Getting started with AngularJS and ASP.NET MVC - Part One

JMK-NI, 17 Nov 2014 CPOL

4.92 (241 votes)

How to hit the ground running with AngularJS on ASP.NET MVC: Part 1

Welcome to Part One of a series of articles regarding AngularJS and ASP.NET MVC, and utilizing and getting the most out of both

Articles in series:

- Part One (this article)
- Part Two
- Part Three

Introduction

Client side MV* frameworks are all the rage at the moment, and my favourite by far is AngularJS. This article will show you how to hit the ground running with AngularJS while using my favourite server side MVC framework, ASP.NET MVC (5).

Source Code

The source code accompanying this article can be found here.

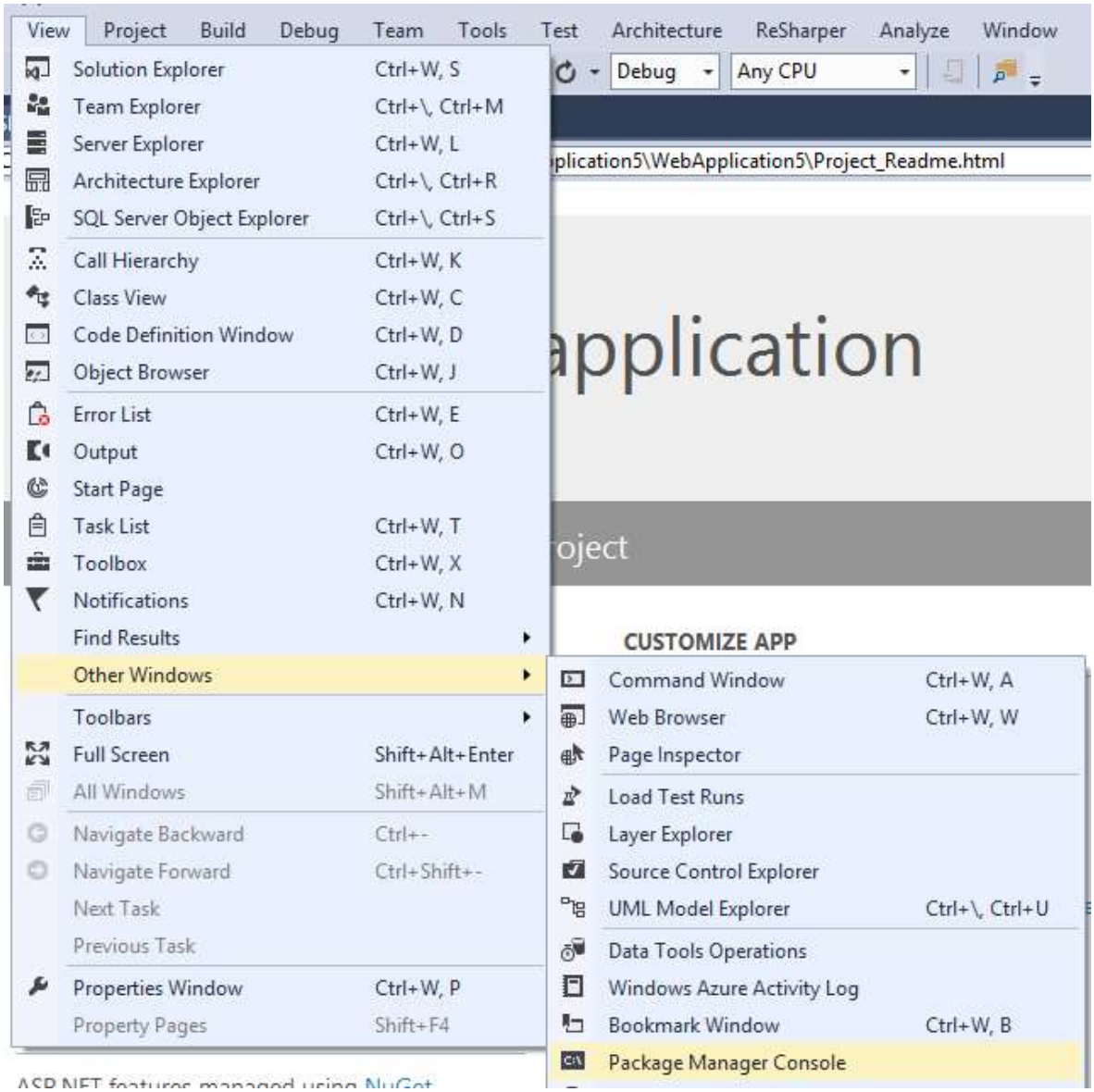
What you need

I am using Visual Studio 2013 Update 3, so I can only guarantee that the following will work as advertised with this IDE, but unofficially you should be able to follow along on any version of VS >= 2010, as long as you are working with at least MVC 4 (things haven't changed that dramatically since then, at least not for what we will be doing). Express should be fine.

Let's get started

Ok, first of all, you want to create a new MVC project by selecting **New Project...** on the start screen, then drill down to *Templates* => *Visual C#* (or *Visual Basic* if you prefer, and are confident translating any C# in this article to it's VB equivalent) => *Web* => *ASP.NET Web Application* => *MVC*. Leave **Authentication** as Individual User Accounts and click OK.

We now have the default MVC template, containing a lot of stuff we do need, and a couple of things we don't so let's get rid of those. Open the **Package manager Console** from View => Other Windows => Package Manager Console as shown below:



Killing spree

Run the following commands:

- Uninstall-Package Microsoft.jQuery.Unobtrusive.Validation
- Uninstall-Package jQuery.Validation
- Uninstall-Package jQuery
- Uninstall-Package Modernizr
- Uninstall-Package Respond
- Uninstall-Package bootstrap

We now need to update our BundleConfig.cs class to reflect this, mine now looks like so:

```
using System.Web.Optimization;

namespace AwesomeAngularMVCApp
{
    public class BundleConfig
    {
        public static void RegisterBundles(BundleCollection bundles)
        {
            bundles.Add(new StyleBundle("~/Content/css").Include(
                      "~/Content/site.css"));

            BundleTable.EnableOptimizations = true;
        }
```

```
    }
}
```

Delete the following files from the **Controllers** directory:

- AccountController.cs
- ManageController.cs

And the following directories from the **Views** directory:

- Account
- Manage
- Shared

While you're at it, delete **_ViewStart.cshtml** from the **Views** directory also, and delete the following from the **Views/Home** directory:

- About.cshtml
- Contact.cshtml

Next, delete the following Action methods from **Controllers/HomeController.cs**:

- About
- Contact

Finally, open **Views** => **Home** => **Index.cshtml**, and delete everything you find in there (leave no survivors), do the same with **Content** => **Site.css** (delete the contents, leave the file).

