Introduction to Angular Routing

To build a Single-Page Application (SPA) using Angular (v1.x), you typically build a single HTML page and inject HTML fragments within this one page as the user navigates within your application. Navigation in Angular employs a mechanism called routing. This blog post explores how to perform navigation within a SPA using Angular routing.

Server-Side Development

As you make the transition from server-side development to client-side development, you will find many of the same concepts that you employ on the server-side have equivalents on the client-side. Of course, they are done differently, but the concepts are there nonetheless.

When developing server-side web applications with MVC or Web Forms, you use a common layout page for all of the standard “chrome” you want around your content. This chrome is your header, footer and maybe a sidebar. The header consists of a menu system and maybe some graphics. The footer might have a copyright and some additional links. You do not want to duplicate the header and footer on each page in your application as that is a maintenance nightmare. Instead, each MVC or Web Forms application has a special “layout” page where you create the chrome. You create your content pages with the HTML fragments you wish to display to the user, and inject those fragments within the layout page.

MVC has a special page named _Layout.cshtml located under the \Shared folder (Figure 1). This page has a piece of Razor code called @RenderBody() which tells MVC into where to inject your fragments of HTML and Razor code. Web Forms uses a concept called a “Master Page”, shown in Figure 2 as Site.Master, which uses a <asp:ContentPlaceHolder /> control to specify where to inject your content pages.

Both these approaches keep the chrome for your web application in a single location. This makes changes to your website easy to accomplish. When creating a SPA using Angular (or any other client-side framework), you should strive to use this same technique.

