") " + temparr[2] +
"-" + temparr[3];
    };
});

myangularapp.controller("mycontroller", function ($scope, $filter) {
    // Custom Filter
    $scope.filteredphonenovalue = $filter('ConvertoPhone')('1234567891');
});
```

In above examples, we have seen the Filter applied on single item. Similarly we can use the filters that can be applied on an array as

```
myangularapp.filter('RemoveSpecialCharacters', function () {
    return function (items) {
        var filtered = [];
        for (var i = 0; i < items.length; i++) {
            var item = items[i];
            filtered.push(item.replace(/[^\w\s]/gi, ''));
        }
        return filtered;
    };
});

myangularapp.controller("mycontroller", function ($scope, $filter) {
    var textValues = ['ab$h#cde&fg@', 'ba$h#dcj&fe@k#', 'ab$hm*hdp&ef@',
'ab$h#cdj&hg$ed@'];
    // Filter applied on array of items
    $scope.filteredtextValues =
$filter('RemoveSpecialCharacters')(textValues);
});
```

In the above example, I have applied the custom filter on array of items in controller. In the same way, pre-defined filter can be used.

## Using Filters in Factory, Service, Directives

As controllers are specific to views, on the other hand we write the service using `Factory` or `Service` are used globally. There could be some situations where we need to apply a Filter in any of the above. Similar as controller, as we inject the `$filter` in controller function, we can pass the same Factory/Service as well. Let's see it

### In Factory

```
// Injecting Filter in Factor
myangularapp.factory("myCustomService", function ($filter) {
    return {
        filteredData: $filter('uppercase')('brij'),
        filteredDataAnotherWay: $filter('lowercase')('Mishra')
    };
});
```

### In Service

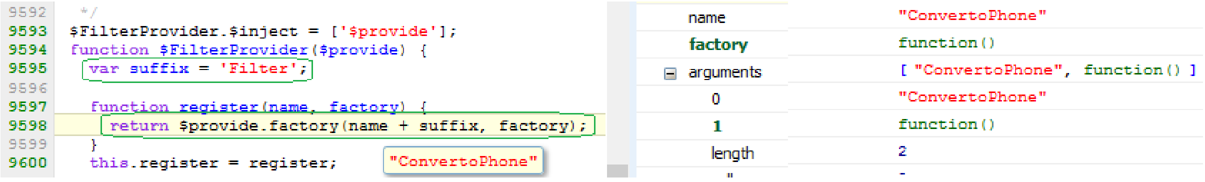
```
// Injecting Filter in Service
myangularapp.service('myCustomServiceV2', function ($filter) {
    this.filteredData = $filter('uppercase')('brij');
});
```

So we can see here that using the filter in other components is also same as controller.

### Second Option

In first way, we invoked the Filter in the said components using `$filter` and provided the filter name as parameter. Which internally invoked the appropriate filter. As we know that in JavaScript, everything is function, so there must be some function getting executed with the provided parameter when we provide a Filter.

To explain it in better way, I will take an example where I created a custom filter name `ConvertoPhone`. While bootstrapping the application, it must be storing the filter definition somewhere and then later must be used from there to invoke when we use (using $filter in above examples). Right!!  Let's see how it is stored



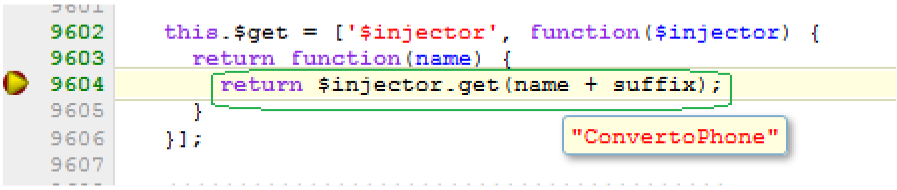
```
 */
$FilterProvider.$inject = ['$provide'];
function $FilterProvider($provide) {
  var suffix = 'Filter';

   function register(name, factory) {
     return $provide.factory(name + suffix, factory);
   }
   this.register = register;
```

| name | "ConvertoPhone" |
|---|---|
| factory | function() |
| ⊟ arguments | [ "ConvertoPhone", function() ] |
| 0 | "ConvertoPhone" |
| 1 | function() |
| length | 2 |

So here we see that a variable suffix with value `'Filter'`. That is used while registering the Filter with name as

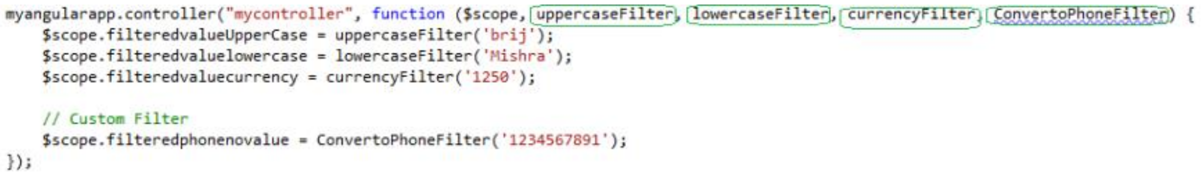
name + suffix => 'ConvertoPhone' + 'Filter' = ConvertoPhoneFilter

So here we can see the Filter is stored as `ConvertoPhoneFilter`. Now let's see that How it is invoked



```
    this.$get = ['$injector', function($injector) {
      return function(name) {
        return $injector.get(name + suffix);
      }
    }];
```

In above code, it is creating the name of the filter (‘`ConvertoPhone`’ + ‘`Filter`’ = `ConvertoPhoneFilter`) and fetching the function to execute. It means we can inject `ConvertoPhoneFilter` directly. The same is true for predefined filters as well. Above used code are part of Angular library. Let’s see the example



```
myangularapp.controller("mycontroller", function ($scope, uppercaseFilter, lowercaseFilter, currencyFilter, ConvertoPhoneFilter) {
    $scope.filteredvalueUpperCase = uppercaseFilter('brij');
    $scope.filteredvaluelowercase = lowercaseFilter('Mishra');
    $scope.filteredvaluecurrency = currencyFilter('1250');