It seems as though we just deleted everything, why didn't we use the blank template?

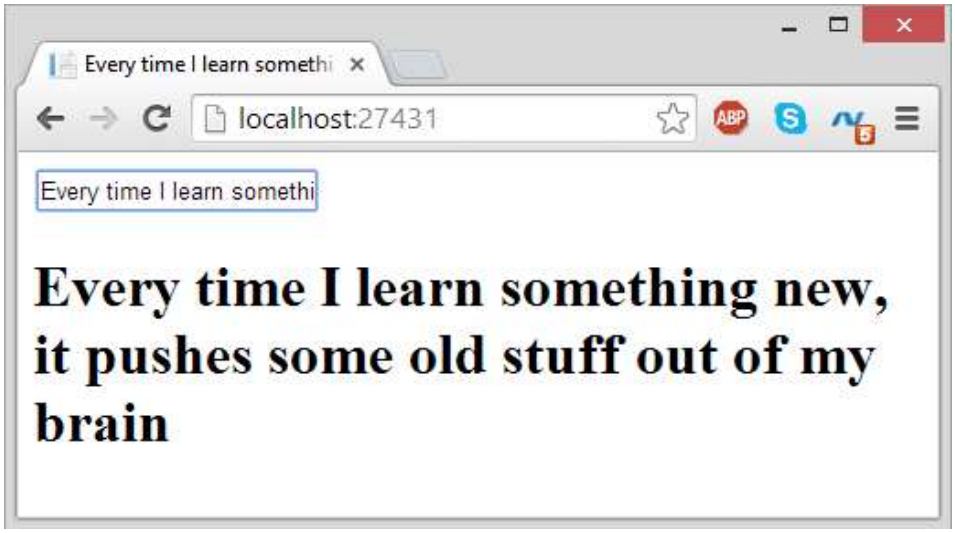
Visual Studio added all of the correct libraries/boilerplate code we need for authentication etc when we selected this template. Deleting the unnecessary html/javascript was a lot faster than using the blank template and adding everything needed for authentication.

Hello AngularJS

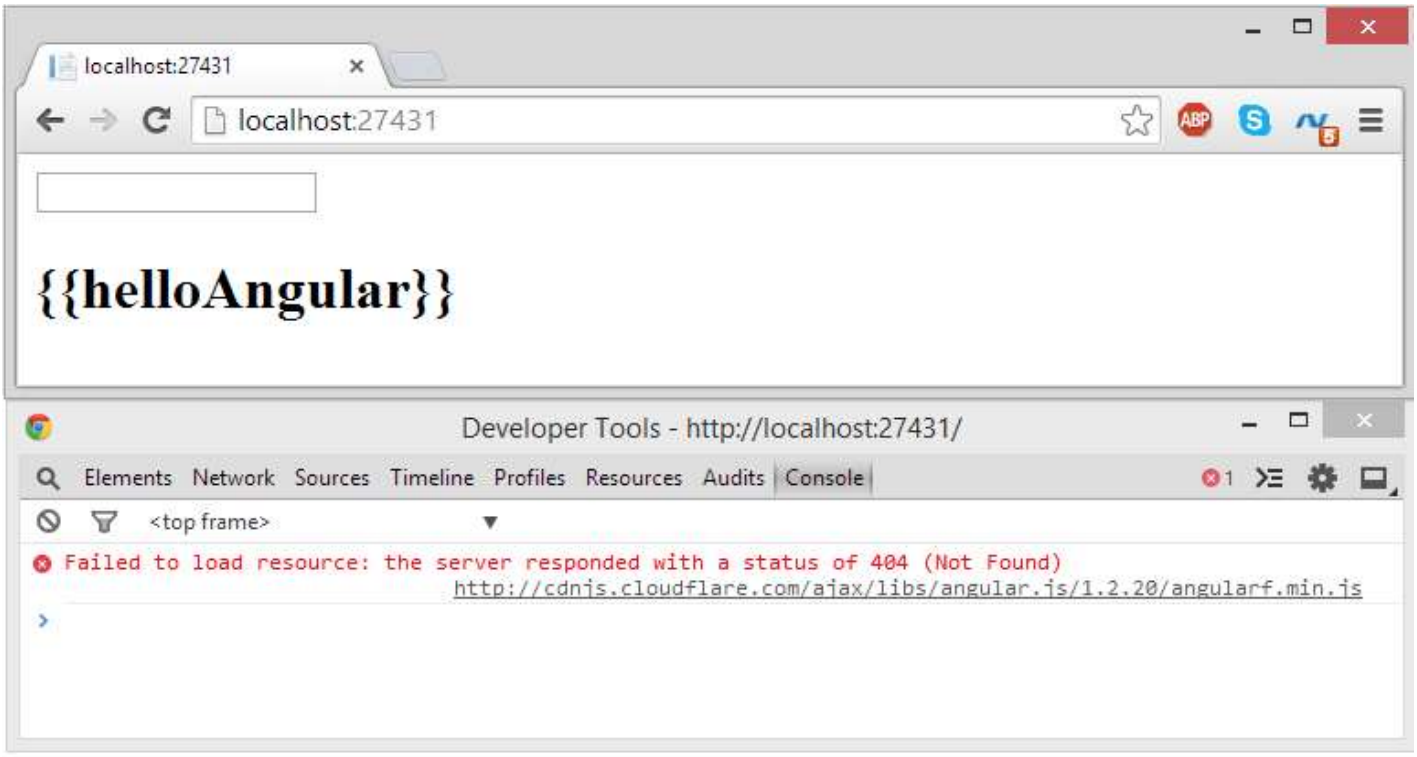
Add the following to the now empty Index.cshtml in **Views** => **Home**

```
<!DOCTYPE html>
<html ng-app>
<head>
    <title ng-bind="helloAngular"></title>
</head>
<body>

    <input type="text" ng-model="helloAngular" />
    <h1>{{helloAngular}}</h1>

    <script src="//cdnjs.cloudflare.com/ajax/libs/angular.js/1.2.20/angular.min.js"></script>
</body>
</html>
```

Cool, now hit F5 (or whatever you have configured your debug key in Visual Studio as), and type something into the textbox. All being well, you should see something like this:



If you are seeing something closer to **{{helloAngular}}** then you have made a mistake somewhere. Your first port of call here should be your browsers developer console. In Chrome, you find this by hitting **F12** and selecting the Console tab on the window that appears. In the example below, my Angular app isn't working because I have mistyped the URL of the angular.js file (d'oh):



What's going on here?

I'm assuming at this point that everything is working as expected. The HTML you typed into your index file is a full Angular application!

All we are doing is loading the Angular runtime itself, declaring a model object (a model in Angular is akin to the property of a model in .Net) and that model is bound in 3 places, the pages title, the contents of the **h1** tag, and **value** property of our text input control.