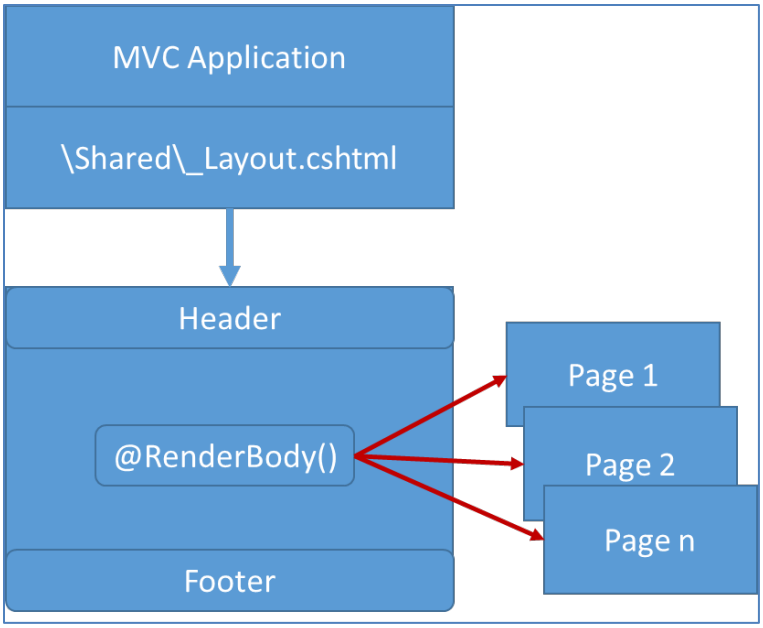

Figure 1: MVC uses a shared layout page

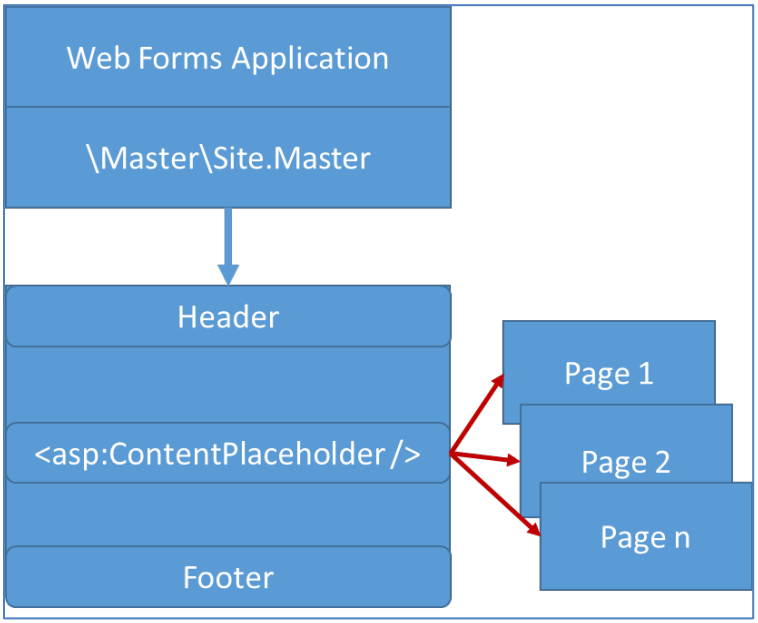

Figure 2: Web Forms uses a master page

Angular ng-view Directive

Angular has the same mechanism for defining a HTML page with the chrome, and a directive for specifying where to inject the HTML fragments that make up each content page. You typically create an index.html page with the chrome and a single tag that uses the Angular directive **ng-view** (Figure 3). This directive is what is used to specify the location in which to inject the content pages.

It is important to note that only one instance of ng-view may be used in your Angular application. In other words, you cannot nest an ng-view within another ng-view. If you are using this approach correctly, you shouldn't have to nest ng-view anyway.

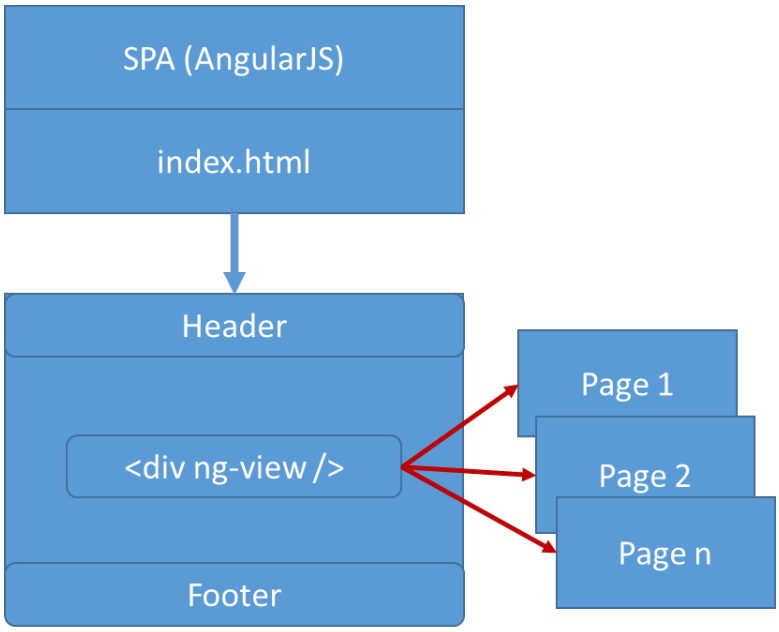

Figure 3: Angular uses any html page for the chrome

Download Routing JavaScript File

In order to use Angular routing, download the angular-route.js file into your project. If you are using Visual Studio, you may use the NuGet Package Manager to search for and install the AngularJS.Route package. Or, open a browser and navigate to https://code.angularjs.org/1.5.8/ and download the angular.route.js file from there. Either way, once you download this script file

you will want to reference it from your main HTML page. Be sure to place it after you have referenced the angular.js file as shown in the following code snippet.

```
<script src="scripts/angular.js"></script>
<script src="scripts/angular-route.js"></script>
```

Declare Your Intention to use Routing

The first step in any Angular application is to define a module that is the main entry point for your application. You typically define a module using the following code.

```
(function () {
  'use strict';

  angular.module('app', []);
})();
```

As you are going to be using routing in your Angular application, this is now a dependency that you need to tell Angular about. The second parameter to the module() is an array of strings for you to specify the names of any dependencies needed for your application. In the code snippet below you are passing in a single element array with the value being ‘ngRoute’. The ‘ngRoute’ value is defined as a provider in the angular-route.js file you downloaded and included in your project.

```
(function () {
  'use strict';

  angular.module('app', ['ngRoute']);
})();
```