    // Custom Filter
    $scope.filteredphonenovalue = ConvertoPhoneFilter('1234567891');
});
```

Here above we can see that instead of passing `$filter` in controller, we are passing all the Filters (predefined and custom) with the updated name that we discussed above. Here I passed three predefined and one custom filter and that we used as a normal JavaScript function. Similarly we can pass the Filters in `Factory`, `Service` etc.

## Conclusion

Today we saw that how can we use Filter at other places except Views. We discussed that there are two ways to achieve that and the discussed the first way for Controller and later for other components for both pre-defined and custom filters. Then we discussed about second option and saw couple of examples as well.

# Day 15

# Dependency Injection (DI)

Today we are going to discuss another very important topic - Dependency Injection (DI). It is a design pattern that got very popular in recent years. As now most of the projects following Agile Methodology and focused towards TDD model, DI plays a key role in that. Now every project try to leverage the benefits of this design pattern irrespective of technology and each framework/technology latest releases are developed keeping in mind Dependency Injection. AngularJS is developed keeping in mind the DI since its inception and the complete framework itself follows that which enables us to replace the existing modules by our own custom one by injecting it wherever required.

## What is Dependency Injection?

DI is software design pattern that implements the Inversion of Control (IoC) to resolve the dependencies. It allows us to write loosely coupled system which enables us write the reusable components and helps to test the code properly without affecting or need of any other component. Injection refers passing the dependent object to the component based on available options. Normally there are three standard ways to inject the dependency.

1. Method Injection (Passing the dependency as parameter)
2. Property Injection (Setter)
3. Constructor Injection

We will first understand DI as a concept and how can it be used in a normal JavaScript. Then we will see that how AngularJS help us to leverage it.

## Implementing DI in a Vanilla JavaScript

In this example, I have a folder watcher, which watches a folder and if there is some change takes place then it calls the event logger which internally calls a `MessageGenerator` which returns a message and logs it. First we will create this logger and later implement DI.

I have a Message Generator which has a method `GetMessage` that returns a message. It is as

```
function MessageGeneartor() {
}
```

```
MessageGeneartor.prototype.getMessage = function() {
    return 'My new custom message';
};
```

My *logmessage* method looks like which has the responsibility to get and log the message

```
function logMessage() {
    var messageCreator = new MessageGeneartor();
    var message = messageCreator.getMessage();
    alert(message + ' logged at ' + new Date());
}
```

*EventLogger* looks like

```
function EventLogger() {
}

EventLogger.prototype.logEvent = function () {
    logMessage();
};
```

Here from `logEvent` method, we are calling the `logMessage` which does the actual work. Last, my `FolderWatcher` looks like

```
function FolderWacther() {
    var myEventLogger = new EventLogger();
    myEventLogger.Logevent();
}

FolderWacther();
```

Here in `FolderWatcher`, we are creating the instance of `EventLogger` and calling. So lets have the complete example

```
function MessageGenerator() {
}

MessageGenerator.prototype.getMessage = function() {
    return 'My new custom message';
};

function logMessage() {
    var messageCreator = new MessageGenerator();
    var message = messageCreator.getMessage();
    alert(message + ' logged at ' + new Date());
}

function EventLogger() {
}

EventLogger.prototype.logEvent = function () {
    logMessage();
};

function FolderWacther() {
    var myEventLogger = new EventLogger();
    myEventLogger.logEvent();
}

FolderWacther();
```

## Using Dependency Injection

Here we are creating the all the dependencies inside the methods. We are going to refactor it and implement DI in two steps and start from lowest level.

### Implementing DI: Iteration 1

Our lowest level method is *logMessage* so we will start from here. In this method, instead of creating instance of *MessageGenerator*, we can pass it as parameter like

```
function logMessage(messageCreator) {
    var message = messageCreator.GetMessage();
    alert(message + ' logged at ' + new Date());
}
```

Inside the above method, now we are not creating any resource/dependency inside the method, instead passed as parameter. To accommodate it, `MessageGenerator` can be passed in `EventLogger` as

```
EventLogger.prototype.logEvent = function () {
    logmessage(new MessageGenerator());
};
```

### Implementing DI: Iteration 2

So in above example, we implemented DI at one level. Let's move further. As in above code, we are creating an instance of `MessageGeneraor` inside `logEvent` so let's now change `EventLogger` as

```
function EventLogger(logm, creator) {
    this.logger = logm;
    this.messageCreater = creator;
}

EventLogger.prototype.logEvent = function () {
    this.logger(this.messageCreater);
};
```

So here, we passed a reference of `logMessage` and instance of `MessageGenerator`. Consequently, we need to change the `FolderWatcher` as

```
function FolderWacther(myEventLogger) {
    myEventLogger.logEvent();
}

FolderWacther(new EventLogger(logMessage, new MessageGenerator()));
```

Now we have implemented DI in whole hierarchy. Now there is no method or class where we created any instance inside. Let's see the complete code now.

```
function MessageGenerator() {
}

MessageGenerator.prototype.getMessage = function() {
    return 'My new custom message';
};

function logMessage(messageCreator) {
    var message = messageCreator.getMessage();
    alert(message + ' logged at ' + new Date());
}
```

```
function EventLogger(logm, creator) {
    this.logger = logm;
    this.messageCreater = creator;
}

EventLogger.prototype.logEvent = function () {
    this.logger(this.messageCreater);
};