Editing the textbox causes the other two places the model is bound to automatically update. This is known as 2 way binding, pretty cool, no?

This isn't good practice

The above is a great way to quickly demonstrate a very basic AngularJS application, but it isn't how you write Angular apps in the real world.

We need to add a little bit of boilerplate, which we will do now:

- Add a directory to **Scripts** called **Controllers**
- Add a Javascript file to the root of **Scripts** with the name of your Application (mine is called **AwesomeAngularMVCApp.js**)
- Add a Javascript file to your **Scripts** => **Controllers** directory called **LandingPageController.js**
- Now let's add a bundle for the above to BundleConfig.cs, something like this:

```
bundles.Add(new ScriptBundle("~/bundles/AwesomeAngularMVCApp")
    .IncludeDirectory("~/Scripts/Controllers", "*.js")
    .Include("~/Scripts/AwesomeAngularMVCApp.js"));
```

And add this bundle to the homepage, after the angular.min script tag itself:

```
<script src="//cdnjs.cloudflare.com/ajax/libs/angular.js/1.2.20/angular.min.js"></script
```

- Next we need to write our LandingPageController, it should look like this

```
var LandingPageController = function($scope) {
    $scope.models = {
        helloAngular: 'I work!'
    };
}

// The $inject property of every controller (and pretty much every other type of object in Angular) needs to be a
string array equal to the controllers arguments, only as strings
LandingPageController.$inject = ['$scope'];
```

- After this, we want to setup our application object itself, and tell it about the controller. Add the following to **AwesomeAngularMVCApp.js** (or whatever yours is called):

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

AwesomeAngularMVCApp.controller('LandingPageController', LandingPageController);
```

Nice job, now we want to update our homepage view to properly bind it to our angular application and Landing Page controller:

```
<!DOCTYPE html>
<html ng-app="AwesomeAngularMVCApp" ng-controller="LandingPageController">
<head>
    <title ng-bind="models.helloAngular"></title>
</head>
<body>
    <input type="text" ng-model="models.helloAngular" />
    <h1>{{models.helloAngular}}</h1>

    <script src="//cdnjs.cloudflare.com/ajax/libs/angular.js/1.2.20/angular.min.js"></script>
    @Scripts.Render("~/bundles/AwesomeAngularMVCApp")
</body>
</html>
```

All being well, you should see this the next time you hit debug:

Routing

Where client side MV* frameworks like Angular get really powerful is when it comes to routing. This provides us with the building blocks for building single page applications.

The way this works is straight forward enough. We put a div on our landing page, and tell angular that, whenever the URL matches a particular pattern, to put contents of another HTML file into that div.

Let's add some routing to our application:

- First of all, we need to include Angular's ngRoute module, which I'm going to pull down from Cloudflare
- Next, let's add that container div to our landing page, inside the body tag, before our script tags
- Finally, we will add a couple of links to our landing page, which, when clicked will update the route and load the appropriate view into the container div

My HTML currently looks like so:

```
<!DOCTYPE html>
<html ng-app="AwesomeAngularMVCApp" ng-controller="LandingPageController">
<head>
    <title ng-bind="models.helloAngular"></title>
</head>
<body>
    <h1>{{models.helloAngular}}</h1>

    <ul>
        <li><a href="/#/routeOne">Route One</a></li>
        <li><a href="/#/routeTwo">Route Two</a></li>
        <li><a href="/#/routeThree">Route Three</a></li>
    </ul>

    <div ng-view></div>

    <script src="//cdnjs.cloudflare.com/ajax/libs/angular.js/1.2.20/angular.min.js"></script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/angular.js/1.2.20/angular-route.min.js"></script>
    @Scripts.Render("~/bundles/AwesomeAngularMVCApp")
    </body>
</html>
```

Now we want to tell our AngularJS application about these routes. We do this in the application module's config function (inside the same Javascript file in which we declared the application, at the root of our Scripts directory), mine now looks like this:

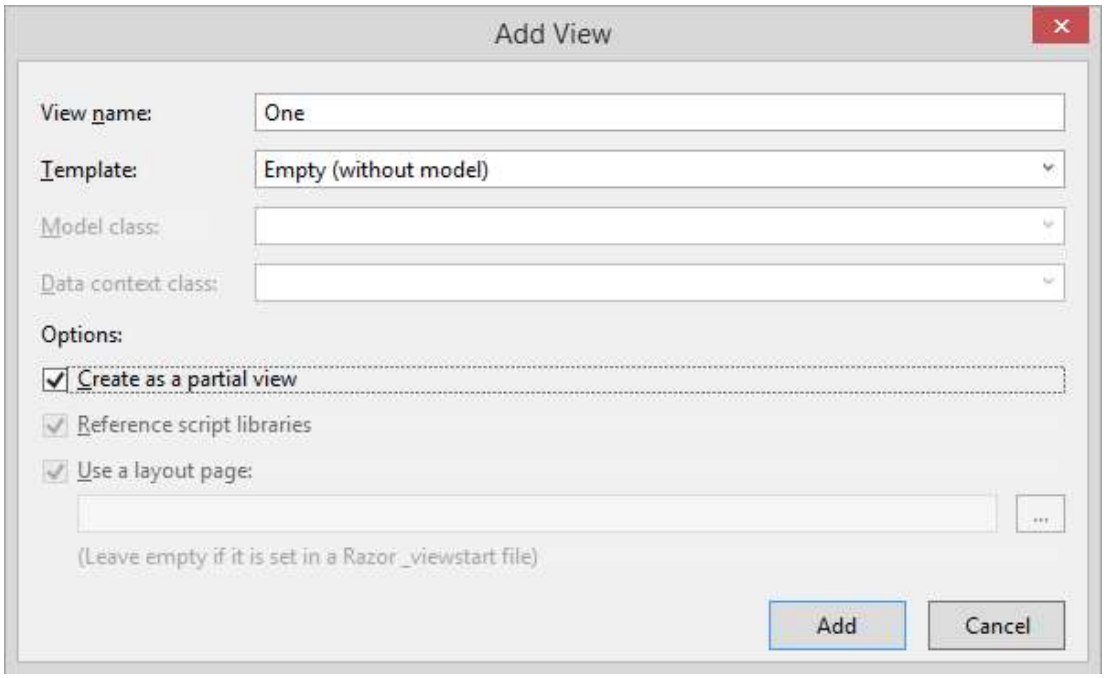
```
var AwesomeAngularMVCApp = angular.module('AwesomeAngularMVCApp', ['ngRoute']);

AwesomeAngularMVCApp.controller('LandingPageController', LandingPageController);

var configFunction = function ($routeProvider) {
    $routeProvider.
        when('/routeOne', {
            templateUrl: 'routesDemo/one'
        })
        .when('/routeTwo', {
            templateUrl: 'routesDemo/two'
        })
        .when('/routeThree', {
            templateUrl: 'routesDemo/three'
        });
}
configFunction.$inject = ['$routeProvider'];