The HTML Page

The complete HTML page, index.html, is shown in Listing 1. This page has a couple of anchor tags <a> that are used for our routing sample. In addition the <div ng-view> element is also defined within a Bootstrap row and column. This is where all HTML fragments will be displayed when you route to a new path.

```
<!doctype html>
<html>
<head>
  <title>Routing Sample</title>

  <link href="Content/bootstrap.min.css"
        rel="stylesheet" />
</head>
<body>
  <div ng-app="app"
       ng-controller="IndexController as vm"
       class="container">

    <div class="row">
      <div class="col-sm-12">
        <a href="#/page1"
           class="btn btn-primary">Page 1</a>
        <a href="#/page2"
           class="btn btn-primary">Page 2</a>
      </div>
    </div>

    <br />

    <div class="row">
      <div class="col-sm-12">
        <div ng-view></div>
      </div>
    </div>
  </div>

  <script src="scripts/angular.js">
  </script>
  <script src="scripts/angular-route.js">
  </script>

  <script src="app.module.js"></script>
  <script src="index.controller.js">
  </script>
</body>
</html>
```

Listing 1: The HTML page for our routing sample

The index.controller.js file that is referenced in the above web page is an empty function as there is no functionality needed for this sample web page. The contents of the index.controller.js file is shown below just for completeness.

```
(function () {
  'use strict';

  angular.module('app')
   .controller('IndexController',
               IndexController);

  function IndexController() {
  }
})();
```

Define Your Routes

After you have told Angular that you are using routing, it is now time to create some routes. Within the app.module.js file (or create another file called index.route.js) add the code shown in Listing 2.

```
angular.module('app')
.config(function ($routeProvider) {
  $routeProvider
  .when('/',
  {
    template: ''
  })
  .when('/page1',
  {
    template: '<p>This is some text for Page1</p>'
  })
  .when('/page2',
  {
    template: '<h2>Page 2</h2>'
  });
});
```

Listing 2: Define routes to your Angular application.

The code in Listing 2 retrieves the module named ‘app’ from Angular. It then chains the config() function to instantiate the routes for the module. Pass in your custom function to the config() function. This function is passed the

$routeProvider provider, which is defined in the angular-route.js file. You use this $routeProvider variable to define the routes you wish to configure using the when() function. Each when() function is passed two parameters. The first parameter is a string that is matched up with the path you define in your web page. For example, **<a href="#/page1" …/>** matches up with **when('/page1')**. The # symbol is used so the browser does not try to navigate to a page. Angular looks for anything that starts with "#/" and knows that you are using a route. Figure 4 shows the index.html page and the code in the $routeProvider definition and how they match up.