function FolderWacther(myEventLogger) {
    myEventLogger.logEvent();
}

FolderWacther(new EventLogger(logMessage, new MessageGenerator()));
```

This must have make some idea of DI. Till now we have seen that how can we write a vanilla JavaScript code using DI. Now let's jump to AngularJS.

## Dependency Injection in AngularJS

As AngularJS is a complete JavaScript framework which provides many out of the box features. And on top of it the whole framework is DI friendly and allows us to inject dependency in different ways. We will discuss it in detail.

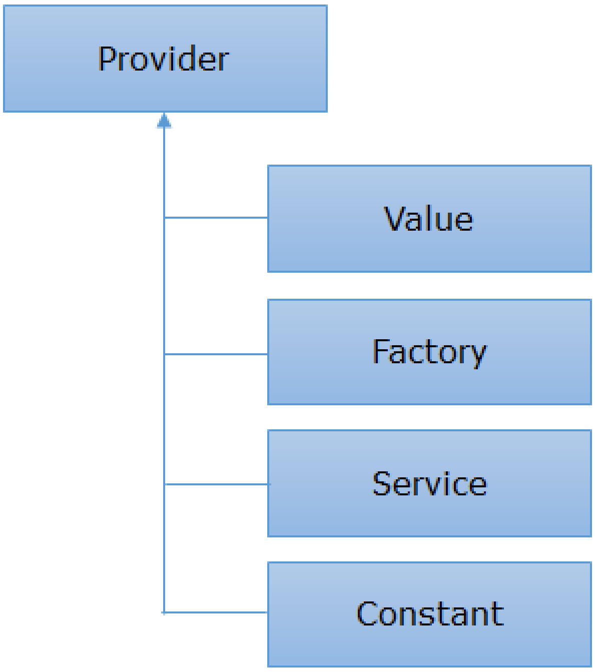
There are two key items when we implement dependency injection

1 Creating the instance. It could be created at while application initialization.
2 Passing the instance to the appropriate place where it can be used.

In Angular, `$injector` is key service which helps in creating the instances and passing it to the appropriate places. We have already seen in one of our earlier posts, the services are kind of global singleton objects which can be reused across the application when needed. `$injector` helps in creating the instances of **services** and **special objects**.

1 There are two types of services in Angular.
    a. One is inbuilt services like $http, $q etc (which starts from $)
    b. And another custom services that we create and register with module.
2 Special objects are like directives, filters, controllers etc.

`$injector` itself does not create instance but it take the help of other component which creates the instance as *$injector* needs. And this component is called *Provider*. Provider is a kind of base mechanism to create the instance but there are four other types which are just a syntactic sugar on top of it. Internally they all use the provider only. Let's see all those



Now let's discuss one by one

## Value

It is the simplest way to provide a JavaScript object that can be used in the entire angular application. The passed value could be any primitive or non-primitive type. It could be number, string, date array or any JavaScript object. As we discussed in many earlier posts, Module works as container in angular app so all the components must be registered with it. So same is true here as well. Let's see an Example

```
var myangularapp = angular.module('myangularapp', []);

// Providing a number and assigning to Value object
myangularapp.value("NumberofItems", 40);

// Providing a string and assigning to value object
myangularapp.value("ApplicationHeader", "AngularApp with DI using value");

// Providing a current date and assigning to value object
myangularapp.value("Person", { name: 'Tom', Age : 31, City : 'Florida' });

// Providing a current date and assigning to value object
myangularapp.value("Time", new Date());
```

Here we registered four objects of different type via value with module. Now these can be used in the entire application as

```
myangularapp.controller("mycontroller", function($scope, NumberofItems,
ApplicationHeader, Person, Time) {
    $scope.ApplicationHeader = ApplicationHeader;
    $scope.CurTime = Time;
    $scope.itemCount = NumberofItems;
    $scope.Person = Person;
});
```

HTML

<code>

Here we can see that we have injected all four in controller and later used that in the view. So it is the simplest one.

### Factory

Using Factory provider, we create a function which takes 0 to n parameters which can be dependent on others items which can be passed as parameters. Based on those parameters and via some business logic (if required), it creates an object and return it. One of the key points here, It is singleton and reusable component. We can use the object as parameter that we set as value above. Let's see an example

```
var myangularapp = angular.module('myangularapp', []);
// Providing a number and assigning to Value object
myangularapp.value("pi", 3.14);

myangularapp.factory('Circle', function(pi) {
    var myfactory = {};
    myfactory.Area = function (r) {
        return pi * r * r;
    }

    myfactory.Circuimference = function (r) {
        return 2* pi * r;
    }
    return myfactory;
});

myangularapp.controller("mycontroller", function ($scope, Circle) {
    $scope.AreaofCircle = Circle.Area(5);
    $scope.CircuimferenceOfCircle = Circle.Circuimference(5);

});
```

Here we created a service using factory which returns an object which has two methods – `Area` and `Circuimference`. Here I have used value of pi as parameter which is set via Value.

## Conclusion

Today we discussed about the basics of DI and how we can use in a plain JavaScript with an example in multiple steps. We discussed about the various opportunities available in AngularJS framework and it provides four options to inject the dependencies – Value, Factory, Service and Constant. Today we discussed Value and Factory in detail. Rest two will discuss tomorrow.

# Day 16

# Dependency Injection (DI) contd.

Last day we started discussing with Dependency Injection from basics and worked on an example in plain JavaScript. Then we discussed the way AngularJS provides us the opportunity to implement DI and various options available with it. We discussed two types *Value* and *Factory* in detail. Today we will discuss other types and try to discover its working behind the scene.

## Service

It provides another way to create a service and similar to factory. Only difference between factory and service is that service returns a new'ed instance by invoking constructor. Let's see the example that will make it clearer. Here the same example that we used in factory last day, is going to be used

```
var myangularapp = angular.module('myangularapp', []);

myangularapp.service('Circle', function() {

    var pi = 3.14;
    this.Area = function (r) {
        return pi * r * r;
    }
    this.Circuimference = function (r) {
        return 2* pi * r;
    }
});

myangularapp.controller("mycontroller", function ($scope, Circle) {
    $scope.AreaofCircle = Circle.Area(5);
    $scope.CircuimferenceOfCircle = Circle.Circuimference(5);

});
```

In above example, you can see that I have removed *myfactory* variable and all the APIs starts with this operator. The returned type is an object which has the APIs that we defined while creating Service.

## Provider

Provider is the key for all the other types as others are just syntactic sugar on it. A different wrapper has been put over that to get different flavor. Writing provider directly is not very intuitive and requires us to write bit more code but very helpful in understanding the concept and fits best in some scenarios. For creating a new Provider, there are two steps involved,

1. Defining a provider
2. Configuring the Provider.

Provider can be used when exposing API throughout the application. These APIs are configured before the application starts. It provides us ability to change the behavior of Provider a bit between multiple applications. Let's see an example

### Defining Provider

```
var myangularapp = angular.module('myangularapp', []);

// creating the provider
myangularapp.provider('RequestProcessor', function () {

    var requestId = '00000000';

    this.setrequestId = function (value) {
        requestId = value;
    };

    this.$get = function () {
        return "Processing Request Id : " + requestId;
    };
});
```

In the above provider, we have two items, one that takes the requestId and assigns it in local variable and another $get property, which is actually key here and returns the instance of the service. Let's see the configuration part

### Configuring Provider

```
// Configuring the provider
myangularapp.config(["RequestProcessorProvider", function
(RequestProcessorProvider) {
    RequestProcessorProvider.setrequestId("123456");
}]);
```

Here we see a Provider is injected in config method (`provider` is added as suffix) and the `setrequestid` is called. It is not the normal instance injector. When an Angular application starts, it configures and instantiates all the providers. And till this time application is not completely initialized, it is in configuration phase, so any type of services that we discussed earlier, will not be accessible here.

Now we can use it in our page as

