Angular JS 1.x Labs

Description: This document contains labs on Angular JS, to enable the learner to have a hands-on experience and understanding of Angular JS.

Document Control

Version	1
Created by	Karthikeyan. J
Reviewed by	
Date	Jan 2017

Table of Contents

1. Pre-Requisites

- Knowledge of JavaScript and jQuery

2. Hardware-Software Requirements

- The learner will require a computer (PC / Laptop) running Windows 7 and above with at least 4 GB of RAM. Additionally, the following software (free for training and learning purpose) is also required:-
- Visual Studio 2012/2013/2015 Community Edition
- SQL Server 2008/2012 with Management Studio
- Angular JS Library

In order to view videos mentioned in this GLP the learner needs submit a request at this link to get Pluralsight video access.

3. Introduction

The purpose of this document is to provide practice labs for JavaScript developers to learn about how to develop Single Page Applications (SPAs) using Angular JS framework & the Model-View-Controller (MVC) architecture.

4. Angular JS Labs

The following points need to be taken care while developing the Angular JS lab projects:

a. Use of proper JavaScript standards and naming conventions.
b. All script files need to be stored inside a folder named **scripts** in the web application.
c. All images need to be stored inside a folder named **images** in the web application.
d. Submission of labs will include project structure and the word document with the work flow diagrams, codes, and output screenshots.

Lab 1: One-Way Data-Binding in Angular JS Duration: 0.5 Hours

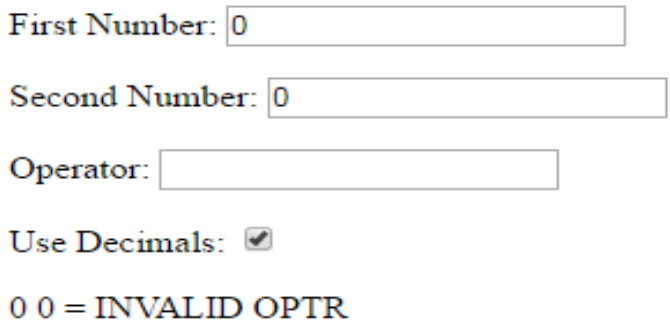
In this lab, the learners will implement simple one-way data-binding using Angular JS interpolation syntax.

1. Create an Angular JS controller named **MessageController.**
2. Create a property named **Message** in the scope of the controller. This property must store some arbitrary string.
3. Bind this property to an **<h2>** element in the view so that the message is displayed.

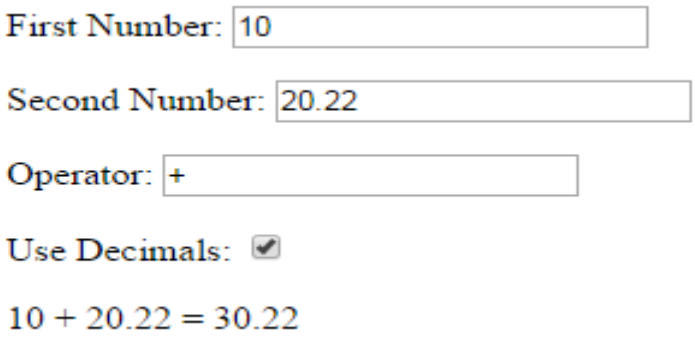
Lab 2: Two-Way Data-Binding in Angular JS Duration: 0.5 Hours

In this lab, the learners will implement two-way data-binding. The binding will be set up using the **ng-model** directive.