AwesomeAngularMVCApp.config(configFunction);
```

So far so good, but these views don't exist yet. Let's fix that, create a C# controller inside the MVC controller directory called **RoutesDemoController**, and add three Action methods called **One**, **Two** and **Three**, like so:

```
using System.Web.Mvc;

namespace AwesomeAngularMVCApp.Controllers
{
    public class RoutesDemoController : Controller
    {
        public ActionResult One()
        {
            return View();
        }
```

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

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

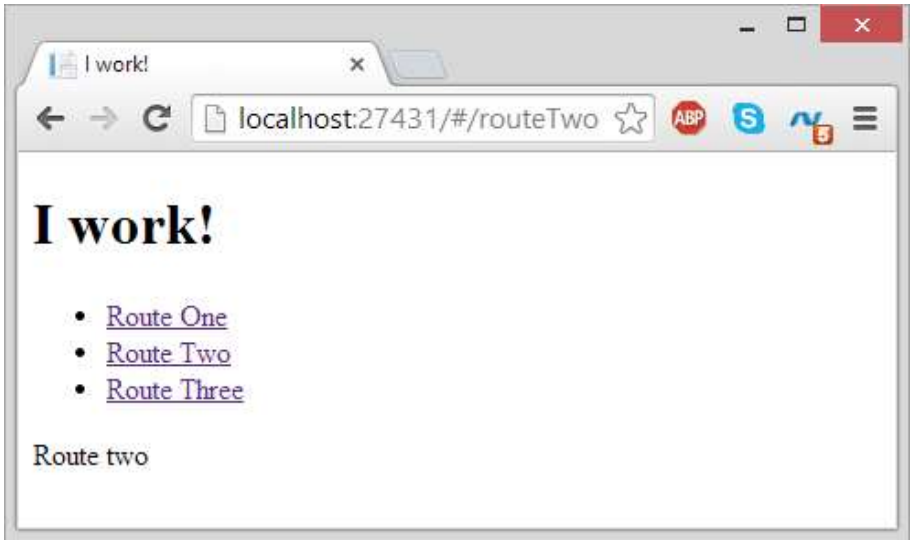
Right click on each instance of **return View();** and select **Add View...**, tick **Create as a partial view** on the following dialog box leaving everything else as-is and click **Add:**



Add some content to each view so you can uniquely identify it, I have added the words "Route One", "Route Two" and "Route three" to each of mine.

***Important**: At this point, you might find that Visual Studio has helpfully added some of the stuff we deleted earlier back in. If so, you may need to skim through the Let's get started section again to make sure nothing is there that shouldn't be. Sorry!*

Now hit F5. If all is well, you should be able to click each of the links, loading the contents of that view into the container div, similar to below:



Hitting refresh also works, as does the back button. We haven't technically moved away from the landing page, but we can dynamically load content into a div, hit the back to go to the previous content, and when we hit refresh the state of the page is the same as before.

'Route two' takes parameters

Let's modify route two to take a parameter, which will be passed all the way from angular through our MVC controller, going to our MVC view. Update the C# action method like so:

```
public ActionResult Two(int donuts = 1)
{
    ViewBag.Donuts = donuts;
    return View();
}
```

Now update the view itself so it looks like this:

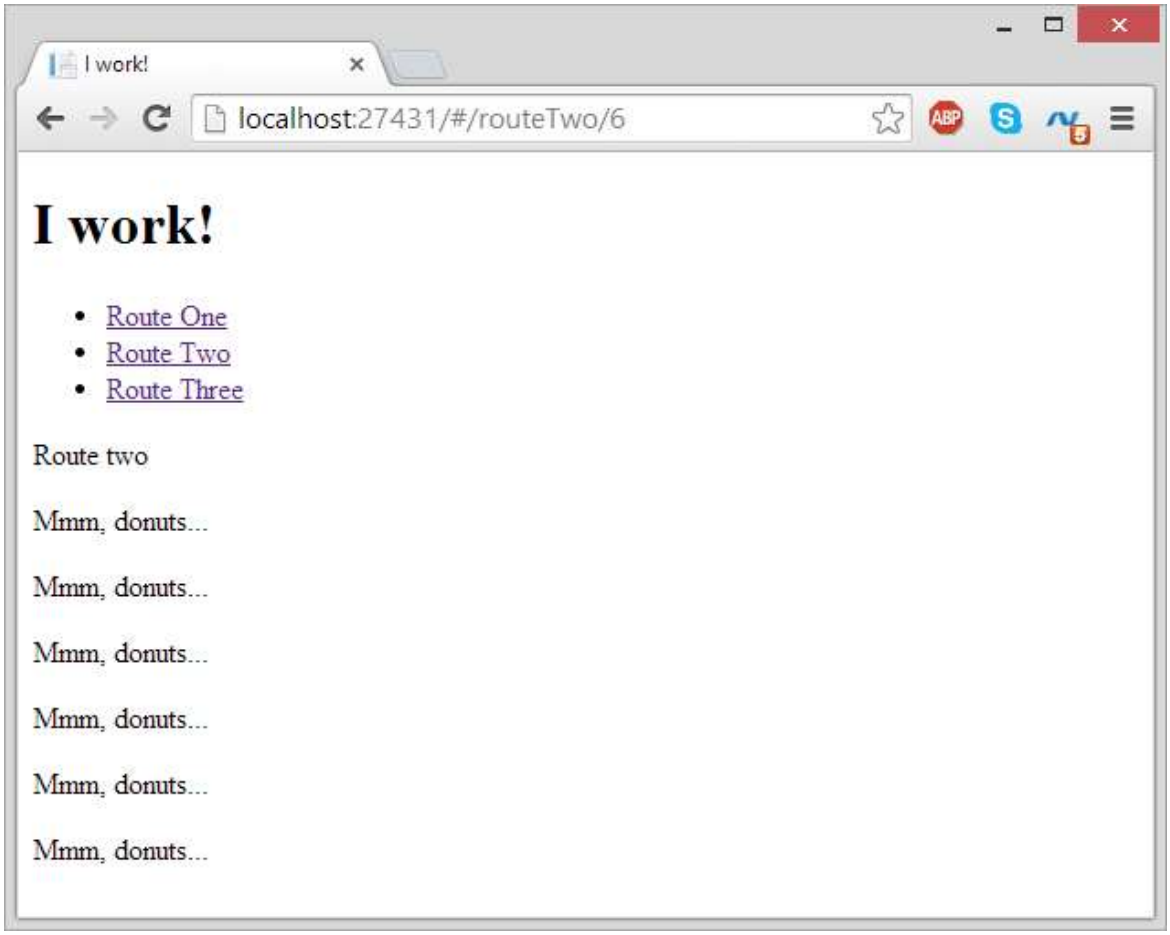
```
Route two