```
myangularapp.controller("mycontroller", function ($scope,
RequestProcessor) {
    $scope.statusmessage = RequestProcessor;
});
```

The same provider can be used at another place with different behavior, if we put the provider in some common file and config part in the view (or in another JS file and include accordingly). Let's say in one view, I have the above config implementation and in other we have as

```
myangularapp.config(["RequestProcessorProvider", function
(RequestProcessorProvider) {
    RequestProcessorProvider.setrequestId("5678");
}]);
```

Here I am passing different Id. So the same provider can be used differently in two views. Complete example is available in sample source code.

## Constant

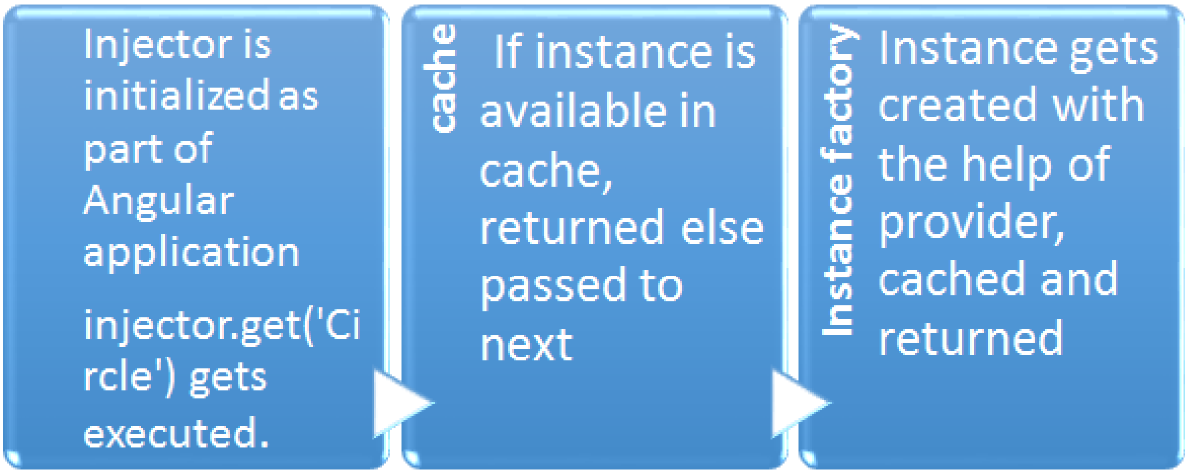
It is similar to Value but the difference lies where it is initialized. Value gets set during application configuration phase as discussed while constant gets set at earlier stage. It means we cannot add any dependency in provider which is initialized via Value or even factory/service while Constant one can be used. Let's see the example.

```
myangularapp.constant('applicationName', 'CodeWala request processor');
myangularapp.constant('applicationmetainfo', {
    version: '1.0'
});
```

We have discussed that how we can create multiple type of providers and then later use them at different places. We pass those services as parameter as when we access them they are properly initialized. But how does that happen?

## Dependency Injection behind the scene

We have talked briefly about the *$injector* last day and said that it is the core of dependency injection in AngularJS. In simple terms, whenever you see that a custom (that we discussed) or an inbuilt service (like *$http*, *$q* etc) are provided as parameter, `$injector` will be at work. It actually locate and find or create the instance and return it. Let's see the flow pictorially


Injector is initialized as part of Angular application
injector.get('Circle') gets executed.
cache
If instance is available in cache, returned else passed to next
Instance factory
Instance gets created with the help of provider, cached and returned

From the above image, we can understand that once the instance gets created, it gets cached and later whenever it is required returned from the cache itself. This is how all the services behaves as singleton as mentioned earlier. We have seen that how injector works behind the scene, let's see some example using it specifically. Earlier we passed the services as parameter, now we will get the instance using injector. Earlier we wrote our controller as

```
myangularapp.controller("mycontroller", function ($scope, Circle) {
    $scope.AreaofCircle = Circle.Area(5);
    $scope.CircuimferenceOfCircle = Circle.Circuimference(5);
});
```

We can also write it as

```
// Getting the injector
var injector = angular.injector(['myangularapp', 'ng']);

myangularapp.controller("mycontroller", function ($scope) {

    // Getting the instance via injetcor
    var Circle = injector.get('Circle');
    $scope.AreaofCircle = Circle.Area(5);
    $scope.CircuimferenceOfCircle = Circle.Circuimference(5);
});
```

Even we can invoke any function as

```
angular.injector(["myangularapp"]).invoke(function (Circle) {
    alert(Circle.Area(5));
});
```

Here we saw that instead of passing services as parameter, we got the instance using injector. And *invoke* allows us to execute that particular service. When we pass the services as a parameter, how does *injector* come into the picture? Actually, when we write as `ng-controller=”mycontroller”` then here the injector comes into the scene and it is the injector which actually resolves all the required dependencies by `injector.instantiate(‘mycontroller’).`

## Conclusion

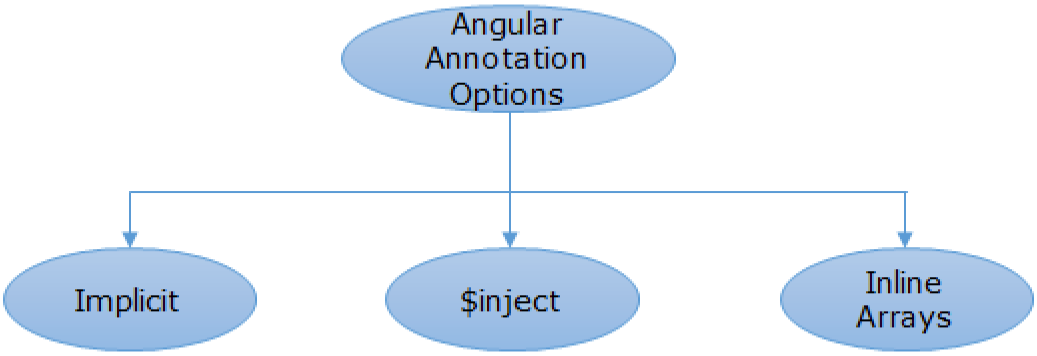
Today we continued our discussion with other options in AngularJS which includes Service, Provider and Constant. We saw the Provider is the root for all other options and it provides us more flexibility and can be used differently in different scenarios. In last section, we discussed that hoe DI works behind the scene and saw the role of $injector.

# Day 17

# Dependency Injection Annotation Process

Today again we are going to talk about Dependency Injection but today we will talk about another very important aspect that is Annotation Process. We have seen that we pass the dependencies as parameter whenever required and we had to always use the parameter name same as the dependencies were defined and this is crucial because Angular internally locate the services using the name itself. So if the name does not match then dependencies won't be resolved.

As the name should be same so minifying the script would break the application which cannot be accepted. So let's first see that the number of options provided by AngularJS.



So we have three ways to inject the dependencies. Let's discuss one by one

## Implicit parameter

This is the way that we have used in last two days, where we use the same name in parameter which we defined while creating. This is most vulnerable as discussed in the initial paragraph as well in case we minify our script files. As we used in one of our example