1. Create an Angular JS controller named **SimpleCalculatorController.**
2. Create the following properties inside the scope of the controller:
   FirstNumber, SecondNumber, Operator, Result, UseDecimals (Boolean property)
3. Create a function named **Calculate()** inside the scope. The method must determine the operator and store the result of the calculation inside the **Result** property.
4. Depending on the **UseDecimals** property, the **Calculate()** method must either store the result in decimal or non-decimal format.
5. Create a view as shown below:



The result must be displayed as the user types the first, second or the operator in the textboxes. Also, if the check box is checked, the result must be displayed in decimals. When an invalid operator is entered, the view must display an error message as shown above. Following is the view when the input is entered correctly:



Lab 3: Using Angular JS in-built directives Duration: 1 Hour

In this lab, the learners will use Angular JS in-built directives to hide and show elements, to apply css classes, to create partial views and to iterate through a model collection.

ng-show & ng-hide directives

1. Create a view as shown below:

When the **Show** checkbox is checked/unchecked, the heading **Hello** must be shown and hidden respectively.

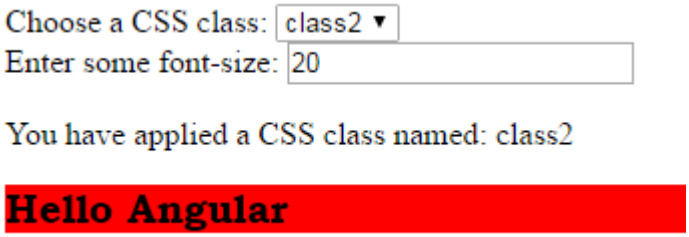
When the **Hide** checkbox is checked/unchecked, the heading **Welcome** must be hidden and shown respectively.

ng-class directive

1. Create two CSS classes as shown below:

```
<style>
    .class1
    {
        color:red;
        font-weight:bold;
        font-style:italic;
    }
    .class2
    {
        background-color:red;
        font-family:'Bookman Old Style';
    }
</style>
```

2. Create two CSS classes as shown below:



The dropdown must display the names of the CSS classes. Whenever the user selects a CSS classname from the dropdown, the selected CSS class must be applied to the text **Hello Angular.** Also, when the user changes the font-size in the textbox, the text's font size must change. The view must also display the name of the currently selected CSS class as shown in the snapshot above.

ng-repeat directive

1. Create an Angular JS controller which exposes the following model via its scope:

```
$scope.todos =
    [
        {
            action: "Get groceries", complete: false,
            assignedto: ["Martin", "Steve"]
        },
        {
            action: "Call plumber", complete: false,
            assignedto: ["Joe", "Steve"]
        },
        {
            action: "Buy running shoes", complete: true,
            assignedto: ["Martin", "Bruce"]
        },
        {
            action: "Buy flowers", complete: false,
            assignedto: ["John", "Bruce"]
        },
        {
            action: "Call family", complete: false,
            assignedto: ["Martin", "Joe"]
        }
    ];
});
```

2. The view must be as follows:

ToDo List

Action	Done?	Assigned To
Get groceries	☐	1. Martin 2. Steve
Call plumber	☐	1. Joe 2. Steve
Buy running shoes	☑	1. Martin 2. Bruce
Buy flowers	☐	1. John 2. Bruce
Call family	☐	1. Martin 2. Joe

3. Modify the view as shown below:

ToDo List

ToDo Number	Action	Done?	Assigned To
FIRST	Get groceries	☐	1. Steve 2. Richard
2	Call plumber	☐	1. Martin 2. Joe
3	Buy running shoes	☑	1. Joe 2. James
4	Buy flowers	☐	1. Richard 2. James
LAST	Call family	☐	1. James 2. Joe

Make use of pre-defined variables provided by the **ng-repeat** directive so that all the even rows are highlighted in a different color as shown in the snapshot above.

ng-options directive

1. Create an Angular JS controller which exposes the following model via its scope:

```
//array of complex type
$scope.employeelist =
    [
        { empid: 1, name: "Richard",place:"Manchester" },
        { empid: 2, name: "Joe", place: "Bristol" },
        { empid: 3, name: "Vincent",place:"California" },
        { empid: 4, name: "Graham",place:"London" },
        { empid: 5, name: "Chris",place:"Auckland" }
    ];
```

