Handling HTML5 Client Route Fallbacks in ASP.NET Core

.NET Core SDK

When using client side applications built with application frameworks like **Angular**, **React**, **Vue** and so on, you will invariably deal with HTML5 client side routing, which handles client side routes to pages and components entirely in the browser. Well almost entirely in the browser...

HTML5 client routes work great on the client, but when deep linking into a site or pressing Refresh in the browser, client side routes have a nasty habit of turning into server HTTP requests. Requests to routes that the server is likely not configured for.

In this post I discuss what you have to do to have your ASP.NET Core (or indirectly ASP.NET application) handle these 'fake' requests by effectively re-attaching the client side application to its routes.

Html 5 Client Side Routing?

If you don't know what HTML5 client side routing is, here's a quick refresher.

Client side frameworks implement their own client side routing mechanisms so they can - just like server applications - navigate from page to page or component.

Angular supports a couple of route types:

- **Hash bang routes**
  http://localhost:4200/#!/albums or **http://localhost:4200/#/albums**)
- **HTML 5 Routes**
  http://localhost:4200/albums

Hash Bang Routes

The former is an older approach that works directly with the HTTP semantics that specify that any URL with a `#` is fired on the client and jumping to a 'local' URL within the page. Frameworks can intercept the navigation and examine the URL content following the `#` to determine the route. The hash bang `#!\` is used to differentiate application urls from plain `#` anchor links.

The nice thing about hash bang routes is that they **just work**. There's no server side bleeding of the routes and if you bookmark or refresh the client side page it just works as expected because the hash logic is executed as part of the native URL parsing in the browser. Easy, right? **It just works**.

But the downside is that the URLs are pretty ugly and non-intuitive if you have ever have to type it in manually. Not really a great argument against hash bang routes, but regardless they are falling out of favor to HTML5 routing.

Hash Bang Routes in Angular

Angular uses HTML5 Client routes by default, but it's a simple switch to enable Hashbang routes instead of HTML5 routes::

```
// in app.module.ts
providers   : [
      ..
      // make sure you use this for Hash Urls rather than HTML 5 routing
      { provide: LocationStrategy, useClass: HashLocationStrategy },
]
```

As long as you use `routerLink` for your link URLs in HTML templates, and `router.navigate()` for in-code links, Angular switches automatically between the two modes.

- In HTML use `<a routerLink="/albums" />` links
- In code use: `router.navigate(["/album",album.id])`

##AD##

HTML5 Routing

HTML5 routing uses a more sophisticated approach - it uses HTML5's **Pushstate API** to control the routes on the client side and manage the address bar display.

The advantage of this approach is that is easily controllable with this relatively easy to work with HTML5 API and **the URLs are much cleaner**, using standard extensionless route conventions you expect from Web applications and APIs today.

But **HTML5 routes require explicit support from the server** to appropriately understand which routes are server routes and which are client routes.

HTML5 Route Problems without Server Handling

The problem is that HTML5 client routes are indistinguishable from server routes.

`http://localhost:4200/albums` can just as easily be a client side URL as a server side URL. While navigating entirely on the client, HTML5 routes work fine - the application can intercept navigation and route to the appropriate client side pages when a specific route is activated.

The problem pops up if you have navigated into a client driven application with a deep link and you then bookmark the page and later navigate back to it with that URL, or you refresh the current active page. In both cases the client application isn't running when the route is requested by the browser and so the browser requests the routed URL from the server. But by default isn't set up to handle say the `/albums` route and so you get an error.

If you don't have any special handling for HTML5 routes setup in your ASP.NET Core application you are going to hit an error page in your application or this default display from Kestrel:

Figure 1 - A unhandled client side route produces a server error

Fixing Client Side Routes on the Server

So how do you fix this?

Client side SPA applications typically have a single or perhaps a few static pages that launch the application. For a typical Angular application that page is `index.html` which launches the application and initiates the client side routing . Most frameworks are smart enough to check the current route as they start up and move to the route requested on the first access.

If a client side route gets fired to the server from a bookmark, link or full refresh, you'll want to serve up `index.html` and leave the original URL intact.

The client side application then bootstraps itself and the internal routing kicks in to hopefully whisk you back to the bookmarked/refresh location.

Serving Index.html from the Server

In order for this to work you need to ensure that the server only serves content the server is responsible for.

There are a couple of ways to do this:

- Host Server URL Rewriting
- Handling the client side routes in the ASP.NET Core app

URL Rewriting on the Host Web Server

If you're running your ASP.NET Core (or ASP.NET) application on a mainstream Web server the easiest and **most efficient solution** is to rewrite the client side URLs and serve `index.html` content for the given URL.