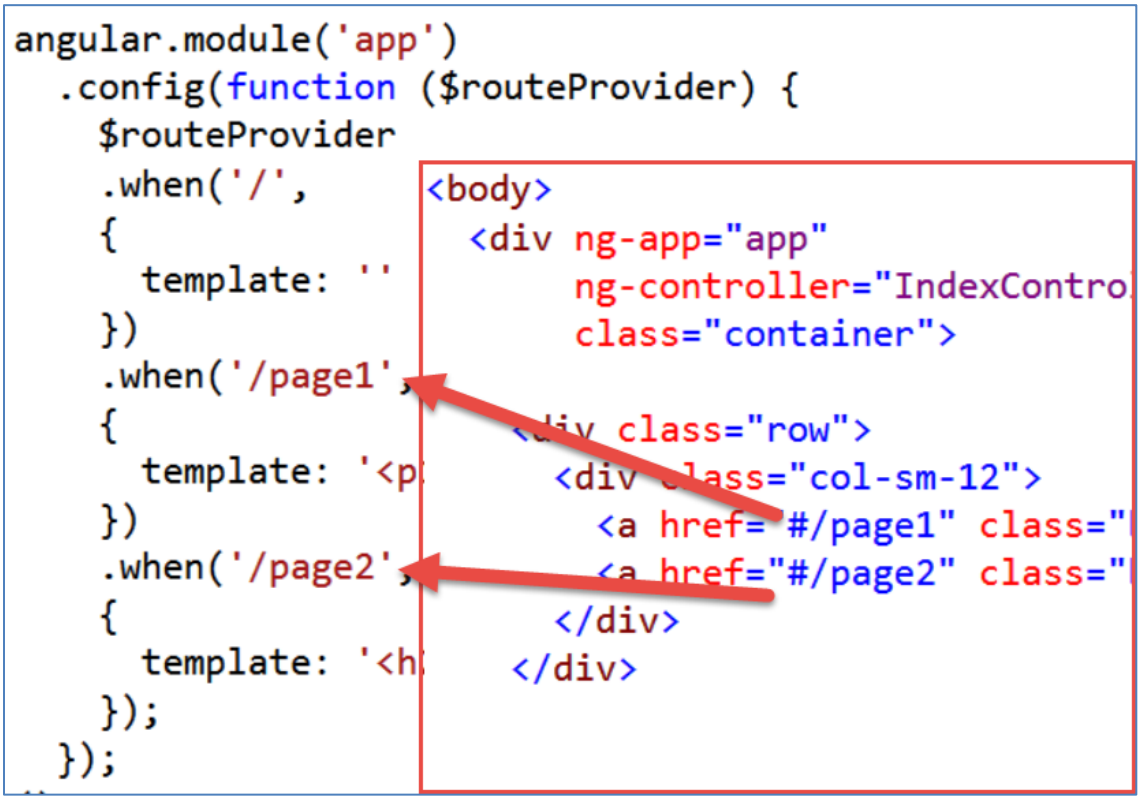

Figure 4: Use a # symbol to specify a route

The second parameter to the when() function is an Angular route object. This object has several properties that can be set. For this initial sample the **template** property is set. The **template** property lets you define any HTML code you wish to display in the ng-view directive when this route is invoked. In Figure 5 and Figure 6 you can see both the final web page and the corresponding code in the $routeProvider which caused that HTML to be displayed.