2. Create a view which displays all employee ids in a drop-down list.

3. When the user selects an employee id, the view must display the output as shown below:

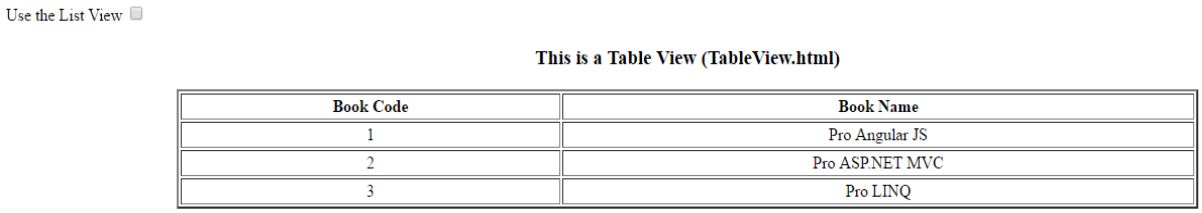
ng-include directive

1. Create an Angular JS controller which exposes the following model via its scope:

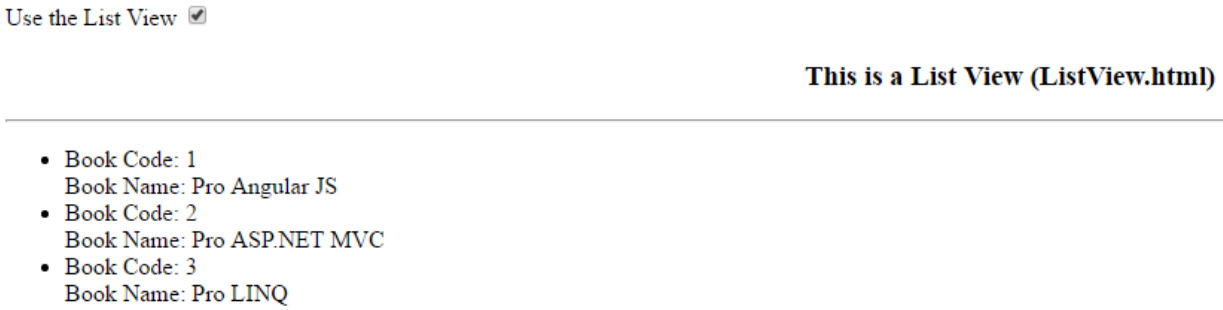
```
$scope.books =
    [
        { bookid: 1, bookname: "Pro Angular JS" },
        { bookid: 2, bookname: "Pro ASP.NET MVC" },
        { bookid: 3, bookname: "Pro LINQ" },
    ];
```

2. Create two HTML files named **TableView.html** & **ListView.html.**

3. Create a view as shown below:



4. When the checkbox **UseListView** is checked, the view named **ListView.html** must be applied as shown below:



5. When the checkbox **UseListView** is unchecked, the view named **TableView.html** must be applied as shown in the first snapshot above:
   Note that this functionality must be implemented using the *ng-include* directive.

Lab 4: Using Angular JS in-built filters Duration: 1 Hour

In this lab, the learners will use Angular JS in-built filters to format the model, data before it is displayed in the view.

currency, number, date, uppercase, lowercase filters

1. Create a controller which exposes the following model via its scope:

```
$scope.products = [
                    { name: "Apples", category: "Fruit", price: 1.20, expiry: 10 },
                    { name: "Bananas", category: "Fruit", price: 2.42, expiry: 7 },
                    { name: "Pears", category: "Fruit", price: 1002.02, expiry: 6 },
                    { name: "Tuna", category: "Fish", price: 20.45, expiry: 3 },
                    { name: "Salmon", category: "Fish", price: 1700.93, expiry: 2 },
                    { name: "Trout", category: "Fish", price: 12345.93, expiry: 4 },
                    { name: "Beer", category: "Drinks", price: 2.99, expiry: 365 },
                    { name: "Wine", category: "Drinks", price: 8.99, expiry: 365 },
                    { name: "Whiskey", category: "Drinks", price: 45.99,expiry: 365 }
];
```

