**Building Web applications with Knockout.js and ASP.NET Core**

**Posted by:**[Francesco Abbruzzese](https://www.dotnetcurry.com/Author.aspx?AuthorName=Francesco%20Abbruzzese) , on 10/3/2017, in **Category**[ASP.NET Core](https://www.dotnetcurry.com/BrowseArticles.aspx?CatID=88)

**Views:**34870

**Abstract:**Integrate Knockout.js and ASP.NET Core to build a Single Page Application (SPA). Also learn to define components for Razor based Views and enhance Razor pages with client side bindings.

[608](https://facebook.com/sharer/sharer.php?u=https%3A%2F%2Fwww.dotnetcurry.com%2Faspnet%2F1394%2Faspnet-core-knockout" \t "_blank)

[24](https://twitter.com/intent/tweet/?text=Building%20Web%20applications%20with%20Knockout.js%20and%20ASP.NET%20Core%20%7C%20DotNetCurry&url=https%3A%2F%2Fwww.dotnetcurry.com%2Faspnet%2F1394%2Faspnet-core-knockout" \t "_blank)

[32](https://www.linkedin.com/shareArticle?mini=true&amp;url=https%3A%2F%2Fwww.dotnetcurry.com%2Faspnet%2F1394%2Faspnet-core-knockout&title=Building%20Web%20applications%20with%20Knockout.js%20and%20ASP.NET%20Core%20%7C%20DotNetCurry" \t "_blank)

Amongst all the client side frameworks backed by big companies, React.js and Angular.js appear to be the most popular. However, Knockout.js still maintains a good market share, thanks to its interesting peculiarities.

Knockout is based on an MVVM paradigm similar to Angular.js, but unlike React.js. While it is adequate for modular complex applications, at the same time, it is very simple to mix with server side templating, similar to React.js, but unlike Angular.js.

Are you keeping up with new developer technologies? Advance your IT career with our Free Developer magazines covering C#, Patterns, .NET Core, MVC, Azure, Angular, React, and more. [**Subscribe to the DotNetCurry (DNC) Magazine for FREE**](https://www.dotnetcurry.com/magazine/) and download all previous, current and upcoming editions.

For this reason, it appears to be better than both React and Angular for building modular complex systems that mix server side techniques (like Razor), with client side techniques.

**Integrating Knockout.js with ASP.NET Core**

In this “how to” article, I’ll show how to integrate Knockout and ASP.NET Core in several ways:

1) To build a Single Page Application that communicates with Asp.net core API controllers;

2) To define “components” for Razor based Views, like the ones we might define with React.js, but MVVM based;

3) To enhance Razor pages with client side bindings, something difficult to achieve with both Angular.js and React.js.

**ASP.NET Core Webpack based SPA templates**

Asp.net Core offers [Nuget packages](https://github.com/aspnet/javascriptservices" \t "_blank) that assists developers with client side technologies and can execute JavaScript Node.js code on the server side. Among them, [Microsoft.AspNetCore.SpaTemplates](https://www.nuget.org/packages/Microsoft.AspNetCore.SpaTemplates/" \t "_blank) installs spa templates for the more common client side frameworks: Angular 2, React.js, Knockout.js, and Aurelia.

All templates use [webpack 2](https://webpack.js.org/guides/getting-started/) to bundle and deploy all client resources (TypeScript and JavaScript files, html, images, and CSS). They can be installed with the dotnet core command:

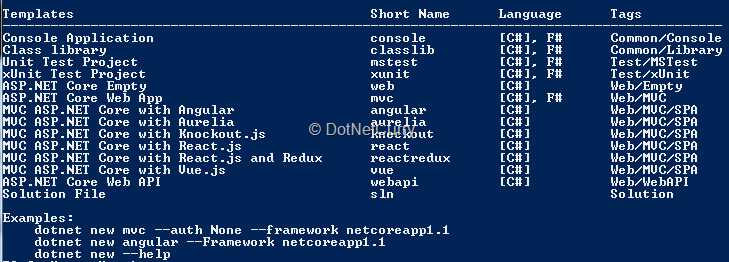
|  |
| --- |
| dotnet new --install Microsoft.AspNetCore.SpaTemplates::\* |

You may run this command in any Windows, Mac, and Linux console.

