Agenda: Controllers

- Understanding Controllers
- Programming Controllers & $scope object
- Possible ways of injecting dependencies in angular application.
- Adding Behavior to a Scope Object
- Passing Parameters to the Methods
- Having Array as members in Controller Scope.
- Nested Controllers and Scope Inheritance.
- Multiple Controllers and their scopes
- Scopeless Controllers

Understanding Controllers

- A controller is a JavaScript object that contains attributes/properties, and functions.
- Controllers are used to add logic to your view
- When a Controller is attached to the DOM via the **ng-controller** directive, Angular will instantiate a new Controller object, using the specified Controller's **constructor function**.
- Each controller accepts **$scope** as a parameter, which refers to the application/module that the controller needs to handle

Use controllers to:

- Set up the initial state of the $scope object.
- Add behavior to the $scope object.

Do not use controllers to:

- Manipulate DOM — Controllers should contain only business logic. Putting any presentation logic into Controllers significantly affects its testability. Angular has databinding for most cases and directives to encapsulate manual DOM manipulation.
- Format input — Use angular form controls instead.
- Filter output — Use angular filters instead.
- Share code or state across controllers — Use angular services instead.
- Manage the life-cycle of other components (for example, to create service instances).

Programming Controllers & $scope object

In AngularJS - 1.2:

- Controllers can be global (scoped to window object) and created by a standard JavaScript **object constructor** having $scope as parameter. AngularJS will internally invoke controllers with a **$scope** object.

- $scope is the application object i.e. the owner of application variables and functions.
- $scope is an object that links Controller to its View. It is controller's responsibility to initialize the data that the view needs to display. This is done by making changes to $scope.
- Scope uses Angular's two-way data binding to bind model data to view

```
<!DOCTYPE html>
<html>
<head>
  <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.26/angular.min.js"></script>
</head>
<body>
  <div ng-app="" ng-controller="personController">
    First Name: <input type="text" ng-model="firstName"><br>
    Last Name: <input type="text" ng-model="lastName"><br>
    <br>
    Full Name: {{fullName()}}
  </div>
  <script>
    function personController($scope) {
      $scope.firstName = "Sandeep"
      $scope.lastName = "Soni"
      $scope.fullName = function () {
        return $scope.firstName + " " + $scope.lastName;
      }
    }
  </script>
</body>
</html>
```

AngularJS - 1.3 will no longer look for controllers on window. The old behavior of looking on window for controllers was originally intended for use in examples and demos apps. Angular found that allowing global controller functions encouraged poor practices, so they resolved to disable this behavior by default.

So now instead we have to do the following:

Option 1: Using $scope object.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title></title>
  <script src="angular.js"></script>
</head>
<body>
  <div ng-app="myApp" ng-controller="personController">
    Enter first name: <input type="text" ng-model="firstName"><br><br>
    Enter last name: <input type="text" ng-model="lastName"><br>
    <br>
    You are entering: {{fullName()}}
  </div>
  <script>
    var myApp = angular.module('myApp', []);
    myApp.controller('personController', ['$scope', function ($scope) {
      $scope.firstName = "Sandeep";
      $scope.lastName = "Soni";
      $scope.fullName = function () {
        return $scope.firstName + " " + $scope.lastName;
      }
    }]);
  </script>
</body>
</html>
```

Correct approach using Model Object

```
<!DOCTYPE html>
<html>
<head>
  <script src="angular.js"></script>
</head>
<body>
  <div ng-app="myApp" ng-controller="personController">
    First Name: <input type="text" ng-model="person.firstName"><br>
```

```
        Last Name: <input type="text" ng-model="person.lastName"><br>
        <br>
        Full Name: {{fullName()}}
    </div>
    <script>
        var myApp = angular.module('myApp', []);
        myApp.controller('personController', function ($scope) {
            $scope.person = {
                firstName: "Sandeep",
                lastName: "Soni"
            }
            $scope.fullName = function () {
                return $scope.person.firstName + " " + $scope.person.lastName;
            }
        });
    </script>
</body>
</html>
```

Also can be written as below:

```
<script>
    var myApp = angular.module('myApp', []);
    function Person(fn, ln)
    {
        this.firstName = fn;
        this.lastName = ln;
    }
    myApp.controller('personController', ['$scope', function (sc) {
        sc.person = new Person("Sandeep", "Soni")
        sc.fullName = function () {
            return sc.person.firstName + " " + sc.person.lastName;
        }
    }])
</script>
```

Using controller members in View without $scope object.

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title></title>
  <script src="angular.js"></script>
</head>
<body>
  <div ng-app="myApp" ng-controller="personController as pc">
    First Name: <input type="text" name="txtFN" ng-model="pc.person.firstName" /> <br />
    Last Name: <input type="text" name="txtLN" ng-model="pc.person.lastName" /> <br />
    <hr />
    {{pc.sayHello()}}
  </div>
  <script>
    var myApp = angular.module("myApp", [])
    myApp.controller("personController", ['$scope', function ($scope) {
      this.person = {
        firstName : "Abc",
        lastName : "Xyz"
      }
      this.sayHello = function () {
        return "Hello " + this.person.firstName + " " + this.person.lastName;
      }
    }])
  </script>
</body>
</html>
```

Possible ways of injecting dependencies in angular application

1. **Passing a dependency as Function Arguments –** Passing dependency as a function argument breaks when we minify the application, because the parameter name will change to a shorter alias name.

```
app.controller("myCtrl", function ($scope) {
   $scope.message = "Hello world";
});
```

2. **Passing a dependency as Array Arguments -** When we pass a dependency as an Array Argument, the application does not break in production when we minify the application. To perform we have two possible ways.

   i. **using named functions** - we are passing a dependency $scope object in the array along with the name of the controller function. More than one dependency can be passed, separated by a comma.

```
function myCtrl($scope) {
    $scope.message = "Hello world";
 };
 app.controller('myCtrl', ['$scope', myCtrl]);
```

   ii. **using Inline Anonymous functions -** it's easy to manage the named controller function. If you prefer the inline function, you can pass dependencies as array arguments exactly the same way you pass them in named controller functions.

```
app.controller('myCtrl', ['$scope', function ($scope) {
         $scope.message = "Hello world";
}]);
```

3. **Passing a dependency using the $inject service -** Here we manually inject the dependencies by using **$inject** service as shown below.

```
function myCtrl($scope) {
   $scope.message = "Hello world";
}
myCtrl.$inject = ['$scope'];
```

Passing Parameters to the Methods

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title></title>
  <script src="angular.js"></script>
</head>
<body>
  <div ng-app="myApp" ng-controller="personController">
    Enter first name: <input type="text" ng-model="firstName"><br><br>
    Enter last name: <input type="text" ng-model="lastName"><br>
    <br>
    You are entering: {{fullName()}}
    <br />
    <button ng-click="conveyWishes('Birthday')">Birthday</button>
    <button ng-click="conveyWishes('Anniversary')">Anniversary</button><br />
    Hello {{fullName()}}, Wishing you a very Happy {{wishes}}
  </div>
  <script>
    var myApp = angular.module('myApp', []);

    myApp.controller('personController', ['$scope', function ($scope) {
      $scope.firstName = "Sandeep";
      $scope.lastName = "Soni";
      $scope.wishes = "(Click on button)";
      $scope.fullName = function () {
        return $scope.firstName + " " + $scope.lastName;
      }
      $scope.conveyWishes = function (wishes) {
        $scope.wishes = wishes;
      }
    }]);
  </script>
</body>
```

```
</html>
```

Array as Model Object

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title></title>
  <script src="angular.js"></script>
</head>
<body>
  <div ng-app="myApp" ng-controller="personController">
    <table>
      <tr>
        <th>First Name</th>
        <th>Last Name</th>
      </tr>
      <tr ng-repeat="person in people">
        <td>{{person.firstName}}</td>
        <td>{{person.lastName}}</td>
        <td><input type="button" value="Select" ng-click="showDetails($index)" /></td>
      </tr>
    </table>
    <hr />
    {{sayHello()}}
  </div>
  <script>
    var myApp = angular.module("myApp", [])
    myApp.controller("personController", ['$scope', function ($scope) {
      $scope.people = [{ firstName: "F1", lastName: "L1" },
                       { firstName: "F2", lastName: "L2" },
                       { firstName: "F3", lastName: "L3" },
                       { firstName: "F4", lastName: "L4" },
                       { firstName: "F5", lastName: "L5" }]
```

```
            $scope.selectedPerson = $scope.people[0];
            $scope.showDetails = function (ind)
            {
               $scope.selectedPerson = $scope.people[ind];
            }
            $scope.sayHello = function () {
               return "Hello " + $scope.selectedPerson.firstName + " " + $scope.selectedPerson.lastName;
            }
         }])
      </script>
</body>
</html>
```