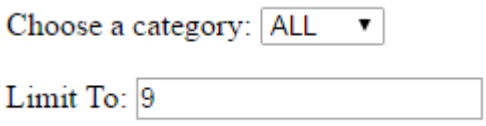
2. Create a view which uses the in-built Angular JS filters as shown below:

Product Name	Category	Price	Expiry Date	PRICE(currency)	PRICE(number)	EXPIRY(date)	NAME-CATEGORY(uppercase-lowercase)
Apples	Fruit	1.2	10	$1.20	1.20000	15 Jun 2016	APPLES, fruit
Bananas	Fruit	2.42	7	$2.42	2.42000	12 Jun 2016	BANANAS, fruit
Pears	Fruit	1002.02	6	$1,002.02	1,002.02000	11 Jun 2016	PEARS, fruit
Tuna	Fish	20.45	3	$20.45	20.45000	08 Jun 2016	TUNA, fish
Salmon	Fish	1700.93	2	$1,700.93	1,700.93000	07 Jun 2016	SALMON, fish
Trout	Fish	12345.93	4	$12,345.93	12,345.93000	09 Jun 2016	TROUT, fish
Beer	Drinks	2.99	365	$2.99	2.99000	05 Jun 2017	BEER, drinks
Wine	Drinks	8.99	365	$8.99	8.99000	05 Jun 2017	WINE, drinks
Whiskey	Drinks	45.99	365	$45.99	45.99000	05 Jun 2017	WHISKEY, drinks

Notice that the *date* filter must display the date based on the value of the *expiry* column. In other words, if the *expiry* column's value is *3,* then the *date* filter must display the date *3 days ahead of the current date*.

limitTo & filter filters

1. Modify the view to include a dropdown & a textbox as shown below:



2. The dropdown must display all the categories. When the user selects a category, the products for the selected category must be displayed as shown below:

Product Name	Category	Price	Expiry Date	PRI
Apples	Fruit	1.2	10	$1.20
Bananas	Fruit	2.42	7	$2.42
Pears	Fruit	1002.02	6	$1,002.

Choose a category: Fruit ▼

Limit To: 9

When the user selects *All* option from the dropdown, all the products must be shown.

3. Using the **limitTo** filter, display only those number of records depending on the value entered in the textbox as shown below:

Product Name	Category	Price	Expiry Date	
Apples	Fruit	1.2	10	$1.
Bananas	Fruit	2.42	7	$2.
Pears	Fruit	1002.02	6	$1,
Tuna	Fish	20.45	3	$20
Salmon	Fish	1700.93	2	$1,

Choose a category: ALL ▼

Limit To: 5

Lab 5: Custom Filters Duration: 1 Hour

In this lab, the learners will create a custom filter for displaying specialized images for Boolean values and to skip a specified number of records from a model collection.

Custom Filter Lab 1

1. Create a custom filter named **CheckMarkFilter** which displays a check mark when applied to a boolean value as shown below:

True with filter applied: ✓

False with filter applied: ✗

"Hello" with filter applied: checkmark filter error: Input must be a boolean

If the filter is applied to a non-boolean value, an error message must be displayed as shown in the snapshot above

Custom Filter Lab 2

1. Create a controller which exposes the following model data via its scope:

```
$scope.Products =
    [
        { name: "Apples", category: "Fruit", price: 1.20, expiry: 10 },
        { name: "Bananas", category: "Fruit", price: 2.42, expiry: 7 },
        { name: "Pears", category: "Fruit", price: 2.02, expiry: 6 },
        { name: "Tuna", category: "Fish", price: 20.45, expiry: 3 },
        { name: "Salmon", category: "Fish", price: 17.93, expiry: 2 },
        { name: "Trout", category: "Fish", price: 12.93, expiry: 4 },
        { name: "Beer", category: "Drinks", price: 2.99, expiry: 365 },
        { name: "Wine", category: "Drinks", price: 8.99, expiry: 365 },
        { name: "Whiskey", category: "Drinks", price: 45.99, expiry: 365 }
    ];
```