```
var myangularapp = angular.module('myangularapp', []);
myangularapp.service('Circle', function () {

    var pi = 3.14;
    this.Area = function (r) {
        return pi * r * r;
    }
    this.Circuimference = function (r) {
        return 2 * pi * r;
    }
});

myangularapp.controller(&quot;mycontroller&quot;, function ($scope,
Circle) {
    $scope.AreaofCircle = Circle.Area(5);
    $scope.CircuimferenceOfCircle = Circle.Circuimference(5);
})
```

Here we are using *Circle* as parameter as *Circle* was used while creating the service. As discussed last day, while resolving the dependencies *injector* comes into picture and as here there are two parameters passed *$scope* and *Circle* services, injector uses the names  and creates the instance using the provider.

This is simplest way to resolve the dependencies but it breaks if minifiers or obfuscators are used as we know these changes the name of parameters based on their algorithm which later does not match with the providers. But if you are sure that minifiers/obfuscators are not going to be used on production then you can happily use this.

## $inject property

This is another way to pass the dependencies which does not break in case of minification. In this case even if the name of the parameters gets changed still all the dependencies correctly get resolved. Let's see how

```
var myangularapp = angular.module('myangularapp', []);

myangularapp.service('Circle', function () {

    var pi = 3.14;
    this.Area = function (r) {
        return pi * r * r;
    }
    this.Circuimference = function (r) {
        return 2 * pi * r;
    }
});

var mycontroller = function ($scope, customCircle) {
    $scope.AreaofCircle = customCircle.Area(5);
    $scope.CircuimferenceOfCircle = customCircle.Circuimference(5);
};

mycontroller.$inject = ['$scope', 'Circle'];
myangularapp.controller('mycontroller', mycontroller);
```

Here we can see that we annotate *$inject* property of the function, here we provide the right name of the services in an array. In controller function we have chosen an arbitrary name (`customCircle` here) but code still runs fine.

As provider names are assigned as values in the array, no minifier or obfuscator changes the values so it works perfectly fine.

## Using Inline Array

This approach is preferable way to resolve the dependencies in the Angular. It is just another way of using array values to provide the service names to appropriate functions but much simpler than previous one. Let's take the similar example as above and implement it using inline array.

```
var myangularapp = angular.module('myangularapp', []);

myangularapp.service('Circle', function () {

    var pi = 3.14;
    this.Area = function (r) {
        return pi * r * r;
    }
    this.Circuimference = function (r) {
        return 2 * pi * r;
    }
});

myangularapp.controller(&quot;mycontroller&quot;, ['$scope','Circle',
function ($scope, customCircle) {
    $scope.AreaofCircle = customCircle.Area(5);
    $scope.CircuimferenceOfCircle = customCircle.Circuimference(5);
}]);
```

Here we can see that we create an inline array then we pass all the dependencies in the array and at end we pass the function where the dependencies need to be passed. Here function takes the input parameter which gets resolved to the dependencies based on the values passed in the inline array. We can pass all the services in similar way. This trick requires less code and easily understandable than the previous one.

Here the key thing to be noted that here order of the values in the array and parameter name is very important if that gets changed then the right dependencies won't be initialized to right parameters. Same holds true for second option as well.

### *Important Note*

As it is a common practice to use minifier while putting the code at production and somehow the dependency got resolved via first way then it could be disastrous for your application. In case of big application, where many developers working on same application everybody may use its own choice which could lead issues at production. So to avoid this, Angular provide us a way which makes sure that no body uses implicit injection and that is called ***Strict Dependency Injection***. For this we need add one directive *ng-strict-di* where we provide *ng-app* as