Nested Controllers and Scope Inheritance

Scope is controller-specific. If we define nested controllers, then the child controller inherits the scope of its parent controller.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
   <title></title>
   <script src="angular.js"></script>
   <style>
      div.spicy div {
         padding: 10px;
         border: solid 2px blue;
      }
   </style>
</head>
<body>
   <div ng-app="myApp">
      <div class="spicy">
         <div ng-controller="MainController">
            <p>Good {{timeOfDay}}, {{name}}!</p>
            <div ng-controller="ChildController">
              <p>Good {{timeOfDay}}, {{name}}!</p>

              <div ng-controller="GrandChildController">
                <p>Good {{timeOfDay}}, {{name}}!</p>
              </div>
            </div>
          </div>
        </div>
      </div>
      <script>
        var myApp = angular.module('myApp', []);
        myApp.controller('MainController', ['$scope', function ($scope) {
          $scope.timeOfDay = 'morning';
          $scope.name = 'Nikki';
        }]);
        myApp.controller('ChildController', ['$scope', function ($scope) {
          $scope.name = 'Mattie';
        }]);
        myApp.controller('GrandChildController', ['$scope', function ($scope) {
          $scope.timeOfDay = 'evening';
          $scope.name = 'Gingerbread Baby';
        }]);
      </script>
    </body>
    </html>
```

Notice how we nested three ng-controller directives in our template. This will result in four scopes being created for our view:

- The root scope.
- The MainController scope, which contains timeOfDay and name properties
- The ChildController scope, which inherits the timeOfDay property but overrides (hides) the name property from the previous
- The GrandChildController scope, which overrides (hides) both the timeOfDay property defined in MainController and the name property defined in ChildController

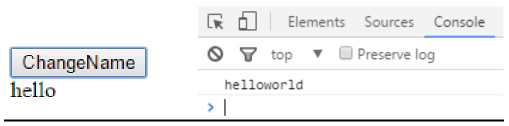
Inheritance works with methods in the same way as it does with properties. So in our previous examples, all of the properties could be replaced with methods that return string values.

Scopeless Controllers

- If an application does not depend upon communication with other controllers, or does not require any inheritance feature, you can use **Scopeless Controllers.** Scopeless Controllers are useful for small applications, to avoid complexities. Scopeless Controlles will be using this keyword instead $scope**.**
- But this is not same as $scope.
- $scope is more powerful and its use allows you to share the same data between controllers.
- $watch services are attached with $scope which allows the view to be updated, where this doesn't have such services. So that views will not get updated when using this.

Example 1:

Using **$scope**	Using **this**
<u>View Code:</u>	
`<div ng-app="myapp">`	<u>View Code:</u>
`    <div ng-controller="myCtrl">`	
`      <button ng-`	`<div ng-app="myapp">`
`click="changeName()">ChangeName</button><br />`	`    <div ng-controller="myCtrl as Ctrl">`
`        {{myname}}`	`      <button ng-`
`    </div>`	`click="Ctrl.changeName()">ChangeName</button><br />`
`  </div>`	`        {{Ctrl.myname}}`
<u>ScriptCode:</u>	`    </div>`
`app.controller('myCtrl', function ($scope) {`	`  </div>`
`    $scope.myname = "hello";`	
`    $scope.changeName = function () {`	<u>Script Code:</u>
`      $scope.myname = "Helloworld";`	
`    }`	`app.controller('myCtrl', function () {`
`  });`	`    this.myname = "hello";`
	`    this.changeName = function () {`
	`      this.name = "helloworld";`
	`      console.log(this.name);`
	`    }`
	`  });`

	O/p: On clicking this button view remains same. That changes can be seen in Console.  hello `helloworld`