2. Create a view which displays this model as shown below:

Product Name	Category	Price	Expiry
Apples	Fruit	1.2	10
Bananas	Fruit	2.42	7
Pears	Fruit	2.02	6
Tuna	Fish	20.45	3
Salmon	Fish	17.93	2
Trout	Fish	12.93	4
Beer	Drinks	2.99	365
Wine	Drinks	8.99	365
Whiskey	Drinks	45.99	365

3. Create a filter named **SkipFilter** which when applied, skips the specified number of records from the start of the list to which it is applied. The user must be able to enter the number of records to be skipped via a textbox in the view as shown below:

Product Name	Category	Price	Expiry
Pears	Fruit	2.02	6
Tuna	Fish	20.45	3
Salmon	Fish	17.93	2
Trout	Fish	12.93	4
Beer	Drinks	2.99	365
Wine	Drinks	8.99	365
Whiskey	Drinks	45.99	365

Skip: 2 product(s)

Lab 6: Custom Directives

Duration: 1.5 Hours

In this lab, the learners will create a custom directive to display a user info card. The directive will contain custom attributes which will allow the directive to display an info card for any number of users. The directive will take input from the controller's scope to display an info card.

Custom Directives Lab 1

1. Create a controller which exposes the following model data via its scope:

```
$scope.User1 = {
    userid:1,
    username: 'Graham',
    address:
        {
            city: 'London',
            state: 'United Kingdom',
            pincode: 1111
        },
    friends:['Allan','Steve']
};
```

2. Create a custom directive named **UserInfoCard** which can be used as a **custom element** only.
3. When used in an HTML page, it must display the information about the model as shown below:

Custom Directives Lab 2 (Usage of a controller inside a custom directive)

1. Modify the custom directive's template to display buttons as shown below:

2. When the buttons are clicked, call methods named **ShowUserd() & HideUserId()** respectively which will show and hide the user id.
   The key point in the lab is that the methods must be placed inside a controller within the directive and not the module.

Custom Directives Lab 3 (Using *isolated scope*)

1. Modify the controller as follows to include the information about another user as follows:

```
$scope.User2 = {
    userid: 2,
    username: 'Chris',
    address:
        {
            city: 'Bristol',
            state: 'United Kingdom',
            pincode: 2222
        },
    friends: ['Graham', 'Tom']
};
```

2. Create a property named **datasource** within the scope of the controller.

3. The directive must be then usable from the HTML page as follows:

```
<userinfocard ds="User1"></userinfocard>

<userinfocard ds="User2"></userinfocard>
```

4. Doing so, must produce the following output:

Lab 7: Routing

Duration: 1 Hour

In this lab, the learners will mimic a simple inbox page. When the user clicks on an email subject, the related email will be displayed on the same page. The learners will also use routing events to execute code before and after the route navigation respectively.

1. Create an Angular JS controller named **MessagesController** which exposes the following model data via its scope as shown below:

```
$scope.AllMessages =
    [
        { MessageId: 1, From: "user1@xyz.com", To: "user1.@xyz.com", Subject: "Message 1",
            Body: "This is the first message" },
        {
            MessageId: 2, From: "user2@xyz.com", To: "user2.@xyz.com", Subject: "Message 2",
            Body: "This is the second message"
        },
        {
            MessageId: 3, From: "user3@xyz.com", To: "user3.@xyz.com", Subject: "Message 3",
            Body: "This is the third message"
        },
        {
            MessageId: 4, From: "user4@xyz.com", To: "user4.@xyz.com", Subject: "Message 4",
            Body: "This is the fourth message"
        },
        {
            MessageId: 5, From: "user5@xyz.com", To: "user5.@xyz.com", Subject: "Message 5",
            Body: "This is the fifth message"
        }
    ];
```

