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

JMK-NI, 17 Nov 2014 CPOL

4.90 (96 votes)

How to hit the ground running with AngularJS on ASP.NET MVC: Part 2

Articles in series:

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

In Part One, we got the ball rolling with a basic AngularJS/ASP.NET MVC application containing the following:

- The boilerplate code needed to build a fully fledged Single Page Application
- Some basic routing, including a route containing parameters and another which is only accessible to authenticated users (using cookie based authentication)
- Login and register forms, including the angular Javascript code needed to make these functional

For the lazy/time constrained, I have attached the Visual Studio solution as it was at the end of part one to this article. However, I have removed the packages directory to make this smaller, so you will need to use Nuget package restore to get them back.

Our Awesome application still has a lot of problems, some of which we are going to solve in part two. This article is shorter (and less exciting) than part one, but will get everything in place for part three.

Source Code

The source code accompanying this article can be found here.

Problems we are going to solve in part two

- Ugly URL's - As we navigate between views, we get this ugly scotch (#) in our URL's. We will fix this in part two by using HTML5 mode (pushstate)
- Ugly views - We didn't even try to apply any styling to our application in part one, let's add Twitter Bootstrap and Angular UI Directives for Bootstrap for added functionality, without the need for jQuery
- Very basic routing - In part one we used ngRoute, which is fine for basic routing, but falls short when we have more advanced requirements. In part two, we will replace this with Angular UI Router

Ugly URL's

Right now our URL's look like this:

Everything after the scotch (#) is ignored by the web server. ngRoute, which we added in part one, currently takes this part of the URL, checks to see if it matches any of the patterns we setup, and if so loads the correct view into the container div on our landing page.

We can have nicer URL's if we instead use AngularJS' HTML5 mode. This will cause Angular to use the HTML5 history API, handling all of that complexity, with just a couple of lines of code on our end (a single line of Javascript, and a couple of lines of C#).

Javascript first, we need to modify our apps config function, it now has Angular's **$locationProvider** module as a dependency, and we call it's **hashPrefix** and **html5mode** functions. That's it.

```
var configFunction = function ($routeProvider, $httpProvider, $locationProvider) {

    $locationProvider.hashPrefix('!').html5Mode(true);

    $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
        });
```

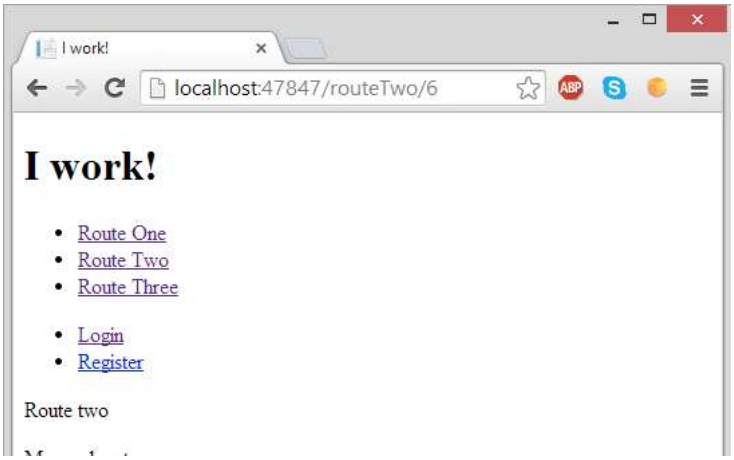
```
    $httpProvider.interceptors.push('AuthHttpResponseInterceptor');
}
configFunction.$inject = ['$routeProvider', '$httpProvider', '$locationProvider'];
```

The links on our landing page need to be updated as well to remove the scotch:

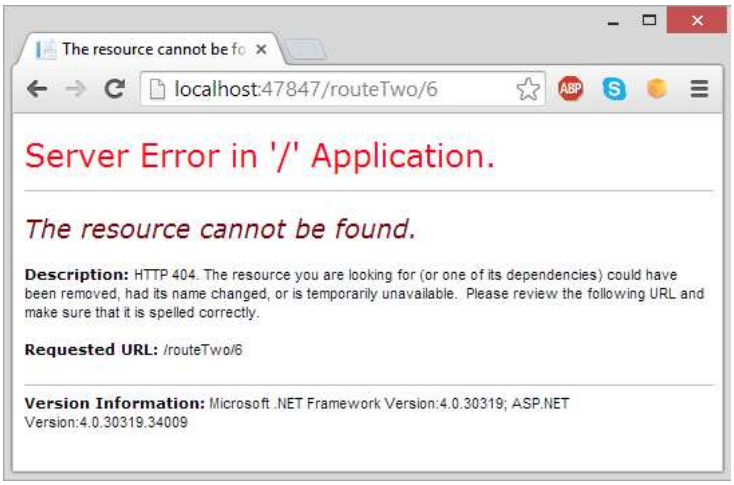
```
<ul>
    <li><a href="/routeOne">Route One</a></li>
    <li><a href="/routeTwo/6">Route Two</a></li>
    <li><a href="/routeThree">Route Three</a></li>
</ul>

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

Great, now let's debug the site and have a browse:



Our URL's look better, but hit refresh:



HTML5 mode is working, but only in a very superficial way. A refresh of the page is sending the full URL to the server (as we have removed the scotch) which doesn't know what to do. We can fix this by reconfiguring MVC's RouteCollection properly. We need to be explicit about the route for each of our views, and then add a catch-all which sends all other URL's to our landing page, to be handled by Angular.

Update the RegisterRoutes method inside **App_Start** => **RouteConfig.cs** like so:

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "routeOne",
        url: "routesDemo/One",
        defaults: new { controller = "RoutesDemo", action = "One" });

    routes.MapRoute(
        name: "routeTwo",
        url: "routesDemo/Two/{donuts}",
        defaults: new { controller = "RoutesDemo", action = "Two", donuts = UrlParameter.Optional });

    routes.MapRoute(
        name: "routeThree",
        url: "routesDemo/Three",
        defaults: new { controller = "RoutesDemo", action = "Three" });

    routes.MapRoute(
        name: "login",
        url: "Account/Login",
        defaults: new { controller = "Account", action = "Login" });
```

```
    routes.MapRoute(
        name: "register",
        url: "Account/Register",
        defaults: new { controller = "Account", action = "Register" });

    routes.MapRoute(
        name: "Default",
        url: "{*url}",
        defaults: new { controller = "Home", action = "Index" });
}
```

Debug the site again, browse around, and test the back/refresh buttons, everything should be working properly now.

Ugly views

Right now out site has no styling at all applied. Let's fix this with Twitter Bootstrap, which I will add from Cloudflare, in the <head> section of our landing page, along with Angular UI Directives for Bootstrap, which I will add just before the closing </body> tag.

By simply applying a couple of CSS classes, and adding a few elements, we can transform the appearance of our entire web application. Update the your landing page like so:

```
<!DOCTYPE html>
<html ng-app="AwesomeAngularMVCApp" ng-controller="LandingPageController">
<head>
    <title ng-bind="models.helloAngular"></title>
    <link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/3.2.0/css/bootstrap.min.css">
    @Styles.Render("~/Content/css")
</head>
    <body>
        <div class="navbar navbar-default navbar-fixed-top" role="navigation">
            <div class="container">
                <div class="navbar-header">
                    <button type="button" class="navbar-toggle" ng-click="navbarProperties.isCollapsed = !navbarProperties.isCollapsed">
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                    </button>
                    <a class="navbar-brand" href="#">Awesome Angular MVC APP</a>
                </div>
                <div class="navbar-collapse collapse" collapse="navbarProperties.isCollapsed">
                    <ul class="nav navbar-nav">
                        <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 class="nav navbar-nav navbar-right">
                        <li><a href="/login">Login</a></li>
                        <li><a href="/register">Register</a></li>
                    </ul>
                </div>
            </div>
        </div>

        <div class="container mainContent">
            <div ng-view></div>
        </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>
        <script src="//cdnjs.cloudflare.com/ajax/libs/angular-ui-bootstrap/0.10.0/ui-bootstrap-tpls.min.js"></script>
        @Scripts.Render("~/bundles/AwesomeAngularMVCApp")
    </body>
</html>
```