On IIS you can use the IIS Rewrite Module to do this. I covered this in more detail in a blog post recently:

- **IIS Rewrite Rules for ASP.NET Core applications**

But here is the relevant IIS Rewrite Rule:

```
<rewrite>
  <rules>
    <!--
        Make sure you have a <base href="/" /> tag to fix the root path
        or all relative links will break on rewrite
    -->
   <rule name="AngularJS-Html5-Routes" stopProcessing="true">
      <match url=".*" />
      <conditions logicalGrouping="MatchAll">
          <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
          <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
          <add input="{REQUEST_URI}" pattern="api/" negate="true" />
      </conditions>
      <action type="Rewrite" url="wwwroot/index.html"  />
    </rule>
  </rules>
</rewrite>
```

You can install the UrlRewrite module from any of these locations:

- **Microsoft Download Site**
- `choco install urlrewrite`
- **Web Platform Installer**

If you're running on Linux with Docker and nginX or Apache, similar Rewrite options are available there.

Letting ASP.NET Core Handle Client Side Routes

As mentioned I typically use a front end Web server like IIS or nginX to handle the redirects, but often while testing or for internal apps it's nice to just have Kestrel serve the application directly. If you directly let Kestrel handle the HTTP traffic then you need to handle the client routes in your ASP.NET Core code.

Catch All `app.Run()` Handler

There are a number of approaches available but I found the easiest way to handle client side routes with a very simple fallback handler in the `Startup` class' `Configure()` method:

```
// set up whatever routes you use with UseMvc()
// you may not need to set up any routes here
// if you only use attribute routes!
app.UseMvc(routes =>
{
    routes.MapRoute(
     name: "default",
     template: "{controller=Home}/{action=Index}/{id?}");
});

//handle client side routes
app.Run( async (context) =>
{
    context.Response.ContentType = "text/html";
    await context.Response.SendFileAsync(Path.Combine(env.WebRootPath,"index.html"));
});
```

The key bit is the `app.Run()` middleware handler which sits at the end of the pipeline after routing. If the server side routing can't find a matching route, this catch-all handler kicks in.

The code above is the simplest thing you can do which simply sends the content of `index.html` to the client. If you have multiple static pages with *SPA Silos* you can put additional logic in there to try and determine which page needs to be loaded.

Note that the content is not redirected to but rather sent as an inline stream to the existing URL request so that the URL the user requested stays intact. This ensures when the user requests `http://localhost:4200/albums` you go back to that client page and not `index.html`.

##AD##

Catch All Route Handler

Another way is to use a **catch-all** MVC Route Handler defined last in your routing definitions. This basically picks up any URLs your MVC routing configuration couldn't handle and then routes to a route you specify.

To set up your MVC routes with a catch-all handler put this code in your `Startup` class' `Configure()` method:

```
app.UseMvc(routes =>
{
    // default routes plus any other custom routes
 routes.MapRoute(
  name: "default",
  template: "{controller=Home}/{action=Index}/{id?}");


    // Catch all Route - catches anything not caught be other routes
 routes.MapRoute(
  name: "catch-all",
  template: "{*url}",
  defaults: new {controller = "AlbumViewerApi", action = "RedirectIndex"}
 );
});
```

The implementation then does the same thing the catch-all middleware handler used: Stream the content of `index.html` to the client using the following code:

```
// we need hosting environment for base path
public IHostingEnvironment HostingEnv { get; }

public AlbumViewerApiController(IHostingEnvironment env)
{
    HostingEnv = env;
}


[HttpGet]
public IActionResult RedirectIndex()
{
    return new PhysicalFileResult(
        Path.Combine(HostingEnv.WebRootPath,"index.html"),
        new MediaTypeHeaderValue("text/html")
    );
}
```

> **⚠ Don't use an Attribute Route for Catch-All Route**
>
> Make sure the route you specify for your fallback route doesn't also have an attribute route assigned to it. When checking this out yesterday I couldn't get the catch-all route to fire and it wasn't until I removed `[Route("api/RedirectIndex")]` from the controller action that the catch-all worked.

SpaServices

Another option is provided by **SpaServices** and using `routes.MapSpaFallbackRoute()` although I haven't tried this out myself, but if you're already using Spa Services with your ASP.NET Core app then this is likely an easy way to get this to work including potential support for server pre-rendering.

##AD##

Summary

HTML5 routing provides client side applications with clean URLs but it comes with the price of having to have server support to make it work. It's not difficult to set this up either with Rewrite Rules in the host Web server or directly in Kestrel's middleware pipeline or custom route handler, but you have to make sure to add this functionality explicitly to each ASP.NET application you create.

Even though the older Hash Bang routes aren't as clean looking, they work fine and don't require any server side support to work. For non-public facing applications or applications that have to support ancient browsers Hash Bang routes are still a viable way to provide routing without server support.

Finally if you are using a full Web server UrlRewriting is the cleanest and most efficient way to handle non API content that isn't directly handled by your ASP.NET Core backend.

Choice is good and you have a few options to choose from providing convenience, clean urls or simple just drop it in functionality. Your choice...