2. Create a view which displays the model as shown below:

This is the main Inbox Page

From	To	Subject
user1@xyz.com	user1.@xyz.com	Message 1
user2@xyz.com	user2.@xyz.com	Message 2
user3@xyz.com	user3.@xyz.com	Message 3
user4@xyz.com	user4.@xyz.com	Message 4
user5@xyz.com	user5.@xyz.com	Message 5

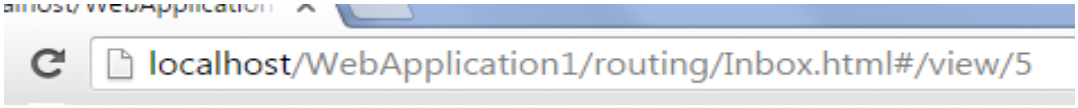
3. When a message link in the **Subject** column is clicked, the corresponding message must be shown on the same page as shown below:

This is the main Inbox Page

From	To	Subject
user1@xyz.com	user1.@xyz.com	Message 1
user2@xyz.com	user2.@xyz.com	Message 2
user3@xyz.com	user3.@xyz.com	Message 3
user4@xyz.com	user4.@xyz.com	Message 4
user5@xyz.com	user5.@xyz.com	Message 5

This is the fifth message

4. In such a case, the URL of the browser must display the following:



Pass the message id to each route when a message is clicked.

5. When the user clicks on a particular message, before the message is displayed ask a confirmation from the user as to whether the user wants to see the message or not.
Hint: use routing events provided by Angular JS

Lab 8: Custom Services Duration: 1.5 Hours

In this lab, the learners will create custom Angular JS service using the factory, service and provider patterns.

Custom Services Lab 1 (Using *factory()* method)

1. Create a service named **MathService** using the ***factory*** method.
2. Create two methods named **Square()** & **Cube()** respectively inside the service.
3. Each of these methods must accept a number as an argument & return the square & cube of the number respectively.
4. Create a controller named **MathController** & inject the service into it. The controller must expose a property named **Number** via its scope. Also create two methods named **CalculateSquare()** & **CalculateCube()** respectively inside the controller's scope respectively. These methods must call the **Square()** & **Cube()** methods of the service created above.
5. Create a property named **Result** in the scope of the controller which will hold the result of the calculation.
6. Create a view which inputs a number in a textbox. Create two buttons named **Calculate Square** & **Calculate Cube** respectively.
7. When the buttons are clicked, invoke the **CalculateSquare()** & **CalculateCube()** methods of the controller respectively.
8. The result must be displayed inside a **span** element within the view. Bind this **span** element to the **Result** property of the controller's scope.

Custom Services Lab 2 (Using *service()* method)

1. Implement the previous lab using the **service()** method.

Custom Services Lab 3 (Using *provider()* method)

1. Create a provider named **LogServiceProvider**.

2. Create a method named **setLogFlag()** in the provider which takes a parameter named **flag** as argument & sets its value into a private member named **logflag** within the service provider.
3. From the **$get()** method return a service instance which contains a method named **Log().**

4. This method examines the value of the private member named **logflag** (created above) & if the value is **console,** it must output a message **HELLO ANGULAR** to the browser's console. If the value is **screen,** it display a message **HELLO ANGULAR** in an alert box.

5. Configure the provider in the **config()** function of the module & set the value of **logflag** to **console** or **screen** by calling the **setLogFlag()** method of the provider.

6. Create two controllers named **Controller1** & **Controller2** in the module.

7. Inject the **LogService** into both the controllers.

8. Add a method named **Log()** in each controller which in turn invokes the **Log()** method of the service.

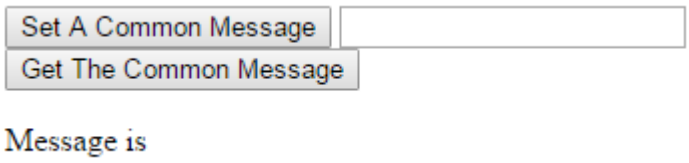
9. Create two **div** elements in the view which use each controller respectively & on the click of a button invoke the **Log()** method of the controller.