Add the following to **Content** => **Site.css**

```
.mainContent {
    margin-top: 60px;
}
```

Now, we need to register the Angular UI Directives for Bootstrap module with our Angular Application module, we do this in **AwesomeAngularMVCApp.js**:

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

Finally, our Angular Landing Page controller needs updated with the default state of the mobile navigation menu (which will initially be collapsed):

```
var LandingPageController = function($scope) {

    ...

    $scope.navbarProperties = {
        isCollapsed: true
    };
}
```

Twitter Bootstrap is an entire topic by itself. The point of this section was to show you how to properly add it to any AngularJS application.

Setting the mobile navigation menu up to expand and collapse properly, without the use of jQuery, is something that throws a lot of people, as does registering Angular UI Directives for Bootstrap with your Application module properly, so I made a point of covering these here.

You can now build on this to fully style your site with Twitter Bootstrap.

Very basic routing

We are currently using AngularJS' own ngRoute module, which is fine for basic routing, but we might find outselves a bit constrained if we have more advanced routing requirements.

Let's replace this with Angular UI Router. This is a fully fledged routing framework for Angular which provides us with such awesomeness as nested, multiple and named views.

We will do enough now to get up and running, but if you plan on using this to build a full single page application, you will need to read the In Depth Guide, and keep the API reference in a nearby tab to refer to.

Update your landing page, replacing the Javascript tag which adds ngRoute with one for Angular UI Router instead:

```
<script src="//cdnjs.cloudflare.com/ajax/libs/angular-ui-router/0.2.10/angular-ui-router.min.js"></script
```

Now we need to register Angular UI Router with our Angular Application module, removing the registration for ngRoute:

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

Angular UI Router is state based. It is based on the mathemetical concept of the Finite State Machine, and transforms your web application into the same. Instead of navigating from URL to URL, you transition from state to state, and setup a route to represent every state that your application can be in.

With UI Router, we can have multiple container views on our landing page, whereas with ngRoute we could only have one. We will now add two views to our landing page and setup our application to have four states:

- **When the application is in State One, we will load route one into our first container div, and route two into our second**
- **When the application is in State Two, we will load route one into our first container div, and route three into our second**
- **When the application is in State Three, we will load route two into our first container div, and route three into our second**
- **When the application is in the LoginRegister state, we will load our login form into our first container div, and the register form into the second**

Update the landing page so we now have two container divs, remove the container div we were using previously with ngRoute:

```
<div class="container mainContent">
    <div class="row">
        <div class="col-md-6">
            <div ui-view="containerOne"></div>
        </div>
        <div class="col-md-6">
            <div ui-view="containerTwo"></div>
        </div>
    </div>
</div>
```

Now let's modify AwesomeAngularMVCApp.js to tell it what views to place where, in which states. We no longer have a dependency on ngRoute's $routeProvider service, and instead have a new dependency on UI Router's $stateProvider:

```
var configFunction = function ($stateProvider, $httpProvider, $locationProvider) {

    $locationProvider.hashPrefix('!').html5Mode(true);

    $stateProvider
        .state('stateOne', {
            url: '/stateOne?donuts',
            views: {
                "containerOne": {
                    templateUrl: '/routesDemo/one'
                },
                "containerTwo": {
                    templateUrl: function (params) { return '/routesDemo/two?donuts=' + params.donuts; }
                }
            }
        })
        .state('stateTwo', {
            url: '/stateTwo',
            views: {
                "containerOne": {
                    templateUrl: '/routesDemo/one'
                },
                "containerTwo": {
                    templateUrl: '/routesDemo/three'
                }
            }
        })
        .state('stateThree', {
            url: '/stateThree?donuts',
            views: {
                "containerOne": {
                    templateUrl: function (params) { return '/routesDemo/two?donuts=' + params.donuts; }
                },
                "containerTwo": {
                    templateUrl: '/routesDemo/three'
                }
            }
        })
        .state('loginRegister', {
            url: '/loginRegister?returnUrl',
            views: {
                "containerOne": {
                    templateUrl: '/Account/Login',
                    controller: LoginController
                },
                "containerTwo": {
                    templateUrl: '/Account/Register',
                    controller: RegisterController
                }
            }
        });

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

We also need to update our AuthHttpResponseInterceptor to go to the loginRegister state whenever a 401 response is returned from the server. To achieve this, we will need to inject UI Router's $state service. However, due to a bug in this library, we can't inject this directly. Instead we inject AngularJS' $injector service and use this to resolve an instance of $state:

```
var AuthHttpResponseInterceptor = function($q, $location, $injector) {
    return {
        response: function (response) {
            if (response.status === 401) {
                console.log("Response 401");
            }
            return response || $q.when(response);
        },
        responseError: function (rejection) {
            if (rejection.status === 401) {
                $injector.get('$state').go('loginRegister', { returnUrl: $location.path() });
            }
            return $q.reject(rejection);
        }
    }
}

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

Our LoginController also needs updated, as it now pulls the return URL from UI Router's $stateParams object, as opposed to ngRoute's $routeParams object:

```
var LoginController = function ($scope, $stateParams, $location, LoginFactory) {
    $scope.loginForm = {
        ...etc
        returnUrl: $stateParams.returnUrl,
        ...etc
    };

    ...etc
}

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

Finally, we need to update our links. What's interesting is that our hyperlinks no longer center around the URL, now we link directly to the state itself, and provide any parameters for that state using JSON. Update the links on your landing page to look like this:

```
<div class="navbar-collapse collapse" collapse="navbarProperties.isCollapsed">
    <ul class="nav navbar-nav">
        <li><a ui-sref="stateOne({ donuts: 12 })">State One</a></li>
        <li><a ui-sref="stateTwo">State Two</a></li>
        <li><a ui-sref="stateThree({ donuts: 4 })">State Three</a></li>
    </ul>
    <ul class="nav navbar-nav navbar-right">
        <li><a ui-sref="loginRegister">Login / Register</a></li>
    </ul>
</div>
```

Now let's it debug and have a browse. Navigating to State One should return routes one and two, side by side:

If we try and navigate to States two or Three, our interceptor will kick in as before, and transition us to our loginRegister state. If we login at this point, we will be able to view states two and three also.

Nested Views

One thing we haven't covered yet is nested views. Let's add another C# action method to our RoutesDemo controller called Four, and use Visual Studio to create the view. Add some content to this view to uniquely identify it. Also remove the Authorize attribute from Route Three for now, this was only there to demonstrate a concept.

We are going to nest route four inside route one, so update the view for route one like so:

```
Route one

<div ui-view="nestedView"></div>
```

Now let's update our routing configuration in Angular to reflect this. When we are adding a nested view to a state, we using the naming configuration viewName@stateName. So to configure nestedView for stateOne we give it the name nestedView@stateOne, like so:

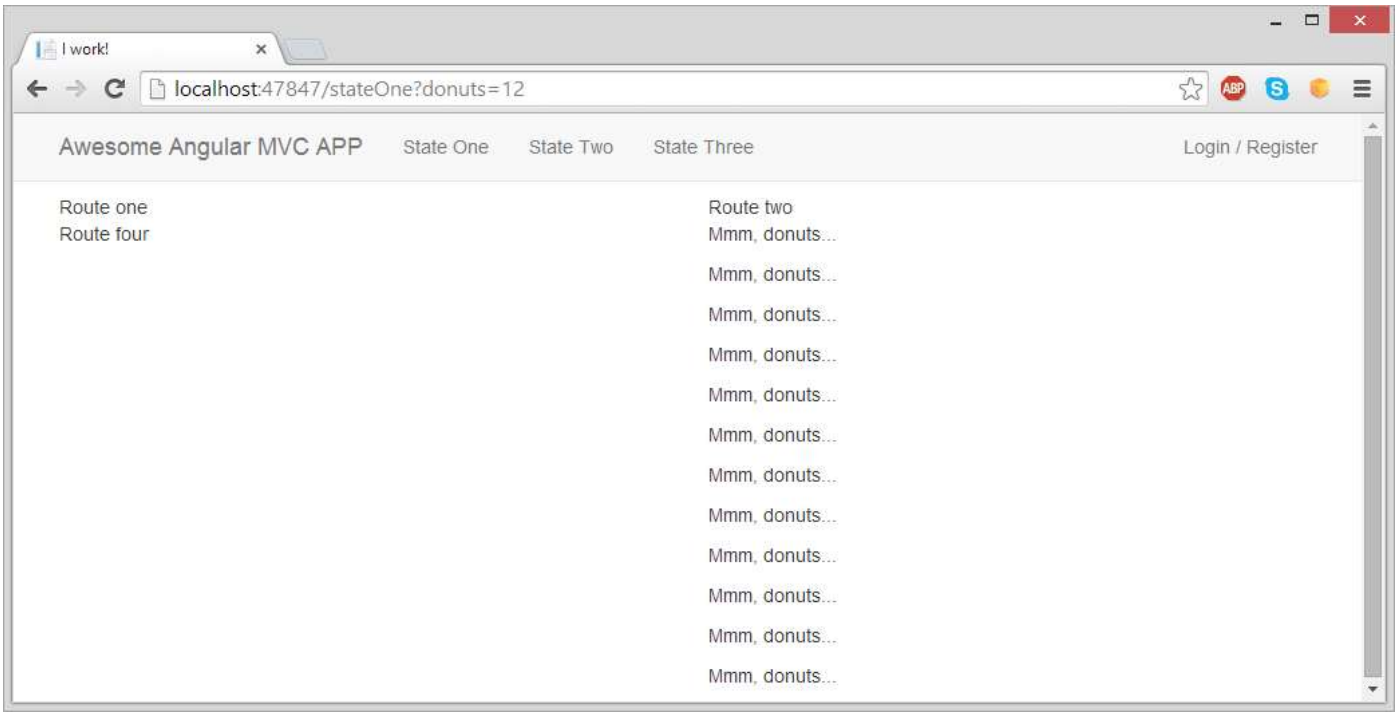
```
    $stateProvider
        .state('stateOne', {
            url: '/stateOne?donuts',
            views: {
                "containerOne": {
                    templateUrl: '/routesDemo/one'
                },
                "containerTwo": {
                    templateUrl: function (params) { return '/routesDemo/two?donuts=' + params.donuts; }
                },
```

```
                "nestedView@stateOne": {
                    templateUrl: '/routesDemo/four'
                }
            }
        })
        .state('stateTwo',
        ...etc
```

We have just added a new view, so we need to update RouteConfig.cs to reflect this:

```
routes.MapRoute(
    name: "routeFour",
    url: "routesDemo/Four",
    defaults: new { controller = "RoutesDemo", action = "Four" });
```

Now let's test it out:



Recap

Ok so in part two we achieved the following:

- **Enabled HTML5 mode (pushstate) in AngularJS and configured MVC as neccessary to make this work**
- **Added Twitter Bootstrap, and Angular UI directives for bootstrap. Everything is now in place for us to add components from either of those libraries as we need to moving forward**
- **Replaced ngRoute with Angular UI router, and added named, multiple and nested views to our application.**

Coming in Part Three

- **SignalR integration**
- **Directives**
- **Anti forgery tokens**

Comments/Criticism/Questions etc

Feel free to comment on the article with any comments/criticisms/questions etc and I will always reply. Thanks for reading :)

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...

IoT JumpWay Intel® Edison Basic LED Example

SAPrefs - Netscape-like Preferences Dialog

Visual COBOL New Release: Small point. Big deal

Generate and add keyword variations using AdWords API

10 Ways to Boost COBOL Application Development

Window Tabs (WndTabs) Add-In for DevStudio

Comments and Discussions

62 messages have been posted for this article Visit https://www.codeproject.com/Articles/806500/Getting-started-with-AngularJS-and-ASP-NET-MVC-P to post and view comments on this article, or click here to get a print view with messages.