```
<div ng-app="myangularapp" ng-strict-di>
```

## Conclusion

Today we discussed about the various annotation process and saw that how the first process can break the application in case script minification. Then we discussed the other two ways which avoids the issue and found that last one is pretty easy, easy to understand and also the most preferable way to resolve the dependencies.

# Day 18

# Getting Started with Unit Test

Unit testing is one of the very important activities in software development. It helps in reducing the number of bugs in the code and maintaining the code quality. As now a days we work on small releases and keep adding/updating features, Unit Tests play vital role in maintaining the quality and helps in making sure that new changes does not break earlier functionality and reduces the testing effort. Unit Test becomes more important for languages like JavaScript because it is loosely typed and we don't find issues until we run the application. Also testing and debugging JavaScript is another time consuming activity.

## What is Unit Testing?

Unit Test is a snippet of code or function which tests a unit of code (function/API) and all the required dependencies are mocked. It means it just test the business logic written in the function and if any other dependent instance is required then mocked version is used. Unit test does not call any real service, database call etc.

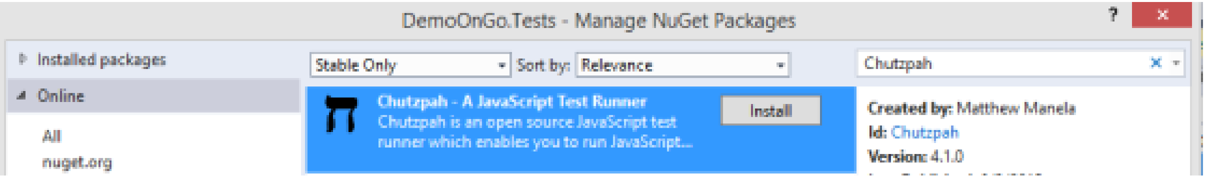
## What do we need in AngularJS to get started?

Being a .NET developer, I have written thousands of unit tests using C# and Visual Studio. You might be knowing that to create and run an unit test, we require unit test framework where we can run our unit test (like `NUnit`, `MSTest` etc), Unit test APIs, Mocking framework (like `NMock`, `Rhino mock`) and then these tests can be also part of Continuous Integration system where we can run the unit tests on every check-in to the repository. Similar infrastructure set up we also need in AngularJS. There are many options but I am going to use the most recommended one's. Let's discuss all

1. **`Jasmine`** - It is a behavior driven development framework for testing JavaScript code and preferred for Angular Applications. There are similar others like Qunit that we can use.
2. **`Angular mock`** - Angular provides its own mocking framework which helps in mocking the dependent objects.
3. **`Test Runner`**- One of the most used Test-runner is Karma but as we are fond of using Visual Studio, there is another nice test runner called Chutzpah which provides a plugin (`Chutzpah Test Adapter`) for Visual Studio which is very intuitive to use.

We have discussed the required tools. Now let's set up our environment as

1. Create an ASP.NET Project (I am starting with empty ASP.NET MVC Project with Unit Tests).
2. Install Chutzpah Test Adapter plugin via Nuget manager



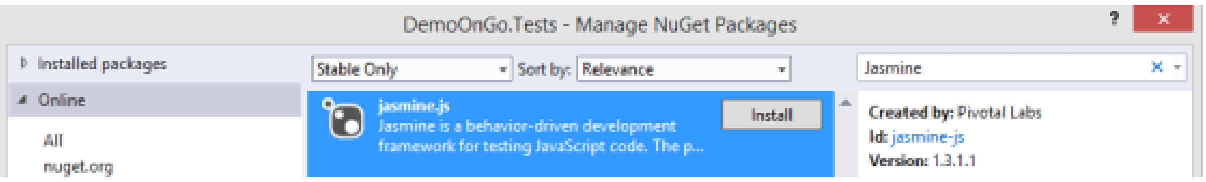
DemoOnGo.Tests - Manage NuGet Packages

Stable Only — Sort by: Relevance — Chutzpah

Chutzpah - A JavaScript Test Runner
Chutzpah is an open source JavaScript test runner which enables you to run JavaScript...

Created by: Matthew Manela
Id: Chutzpah
Version: 4.1.0

3. Install Jasmine Nuget package.



DemoOnGo.Tests - Manage NuGet Packages

Stable Only — Sort by: Relevance — Jasmine

jasmine.js
Jasmine is a behavior-driven development framework for testing JavaScript code. The p...

Created by: Pivotal Labs
Id: jasmine-js
Version: 1.3.1.1

4. Add Angular mock (angular-mocks.js) library for mocking purposes

Now we have set up our solution. First we will write a simple (addition) method and a unit test to verify the set up. Before writing test let's understand the following three items which are minimum to write any unit test using Jasmine.

1. **describe** - This is a global function that takes string and function as parameter which represents a suite of test. In another way, it provides us a way to group multiple tests.
2. **it** - This is another function which is written inside global function and takes two parameter as above string and function and this function is actual test.
3. **expect** - It takes the function that need to be tested as parameter and provide a list of matchers to match the result.

Let's write a JavaScript function and write unit test for that as

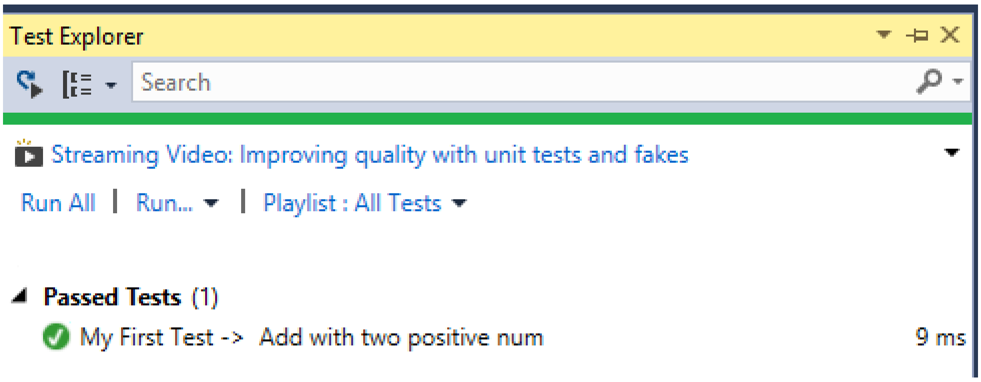
1. Add a folder in scripts folder (say CustomAngular) and add new JavaScript file say Home.js
2. Write a Add function in the Home.js as

```
function Add(firstnum, secondnum) {
    return firstnum + secondnum;
}
```

3. Add a file Home.test.js in scripts folder of Unit Test project to write our unit tests cases.
4. It's time to write our unit test for the same as

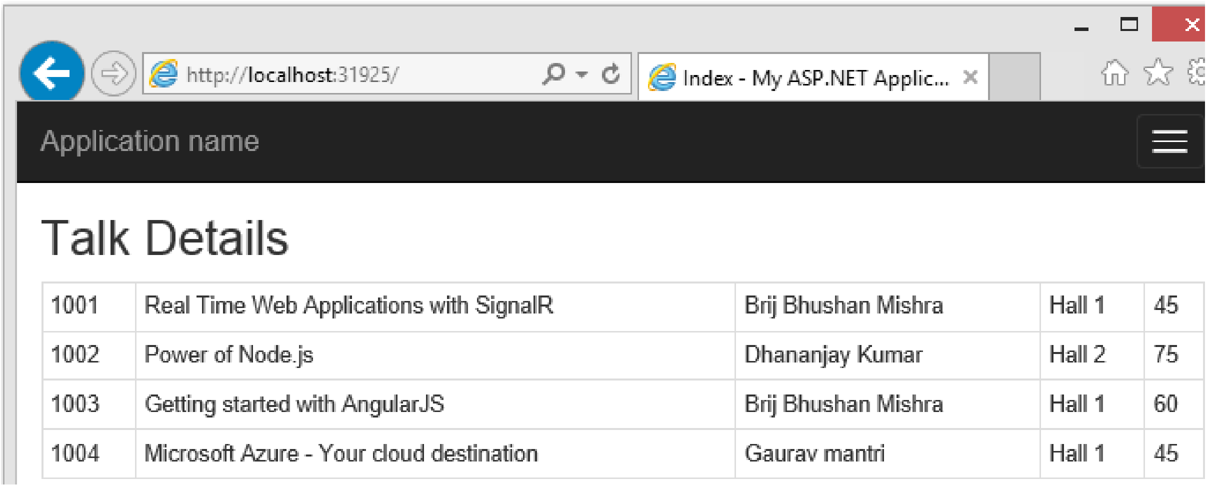
```
describe("My First Test -&amp;gt; ", function () {
    it("Add with two positive num", function() {
        expect(Add(2, 3)).toEqual(5);
    });
});
```

So we have written our first test. I already explained the special key words used here. You also need to include the references of Jasmine library and Home.js here. To run this Unit test, we just need to build the solution and open the Test Explorer and run the Unit Test. After running the Unit Test it will show green as


Test Explorer
Search
Streaming Video: Improving quality with unit tests and fakes
Run All | Run... | Playlist : All Tests

Passed Tests (1)
✓ My First Test -> Add with two positive num 9 ms

It means we have set up our infrastructure correctly. So let's move to real stuff. First we will create a MVC sample application then write unit test. This application would be similar to which we have created in our series of post. But we will take small steps to understand it better. Our MVC application looks as



## Talk Details

| | | | | |
|---|---|---|---|---|
| 1001 | Real Time Web Applications with SignalR | Brij Bhushan Mishra | Hall 1 | 45 |
| 1002 | Power of Node.js | Dhananjay Kumar | Hall 2 | 75 |
| 1003 | Getting started with AngularJS | Brij Bhushan Mishra | Hall 1 | 60 |
| 1004 | Microsoft Azure - Your cloud destination | Gaurav mantri | Hall 1 | 45 |

Let's see code quickly. Our MVC controller (`HomeController`) has just one *Index* method which returns a view. Our Index View is as