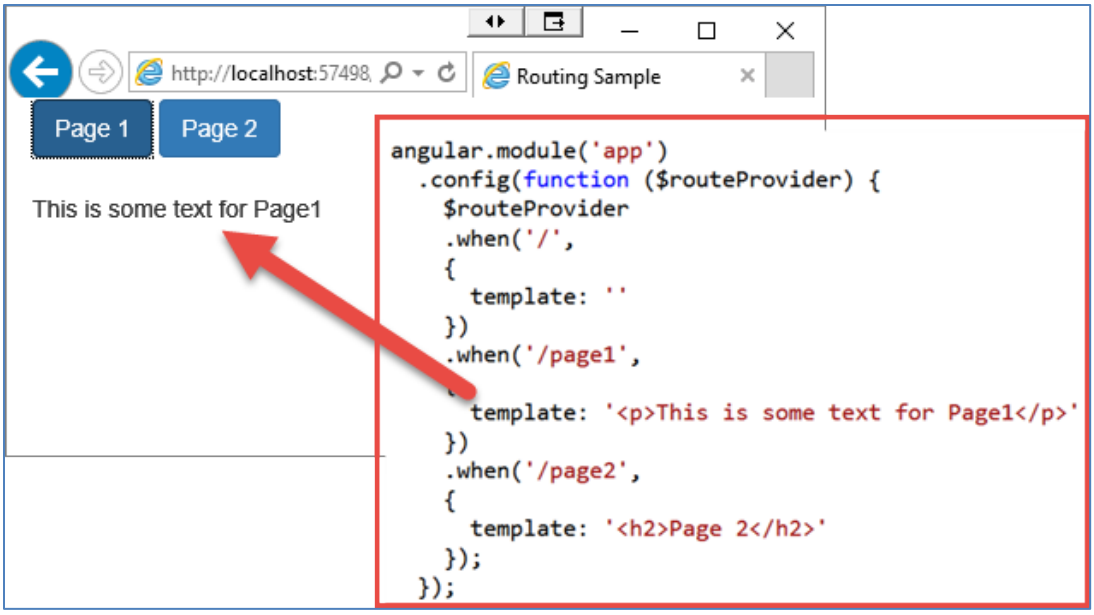

Figure 5: Page 1 shows some standard text

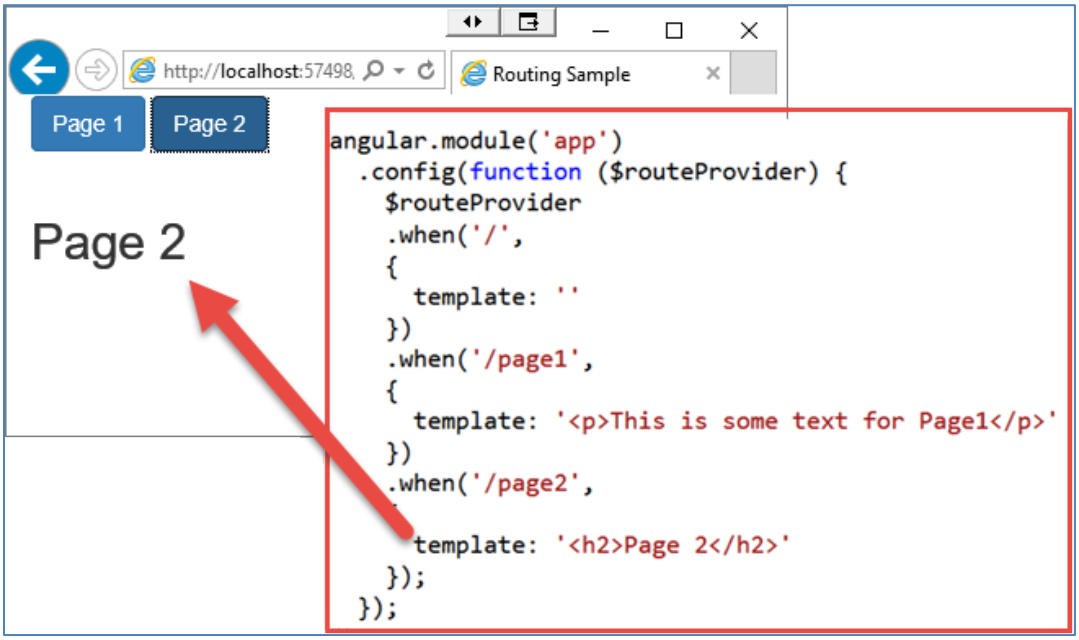

Figure 6: Page 2 shows some bold text

Handling a Bad Link

If you have a link in your HTML page that does not have a corresponding when() function configured, you should display some error text to your user.

After the last when() function is called, add an **otherwise**() function as shown in the following code snippet.

```
.otherwise(
{
  template: '<h2>Bad Link!</h2>'
});
```

Avoid Hard-Coding HTML

The problem with the previous example is you hard-coded some HTML within your JavaScript. It is a best practice to keep all your HTML in .html files in your project. Instead of defining your routes using the **template** property, use the **templateUrl** property instead as shown in Listing 3. The **templateUrl** property must be set with the full path in relation to the index.html page. In this sample, all the .html pages are in the same folder. However, you might need to specify something like the following: **templateUrl: 'app/templates/page1.template.html'**.

```
angular.module('app')
.config(function ($routeProvider) {
  $routeProvider
  .when('/',
  {
    template: ''
  })
  .when('/page1',
  {
    templateUrl: 'page1.template.html'
  })
  .when('/page2',
  {
    templateUrl: 'page2.template.html'
  })
  .when('/error',
  {
    templateUrl: 'badlink.template.html'
  })
  .otherwise(
  {
    redirectTo: '/error'
  });
});
```

Listing 3: Use the templateUrl property to keep HTML code in .html files

In Listing 3 another new property was introduced in the otherwise() function, **redirectTo**. The **redirectTo** property allows you to redirect to another route. This property is most typically used in the otherwise() function. In the sample in Listing 3 if you attempted to go to a href such as the one shown in the following code snippet, you would be redirected to the path **/error**, which would then display the HTML in the badlink.template.html file.

```
<a href="#/badLink"
   class="btn btn-primary">
     Bad Link
</a>
```

Another option for the otherwise() function is to simply specify a string as the first parameter. If a string is specified instead of an object, it interprets the string as a redirect to.

```
.otherwise('/error')
```

Summary

In this blog post you learned the basics of routing in Angular. Just like server-side web development, the concept of having a single place for all your chrome is also present client-side. The ng-view directive is used to specify where in your HTML page you wish to display other HTML pages. You will need to download angular-route.js in order to use routing in your Angular web pages.

Sample Code

You can download the code for this sample at www.pdsa.com/downloads. Choose the category “PDSA Blog”, then locate the sample **PDSA Blog Sample: Introduction to Angular Routing**.