@for (var i = 0; i < ViewBag.Donuts; i++)
{
    <p>Mmm, donuts...</p>
}
```

Now we need to tell angular about this parameter. Modify the config function in AwesomeAngularMVCApp.js like so:

```
var configFunction = function ($routeProvider) {
    $routeProvider.
        when('/routeOne', {
            templateUrl: 'routesDemo/one'
        })
        .when('/routeTwo/:donuts', {
            templateUrl: function (params) { return '/routesDemo/two?donuts=' + params.donuts; }
        })
        .when('/routeThree', {
            templateUrl: 'routesDemo/three'
        });
}
```

And the route two hyperlink in our landing page could look like this:

```
<li><a href="/#/routeTwo/6">Route Two</a></li>
```



'Route three' is for registered users only

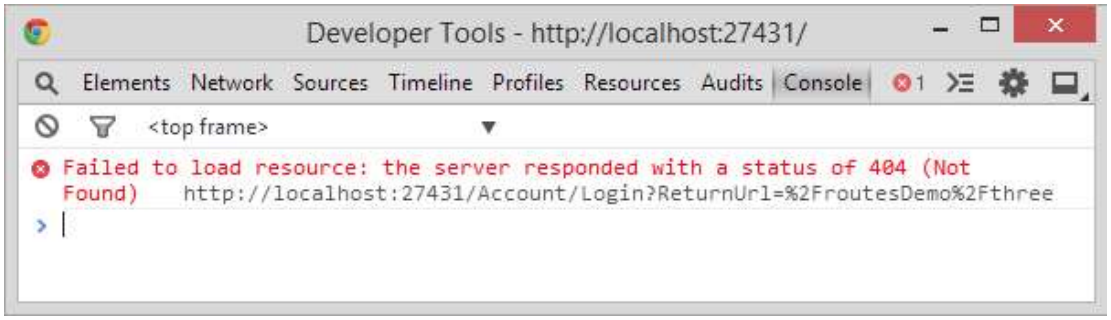
The way things are right now, anybody on the internet can visit our site and see the contents of routes one two and three. Epic, but route three is going to contain some sensitive data, how do we protect that?

Using ASP.NET authentication of course:

- Add the Authorize attribute to Route three's action method, like so:

```
[Authorize]
public ActionResult Three()
{
    return View();
}
```

If we run our app at this point and try to navigate to route three, we will get this epicness in our browsers developer console:



This actually means that everything is working as expected so far, but before we go any further with route three, let's get some Authentication on the go.

Authentication

At the moment, ASP.NET is detecting that the user doesn't have permission to access 'Route 3' and trying to redirect to "/Account/Login", directing us away from our landing page. What we really want is to deal with this 401 response ourselves in Angular. You do this with an interceptor.

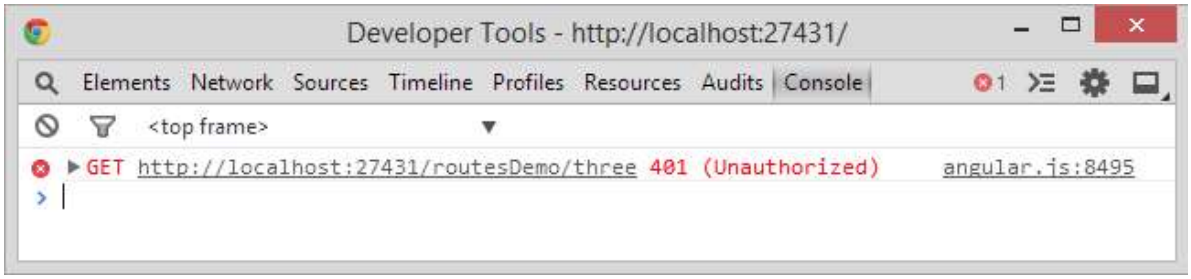
Let's do this now. The first step is stopping ASP.NET MVC from redirecting on a 401, edit **App_Start** => **Startup.Auth.cs.** Look for the following:

```
LoginPath = new PathString("/Account/Login")
```

And replace it with this:

```
LoginPath = new PathString(string.Empty)
```

Now when we try to access 'Route 3' we get the following error in the browsers developer console:



Much better. Let's handle this with an interceptor. Create a directory inside **Scripts** called **Factories**, and update the ScriptBundle inside **BundleConfig** to include this directory:

```
bundles.Add(new ScriptBundle("~/bundles/AwesomeAngularMVCApp")
   .IncludeDirectory("~/Scripts/Controllers", "*.js")
   .IncludeDirectory("~/Scripts/Factories", "*.js")
   .Include("~/Scripts/AwesomeAngularMVCApp.js"));
```

Create a new Javascript file inside **Scripts** => **Factories** called **AuthHttpResponseInterceptor.js** containing the following Angular factory (borrowed from the awesome SparkTree blog):

```
var AuthHttpResponseInterceptor = function($q, $location) {
    return {
        response: function (response) {
            if (response.status === 401) {
                console.log("Response 401");
            }
            return response || $q.when(response);
        },
        responseError: function (rejection) {
            if (rejection.status === 401) {
                console.log("Response Error 401", rejection);
                $location.path('/login').search('returnUrl', $location.path());
            }
            return $q.reject(rejection);
        }
    }
}

AuthHttpResponseInterceptor.$inject = ['$q', '$location'];
```

Now we need to tell our Angular app about, and configure it to use the interceptor. Update **AwesomeAngularMVCApp.js** like so:

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

AwesomeAngularMVCApp.controller('LandingPageController', LandingPageController);
AwesomeAngularMVCApp.controller('LoginController', LoginController);

AwesomeAngularMVCApp.factory('AuthHttpResponseInterceptor', AuthHttpResponseInterceptor);

var configFunction = function ($routeProvider, $httpProvider) {
    $routeProvider.
        when('/routeOne', {
```

```
            templateUrl: 'routesDemo/one'
        })
        .when('/routeTwo/:donuts', {
            templateUrl: function (params) { return '/routesDemo/two?donuts=' + params.donuts; }
        })
        .when('/routeThree', {
            templateUrl: 'routesDemo/three'
        })
        .when('/login', {
            templateUrl: '/Account/Login',
            controller: LoginController
        });

    $httpProvider.interceptors.push('AuthHttpResponseInterceptor');
}
configFunction.$inject = ['$routeProvider', '$httpProvider'];

AwesomeAngularMVCApp.config(configFunction);
```

We now have Angular redirecting properly when authentication fails. We just need something to redirect to.