Depending on the value of *logflag* variable of the *LogServiceProvider,* the message *HELLO ANGULAR* must either be displayed on the browser's console or in an alert box.

Lab 9: Sharing data between controllers using $rootScope Duration: 0.5 Hours

In this lab, the learners will use the $rootScope service of Angular JS to share data between two or more controllers.

1. Create a view as shown below:



2. When the **Set A Common Message** button is clicked, call a method within the controller which copies the text entered in the textbox into **$rootScope.**

3. When the **Get The Common Message** button is clicked, call a method within the controller which reads the text from **$rootScope.**

4. The text must be displayed in the **span** element below the buttons.

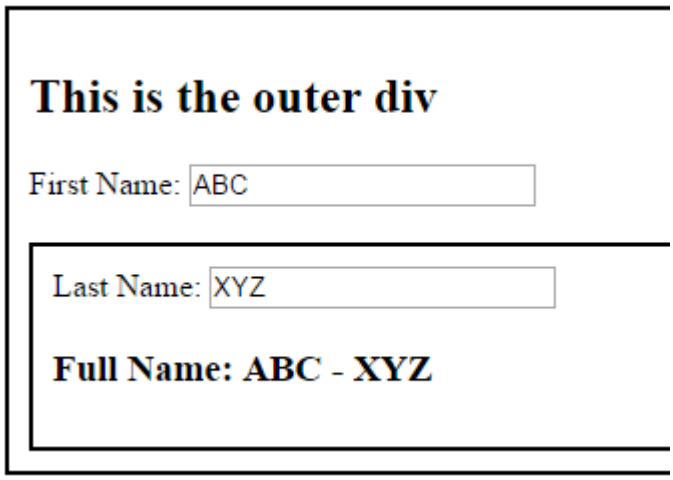
Lab 10: Scope inheritance Duration: 0.5 Hours

In this lab, the learners will implement scope inheritance by using the nested views methodology.

1. Create a controller named **Controller1** & expose a property named **FirstName** via its scope.

2. Create another controller named **Controller2** & expose a property named **LastName** via its scope.

3. Create a view with two nested **div** elements as shown below:



4. The **h3** element inside the second div must display the **FirstName** & **LastName** respectively.

Lab 11: Using $http service for server communication Duration: 2 Hours

In this lab, the learners will use the $http service to perform CRUD operations on an ASP.NET Web API. The learners will also use Angular JS promises to execute callbacks as a part of server communication.

$http Lab 1 (Creating & testing an ASP.NET Web API)

1. Create an ASP.NET Web API.
2. In Visual Studio 2013, choose **File→New→Project.**
3. From the left pane, expand **Visual C#.**
4. Choose **Web.**
5. On the right hand side, choose **ASP.NET Web Application.**
6. Type the project name as **NamesWebAPI** & click **OK.**
7. In the solution explorer, right click on the **Controllers** folder & choose **Add→Controller.**
8. In the list, select **Web API 2 Controller – Empty** & click **Add.**
9. Type the controller name as **NamesController** & click on **Add.**

10. Add the following code inside the **NamesController** class:

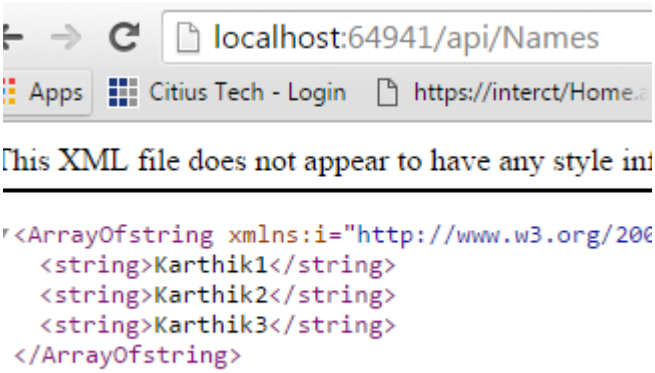
```
List<string> names = new List<string>()
{
            "Karthik1","Karthik2","Karthik3"
};

public HttpResponseMessage GetAllNames()
{
            return Request.CreateResponse<string[]>(HttpStatusCode.OK,
            names.ToArray());
}

public HttpResponseMessage GetNamesStartingWith(string start)
{
            var result = from n in names
                         where n.StartsWith(start)
                         select n;
            if(result.Count() == 0)
            {
                   return Request.CreateErrorResponse(HttpStatusCode.NotFound,
                       "No names found  starting with: " + start);
            }
            return Request.CreateResponse<string[]>(HttpStatusCode.OK,
          result.ToArray());
}
```