```
<h2>Talk Details</h2>
<div class="container">
<div class="row" ng-controller="talkController">
<table class="table table-bordered table-condensed table-hover">
<tr ng-repeat="talk in talks">
<td> {{talk.id}}</td>
<td> {{talk.name}}</td>
<td> {{talk.speaker}}</td>
<td> {{talk.venue}}</td>
<td> {{talk.duration}}</td>
 </tr>
 </table>
 </div>
 </div>
```

Now let's see our JavaScript file where we have put up our Angular code

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

homeModule.controller("talkController", ['$scope', function ($scope)
{

    $scope.talks =  [
        { id: '1001', name: 'Real Time Web Applications with
SignalR', speaker: 'Brij Bhushan Mishra', venue: 'Hall 1', duration:
'45' },
        { id: '1002', name: 'Power of Node.js', speaker: 'Dhananjay
Kumar', venue: 'Hall 2', duration: '75' },
        { id: '1003', name: 'Getting started with AngularJS',
speaker: 'Brij Bhushan Mishra', venue: 'Hall 1', duration: '60' },
        { id: '1004', name: 'Microsoft Azure - Your cloud
destination', speaker: 'Gaurav mantri', venue: 'Hall 1', duration:
'45' }
      ];

}]);
```

Also I have included Angular library, Home.js and added ng-app attribute as well. Now we will write the unit test for our Angular Controller.

First let's understand our Angular controller. Here we have a module and a controller which takes one parameter $scope. So while writing test, we require these three items and these need to be initialized first before running the test. To initialize, Jasmine provides us beforeEach function which can be used to initialize the items which runs before the test. And we need Angular mock library to set up all. Let's see the test and then discuss each

```
describe("Talk Controller Tests -> ", function () {
    var scope;
    var $ctrlCreator;

    beforeEach(module("homeModule"));
    beforeEach(inject(function ($controller, $rootScope) {
        $ctrlCreator = $controller;
        scope = $rootScope.$new();
    }));


    it("It should have four talks", function () {
        $ctrlCreator("talkController", { $scope: scope });

        expect(scope.talks.length).toBe(4);
    });

});
```

In the above code snippet, we first created two variables then initialized homeModule after that we *injected* controller and scope instance with the help of mock. In our Test, we are testing that length of talks returned by our controller is *four.* Now we can run our unit test via Test Explorer and it will pass. Similarly we can write many more tested for controller's functions.

## Conclusion

Today we talked about Unit test in angular. What are the basic things required to set up the project and get it started. Then we created a simple *function Add* and wrote unit test for that to check the setup. We continued writing the unit test of our controller and saw that how to initialized items before running the test. Hope you have enjoyed the day. Next day we will continue talking about Unit Testing and write unit tests for some other components.

# Day 19

# Writing Unit Test for Service, Custom Filter and Directives

We will continue our previous day discussion on Unit Testing in AngularJS where we talked about the basics of Unit Test in AngularJS, setting up the infrastructure and then wrote couple of unit tests. First we started with test for plain JavaScript method then we wrote an angular application and added unit test for Angular controller. While writing unit test, we also learned basics of Jasmine framework.

Today we will write unit tests for following components.

- Angular Service
- Custom Filters
- Custom Directives

## Testing your Service

As we know that service is an independent object which does some specific work and can be reused at multiple places. We normally used to have many services in our application which does various tasks. These services are the first candidates which should be considered for writing unit tests. Broadly in our service, we do two type of tasks, first where we take some input, write some logic and return the output accordingly. Second, where we connect some third party services via AJAX, process the response and return it. First type of service can be tested easily as normal JavaScript function. We are going to write Unit Test for second type.

We will extend our previous application where we hard coded the values in Angular Controller. Now instead, we will be creating an Angular service which will get the data from server and return that. Let's see our service

```
homeModule.factory("TalksService", function ($http, $q) {
    return {
        getTalks: function () {
            // Get the deferred object
            var deferred = $q.defer();
      // Initiates the AJAX call

            $http({ method: 'GET', url: '/home/GetTalkDetails'
}).success(deferred.resolve).error(deferred.reject);
            // Returns the promise - Contains result once request
completes
            return deferred.promise;
        }
    }
});
```

We have added our service using *Factory* which uses *$http* service to get the data from the server. One of the key points is that we are returning a *promise* here. Now let's make the required changes in the controller.

After these changes, our application would run same as we are now returning the same data server via Web API. Now it's time to write the Unit Test.