At the beginning of this tutorial, I asked you to delete the C# AccountController, and now I am about to ask you to create a C# AccountController. This may seem counterproductive, but stick with me. The file you deleted was nearly 500 lines long, and we only need a fraction of that. Also, it's good to know exactly what each line of code does, and not treat Authentication in ASP.NET as a black box.

So go ahead and add a C# Account controller, initially containing the following:

```
using System.Web.Mvc;

namespace AwesomeAngularMVCApp.Controllers
{
    [Authorize]
    public class AccountController : Controller
    {
        [AllowAnonymous]
        public ActionResult Login()
        {
            return View();
        }
    }
}
```

Use Visual Studio to create the Login view, the same way you did before. This view is going to need an Angular controller, so let's add a Javascript file called **LoginController.js** to **Scripts** => **Controllers,** containing the following Javascript:

```
var LoginController = function($scope, $routeParams) {
    $scope.loginForm = {
        emailAddress: '',
        password: '',
        rememberMe: false,
        returnUrl: $routeParams.returnUrl
    };

    $scope.login = function() {
        //todo
    }
}

LoginController.$inject = ['$scope', '$routeParams'];
```

The route in **AwesomeAngularMVCApp.js** needs to be updated so Angular knows to provide the above controller to the login view. We also need to tell our application module that the controller exists:

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

AwesomeAngularMVCApp.controller('LandingPageController', LandingPageController);
AwesomeAngularMVCApp.controller('LoginController', LoginController);

var configFunction = function($routeProvider, $routeParams) {
    $routeProvider.
        when('/routeOne', {
            templateUrl: 'routesDemo/one'
        })
```

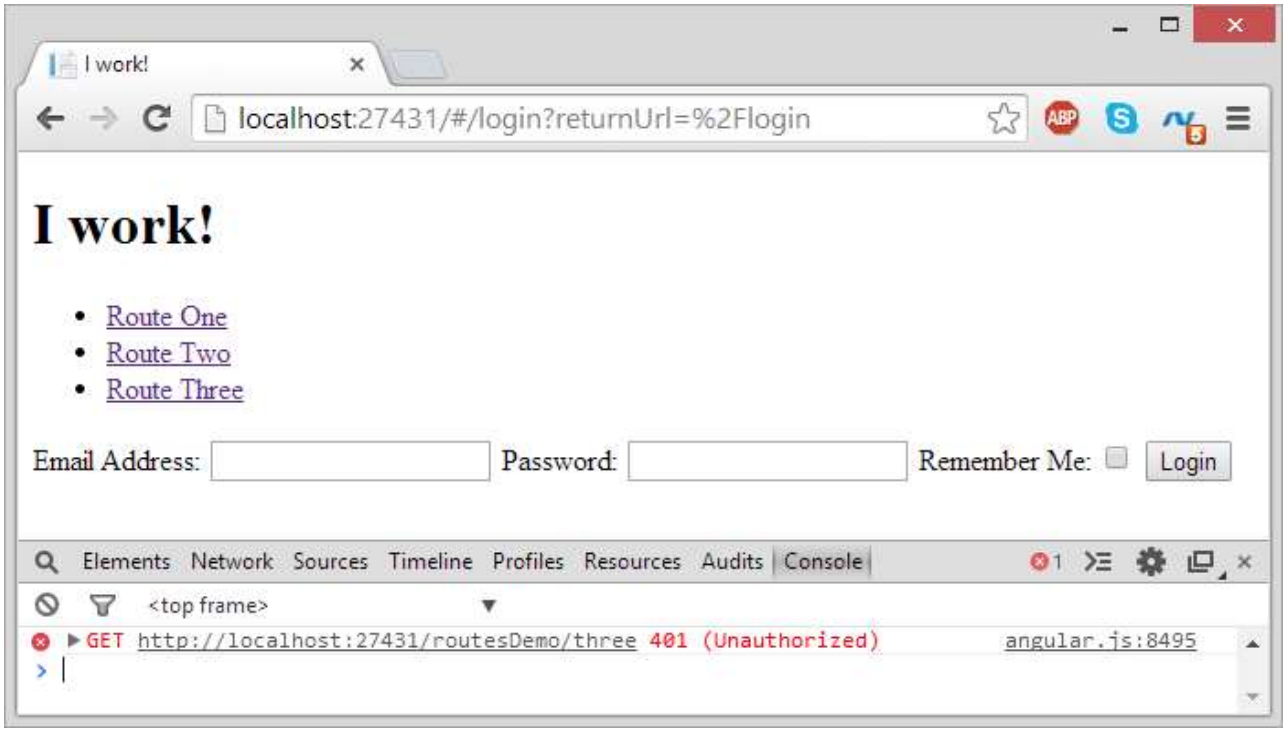
```
        .when('/routeTwo/:donuts', {
            templateUrl: function(params) { return '/routesDemo/two?donuts=' + params.donuts; }
        })
        .when('/routeThree', {
            templateUrl: 'routesDemo/three'
        })
        .when('/login?returnUrl', {
            templateUrl: 'Account/Login',
            controller: LoginController
        });
}
configFunction.$inject = ['$routeProvider'];

AwesomeAngularMVCApp.config(configFunction);
```

Now lets add the login form itself to the view at **Views** => **Account** => **Login.cshtml**:

```
<form ng-submit="login()">
    <label for="emailAddress">Email Address:</label>
    <input type="text" ng-model="loginForm.emailAddress" id="emailAddress" required />

    <label for="password">Password:</label>
    <input type="password" id="password" ng-model="loginForm.password" required />

    <label for="rememberMe">Remember Me:</label>
    <input type="checkbox" id="rememberMe" ng-model="loginForm.rememberMe" required />

    <button type="submit">Login</button>
</form>
```

Now whenever we try to access 'Route three', two things will happen:

1. We get a 401 unauthorized response, which is logged in the browser console by angular
2. Our inteceptor kicks into action, and redirects us to the Login view we created



We're getting close! We also need a register form for new users, lets create this now.

Add the following Action Method to **AccountController.cs**.