11. Build the project.

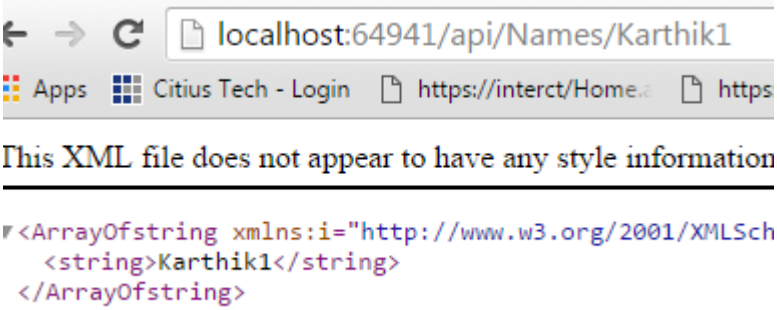
12. From within Visual Studio, choose **Debug→Start Without Debugging.**

13. In the browser's address bar append, **/api/Names:**

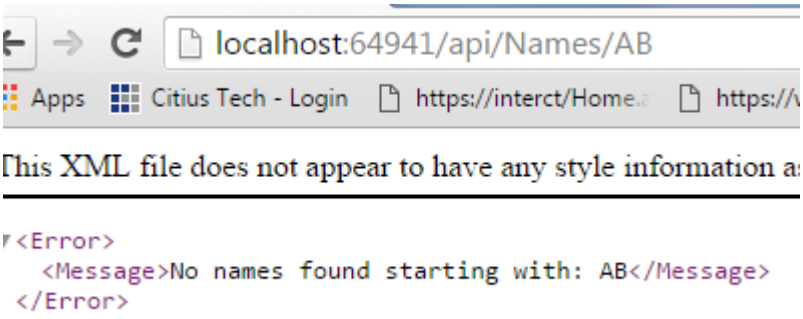
    You should see all the names as shown below:



14. Change the URL in the browser's address bar as shown below to see names starting with a particular text:



15. Change the URL in the browser's address bar as shown below to see an error message if no name exists that starts with a given text:

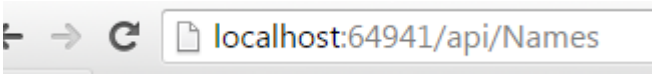


$http Lab 2 (Consuming the ASP.NET Web API using Angular JS)

1. Create an Angular JS controller named **NamesController.**

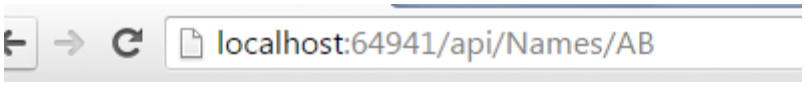
2. Expose a property named **Names** of type array in the scope of the controller. This array will hold the result of the ASP.NET Web API call.

3. Create a method named **GetAllNames()** in the scope of the controller. This method must use the **$http** Angular JS service to make a call to the following URL as shown below:



4. Call this method on a button click from the view. Display the names as a bulleted list inside the view.

5. Create another method named **GetAllNamesStartingWith()** in the scope of the controller. This method must use the **$http** Angular JS service to make a call to the following URL as shown below:



Call this method on a button click from the view & display the results in a bulleted list. The search pattern must be inputted by the user in a textbox in the view & the same must be passed on to the URL above.