## Writing Unit Test

There are two new things here, usage of $http Service and returning *promise*. To test $http service, AngularJS provides a Fake implementation as *$*httpBackend which helps in mocking the service and setting up the response. To write the test, we need to initialize TalkService and httpBackend that we can inject at before each. Also we need to initialize homeModule as

```
var TalksServiceFactory, httpBackend;

beforeEach(module("homeModule"));

beforeEach(inject(function($httpBackend, TalksService) {
    httpBackend = $httpBackend;
    TalksServiceFactory = TalksService;
}));
```

Now unit test looks like

```
it("Should Return four Talks", function () {
    var talks;

        // Setting the mock up mock http response
        httpBackend
        .expect('GET', '/home/GetTalkDetails')
        .respond(200, [
            { id: '1001', name: 'Real Time Web Applications with
SignalR', speaker: 'Brij Bhushan Mishra', venue: 'Hall 1', duration:
'45' },
            { id: '1002', name: 'Power of Node.js', speaker:
'Dhananjay Kumar', venue: 'Hall 2', duration: '75' },
            { id: '1003', name: 'Getting started with AngularJS',
speaker: 'Brij Bhushan Mishra', venue: 'Hall 1', duration: '60' },
            { id: '1004', name: 'Microsoft Azure - Your cloud
destination', speaker: 'Gaurav mantri', venue: 'Hall 1', duration:
'45' }
        ]);

        // calling service
        TalksServiceFactory.getTalks().then(function (response) {
            talks = response;
        });

        // Flushing httpBackend
        httpBackend.flush();
```

```
            // verification
            expect(talks.length).toBe(4);
        });
```

Above code is self-explanatory. First we are initializing the mock service, calling the service and finally verifying the response. We can configure *httpBackend* for different scenarios based on usage of *$http* in actual service.

## Custom Filter

Filter is another one of the most used features of AngularJS. Here we are going to use one custom filter that we wrote in one of earlier days where we discussed about Custom Filters. So let's quickly see the Filter first

```
    homeModule.filter('ConvertoPhone', function () {
        return function (item) {
            var temp = ("" + item).replace(/\D/g, '');
            var temparr = temp.match(/^(\d{3})(\d{3})(\d{4})$/);
            return (!temparr) ? null : "(" + temparr[1] + ") " +
    temparr[2] + "-" + temparr[3];
        };
    });
```

Now to write unit test, we need to inject the *$filter* and then instantiate our custom filter in the init test. Let's see our unit test

```
    describe("Filter Tests ->;", function () {

        var filter;
        beforeEach(module('homeModule'));

        beforeEach(inject(function (_$filter_) {
            filter = _$filter_;
        }));

        it('if the number formatted', function () {
            var phoneFilter = filter('ConvertoPhone');

           expect(phoneFilter('1234567891')).toEqual('(123) 456-7891');
        });

    });
```

## Custom Directive

Directives are again one of the most important components for AngularJS. Writing Custom Directive is a complex task because it is not just another function which can be injected and called from anywhere. Custom Directives are declaratively used in HTML. As it directly changes the view and also designed in a way to be reused at different views, provided the scope is properly isolated based on requirement, these should be properly tested.

We are going to write two Custom Directives: First would be a simple one and another using isolate scope and we will write unit test for both the cases. First directive is an element directive which reads some information from scope and replaces the directive with the provide html in directive as

```
    homeModule.directive('myelementdirective', function () {
        var directive = {};
        directive.restrict = 'E'; //restrict this directive to elements
        directive.template = "Hello {{name}} !! Welcome to this Angular
    App";
        return directive;
    });
```

### Writing Unit Test

```
    var compileService, rootScope;
```

```
        beforeEach(module('homeModule'));

        // Store references to $compile and $rootScope so they can
        // be uses in all tests in this describe block
        beforeEach(inject(function (_$compile_, _$rootScope_) {
            compileService = _$compile_;
            rootScope = _$rootScope_;
            rootScope.talk = {
                name: 'abc', duration: '25m'
            };
            rootScope.name = 'Brij' ;
        }))

        it('My element Custom Directive defined', function () {

            var compiledDirective =
    compileService(angular.element('<myelementdirective/>'))(rootScope);

            rootScope.$digest();

            expect(compiledDirective).toBeDefined();

        });
```

Here we are compiling the directive and running the digest cycle and checking whether it is defines. Then we can write another test which checks whether the correct html is rendered or not as

```
    it('My element Custom Directive renders proper html', function () {

            var compiledDirective =
    compileService(angular.element('<myelementdirective/>'))(rootScope);

            rootScope.$digest();

            expect(compiledDirective.html()).toContain("Hello Brij !!
    Welcome to this Angular App");
        });
    Angular App");
        });
```

## Testing Custom Directive with isolated scope

Now we are going to write another directive with isolate scope. If you know or referred my previous post then we find that three types of isolated scope are available in Custom Directives which is also known as *Local scope properties*. We are going to write two way binding scope where the data is always in sync with parent regardless where it is getting changed. So let's see the custom directive first

```
    homeModule.directive('bindcustomdirective', function () {
        var directive = {
            restrict: 'E', // restrict this directive to elements
            scope: { talkinfo: '=' },
            template: "<input type='text' ng-model='talkinfo.name'/>" +
                "


    <div>{{talkinfo.name}} : {{talkinfo.duration}}</div>",
        };
        return directive;
    });
```

Here *talkinfo* gets initialize with the scope passed via an attribute while using Directive as

```
    <bindcustomdirective talkdetails="talk" />
```

As in the template, we have input which allows to change the scope object, this reflects in the parent scope as well.

## Writing Unit Test

To write the unit test, most of things would be same as above like initialization of compiler service, scope and assign some initial value to talk object in parent scope. So let's move to the test itself

```
it('Bind Custom Directive defined', function () {

    var compiledDirective = compileService(angular.element('
<bindcustomdirective talkinfo="talk" />'))(rootScope);

    rootScope.$digest();

    var isolatedScope = compiledDirective.isolateScope();

    expect(isolatedScope.talkinfo).toBeDefined();
});
```

Here we got the compiled directive using compiler service and run the digest cycle same as earlier one. One extra line added to get the isolate scope from the compiled directive and checking whether `talkInfo` is defined.

We will write another test and here we will check that if we change the isolated object's property whether that get reflected in parent scope or not as

```
it('Bind Custom Directive two way binding check', function () {
    var compiledDirective = compileService(angular.element('
<bindcustomdirective talkinfo="talk" />'))(rootScope);

    rootScope.$digest();

    compiledDirective.isolateScope().talkinfo.name = "Building
modern web apps with ASP.NET2";

    expect(rootScope.talk.name).toEqual("Building modern web apps
with ASP.NET2");
});
```

## Conclusion

We have written the unit tests few very important components of AngularJS. Although many more test could be written and even these components vary based on requirement, accordingly different unit test may be required. But today, I tried to provide the basics of writing unit test. Now you can add more test based on the requirement.