```
[AllowAnonymous]
public ActionResult Register()
{
    return View();
}
```

We need an Angular controller for our Register view. Add a Javascript file called **RegisterController.js** to **Scripts** => **Controllers** containing the following code:

```
var RegisterController = function($scope) {
    $scope.registerForm = {
        emailAddress: '',
        password: '',
        confirmPassword: ''
    };

    $scope.register = function() {
        //todo
    }
}

RegisterController.$inject = ['$scope'];
```

Use Visual Studio to create the view, containing the following HTML (hopefully you will start to recognise a pattern):

```
<form ng-submit="register()">
    <label for="emailAddress">Email Address:</label>
    <input id="emailAddress" type="text" ng-model="registerForm.emailAddress" required />

    <label for="password">Password:</label>
    <input id="password" type="password" ng-model="registerForm.password" required />

    <label for="confirmPassword">Confirm Password:</label>
    <input id="confirmPassword" type="password" ng-model="registerForm.confirmPassword" required />

    <button type="submit">Register</button>
</form>
```

Now just update **AwesomeAngularMVCApp.js** with information about this new angular controller and route:

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

AwesomeAngularMVCApp.controller('LandingPageController', LandingPageController);
AwesomeAngularMVCApp.controller('LoginController', LoginController);
AwesomeAngularMVCApp.controller('RegisterController', RegisterController);

AwesomeAngularMVCApp.factory('AuthHttpResponseInterceptor', AuthHttpResponseInterceptor);

var configFunction = function ($routeProvider, $httpProvider) {
    $routeProvider.
        when('/routeOne', {
            templateUrl: 'routesDemo/one'
        })
        .when('/routeTwo/:donuts', {
            templateUrl: function (params) { return '/routesDemo/two?donuts=' + params.donuts; }
        })
        .when('/routeThree', {
            templateUrl: 'routesDemo/three'
        })
        .when('/login', {
            templateUrl: '/Account/Login',
            controller: LoginController
        })
        .when('/register', {
            templateUrl: '/Account/Register',
            controller: RegisterController
        });

    $httpProvider.interceptors.push('AuthHttpResponseInterceptor');
}
configFunction.$inject = ['$routeProvider', '$httpProvider'];

AwesomeAngularMVCApp.config(configFunction);
```

Now would be a good time to add login and register links to out landing page also:

```
<!DOCTYPE html>
<html ng-app="AwesomeAngularMVCApp" ng-controller="LandingPageController">
<head>
```

```
    <title ng-bind="models.helloAngular"></title>
</head>
<body>
    <h1>{{models.helloAngular}}</h1>

    <ul>
        <li><a href="/#/routeOne">Route One</a></li>
        <li><a href="/#/routeTwo/6">Route Two</a></li>
        <li><a href="/#/routeThree">Route Three</a></li>
    </ul>

    <ul>
        <li><a href="/#/login">Login</a></li>
        <li><a href="/#/register">Register</a></li>
    </ul>

    <div ng-view></div>

    <script src="//cdnjs.cloudflare.com/ajax/libs/angular.js/1.2.20/angular.min.js"></script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/angular.js/1.2.20/angular-route.min.js"></script>
    @Scripts.Render("~/bundles/AwesomeAngularMVCApp")
    </body>
</html>
```

Our C# **AccountController** needs a little work. It is missing a couple of dependencies so we are going to update it to include an **ApplicationUserManager** and an **ApplicationSignInManager**, which are provided through constructor injection, and created on the fly as a fallback. We also need to add methods for logging in and registration:

```
using Microsoft.AspNet.Identity.Owin;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;
using AwesomeAngularMVCApp.Models;

namespace AwesomeAngularMVCApp.Controllers
{
    [Authorize]
    public class AccountController : Controller
    {
        private ApplicationUserManager _userManager;
        private ApplicationSignInManager _signInManager;

        public AccountController() { }

        public AccountController(ApplicationUserManager userManager, ApplicationSignInManager signInManager)
        {
            UserManager = userManager;
            SignInManager = signInManager;
        }

        public ApplicationUserManager UserManager
        {
            get { return _userManager ?? HttpContext.GetOwinContext().GetUserManager<ApplicationUserManager>(); }
            private set { _userManager = value; }
        }

        public ApplicationSignInManager SignInManager
        {
            get { return _signInManager ?? HttpContext.GetOwinContext().Get<ApplicationSignInManager>(); }
            private set { _signInManager = value; }
        }

        [AllowAnonymous]
        public ActionResult Login()
        {
            return View();
        }

        [AllowAnonymous]
        public ActionResult Register()
        {
            return View();
        }

        [HttpPost]
```

```
        [AllowAnonymous]
        public async Task<bool> Login(LoginViewModel model)
        {
            var result = await SignInManager.PasswordSignInAsync(model.Email, model.Password, model.RememberMe,
false);
            switch (result)
            {
                case SignInStatus.Success:
                    return true;
                default:
                    ModelState.AddModelError("", "Invalid login attempt.");
                    return false;
            }
        }

        [HttpPost]
        [AllowAnonymous]
        public async Task<bool> Register(RegisterViewModel model)
        {
            var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
            var result = await UserManager.CreateAsync(user, model.Password);
            if (!result.Succeeded) return false;
            await SignInManager.SignInAsync(user, false, false);
            return true;
        }
    }
}
```

We are very close. The next step is to populate the **login** and **register** functions in the respective AngularJS controllers. We don't want to put this logic into the controllers themselves however, a much better approach is to create an Angular factory for each, adhering to the Single Responsibility principle, as well as the principle of least knowledge.

Add a Login factory called **LoginFactory.js** to **Scripts** => **Factories.** Add the following Javascript to the factory:

```
var LoginFactory = function ($http, $q) {
    return function (emailAddress, password, rememberMe) {

        var deferredObject = $q.defer();

        $http.post(
            '/Account/Login', {
                Email: emailAddress,
                Password: password,
                RememberMe: rememberMe
            }
        ).
        success(function (data) {
            if (data == "True") {
                deferredObject.resolve({ success: true });
            } else {
                deferredObject.resolve({ success: false });
            }
        }).
        error(function () {
            deferredObject.resolve({ success: false });
        });

        return deferredObject.promise;
    }
}

LoginFactory.$inject = ['$http', '$q'];
```

Now let's tell our Angular application about this factory:

```
AwesomeAngularMVCApp.factory('LoginFactory', LoginFactory);
```

Inject this factory into our LoginController, and call it's function when the controllers **login** function is called:

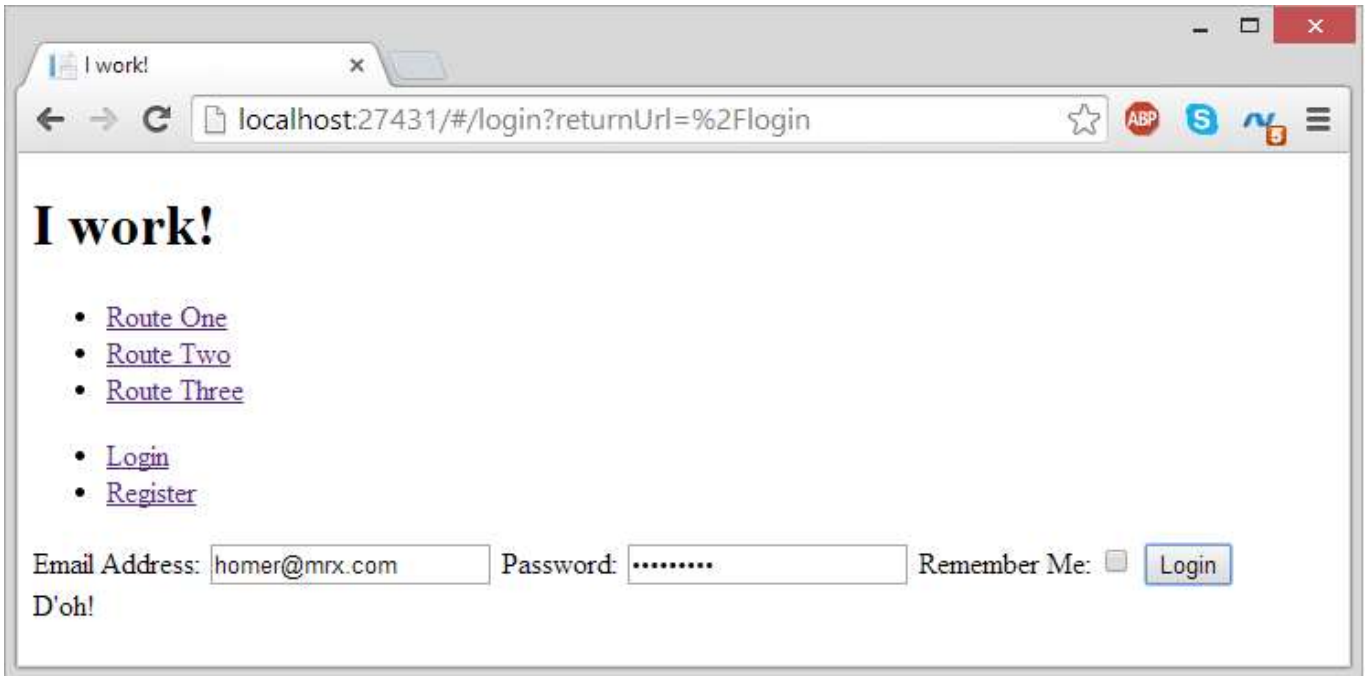
```
var LoginController = function ($scope, $routeParams, $location, LoginFactory) {
    $scope.loginForm = {
        emailAddress: '',
        password: '',
```

```
        rememberMe: false,
        returnUrl: $routeParams.returnUrl,
        loginFailure: false
    };

    $scope.login = function () {
        var result = LoginFactory($scope.loginForm.emailAddress, $scope.loginForm.password,
$scope.loginForm.rememberMe);
        result.then(function(result) {
            if (result.success) {
                if ($scope.loginForm.returnUrl !== undefined) {
                    $location.path('/routeOne');
                } else {
                    $location.path($scope.loginForm.returnUrl);
                }
            } else {
                $scope.loginForm.loginFailure = true;
            }
        });
    }
}

LoginController.$inject = ['$scope', '$routeParams', '$location', 'LoginFactory'];
```

We also need a minor update to our Login view:

```
<form ng-submit="login()">
    <label for="emailAddress">Email Address:</label>
    <input type="email" ng-model="loginForm.emailAddress" id="emailAddress"/>

    <label for="password">Password:</label>
    <input type="password" id="password" ng-model="loginForm.password"/>

    <label for="rememberMe">Remember Me:</label>
    <input type="checkbox" id="rememberMe" ng-model="loginForm.rememberMe" />

    <button type="submit">Login</button>
</form>

<div ng-if="loginForm.loginFailure">
    D'oh!
</div>
```

If all is right with the world, attempting to login should yield the following result:



That's because we don't have any users. Let's create our Registration factory. Add **RegistrationFactory.js** to **Scripts** => **Factories** and add the following Javascript:

```
var RegistrationFactory = function ($http, $q) {
    return function (emailAddress, password, confirmPassword) {

        var deferredObject = $q.defer();

        $http.post(
            '/Account/Register', {
                Email: emailAddress,
                Password: password,
                ConfirmPassword: confirmPassword
            }
        ).
        success(function (data) {
            if (data == "True") {
                deferredObject.resolve({ success: true });
            } else {
                deferredObject.resolve({ success: false });
            }
        }).
        error(function () {
            deferredObject.resolve({ success: false });
        });

        return deferredObject.promise;
    }
}

RegistrationFactory.$inject = ['$http', '$q'];
```

Now tell Angular about the new factory:

```
AwesomeAngularMVCApp.factory('RegistrationFactory', RegistrationFactory);
```

Then inject it into RegisterController, and call it's function:

```
var RegisterController = function ($scope, $location, RegistrationFactory) {
    $scope.registerForm = {
        emailAddress: '',
        password: '',
        confirmPassword: '',
        registrationFailure: false
    };

    $scope.register = function () {
        var result = RegistrationFactory($scope.registerForm.emailAddress, $scope.registerForm.password,
$scope.registerForm.confirmPassword);
        result.then(function (result) {
            if (result.success) {
                $location.path('/routeOne');
            } else {
                $scope.registerForm.registrationFailure = true;
            }
        });
    }
}

RegisterController.$inject = ['$scope', '$location', 'RegistrationFactory'];
```

Right now, we should be able to run our application, register as a user and view 'Route Three'.

That's all for Part One

I have tried to cover quite a broad range of topics in a single article, and will be going more into depth in all of these aread in future articles, including:

- Refactoring the code written in part one
- Anti forgery tokens
- Directives
- Services
- Providers

- Angular UI Router - A more advanced router for Angular than ng-route
- Angular UI directive for bootstrap - Twitter Bootstrap without jQuery
- And much more

I love criticism, it makes me better so I look forward to your comments.

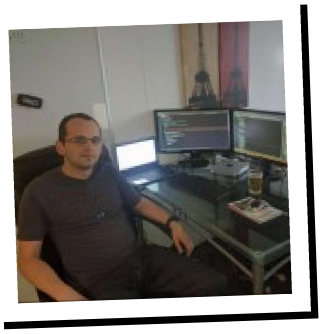
History

v 1.0 - 21:30GMT 2014-08-10

License

Share

About the Author



JMK-NI

Software Developer
United Kingdom

26 years old, from Belfast, lunatic, love what I do

Follow me on Twitter @chancymajormusk

You may also be interested in...

- 10 Ways to Boost COBOL Application Development
- SAPrefs - Netscape-like Preferences Dialog
- Flashing the Zephyr Application Using a JTAG Adapter on the Arduino 101 (branded Genuino 101 outside the U.S.) on Ubuntu under VMware
- Generate and add keyword variations using AdWords API
- Sony's N – The Smart Wireless Headset that You Can Program
- Window Tabs (WndTabs) Add-In for DevStudio

Comments and Discussions

166 messages have been posted for this article Visit **https://www.codeproject.com/Articles/806029/Getting-started-with-AngularJS-and-ASP-NET-MVC-Par** to post and view comments on this article, or click **here** to get a print view with messages.