After the installation run:

|  |
| --- |
| dotnet new --help |

It should list all available project templates, as shown in the following image:



Here choose the template whose short name is “knockout”.

Create a folder called “KnockoutDemo” for our project. Then open a console in this directory. In Windows, we can do this by holding down “shift” while right clicking on the newly created folder, and then selecting "open command window here".

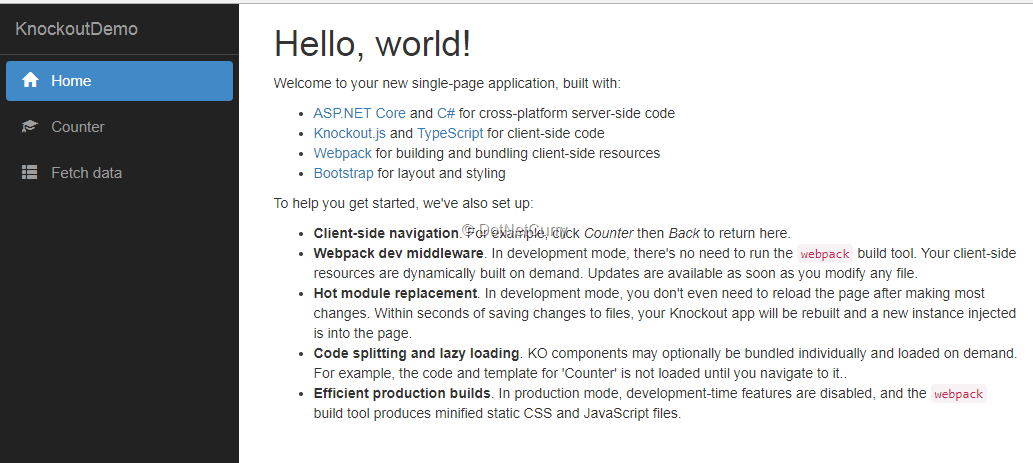
Once *KnockoutDemo* is the default folder in your console, in order to create a new knockout SPA project, type the following:

|  |
| --- |
| dotnet new knockout |

After that you may open the newly created KnockoutDemo.csproj in Visual Studio 2017.

Once all Nuget, and NPM packages have been restored, save the whole solution.

When you run the project, you should see something like this:



It is a Single Page Application with three pages. We will analyze it in the following subsection.

**Structure of a Knockout.js Single Page Application**

When you click a menu link, the content of the browser address bar changes, but the browser doesn’t perform any GET.

Link URLs are mapped into [knockout.js components](http://knockoutjs.com/documentation/component-overview.html) by the code in wwwroot\ClientApp\router.ts, that in turn uses [Crossroad.js](https://www.npmjs.com/package/crossroads) to handle routes (url-components mappings).

After that, knockout.js components are retrieved by the custom [knockout.js component loader](http://knockoutjs.com/documentation/component-loaders.html) contained in wwwroot\ClientApp\router.ts\webpack-component-loader.ts that in turn relies on webpack 2 loader to communicate with the server.

Browser history and address bar are handled by [History.js](https://www.npmjs.com/package/history).

The whole application is hosted by a single View, namely the view rendered by the Index action method of the Home controller:

|  |
| --- |
| public IActionResult Index()  {       return View();  }  @{      ViewData["Title"] = "Home Page";  }    <app-root params="history: history"></app-root>      @section scripts {      <script src="~/dist/main.js" asp-append-version="true"></script>  } |

The app-root component contains the “hole” that hosts the various SPA pages with the appropriate knockout.js bindings. We will analyze it in the next section.

main.js contains all application specific JavaScript packaged by webpack 2. The code of each spa component is loaded dynamically by webpack-component-loader.ts.

The layout page instead references the following:

1. the vendor.js file where webpack 2 bundles all JavaScript contained in the npm modules that are needed at runtime.

2. the vendor.css file where webpack 2 bundles all npm modules CSS that is needed at runtime

3. The site.css file with the site-specific CSS. Actually, site.css is added only in production and staging. This is done since webpack 2 is configured to add all CSS rules within a style tag during development, so that changes made while the program is running, takes immediate effect thanks to webpack 2 hot module replacement.

|  |
| --- |
| <!DOCTYPE html>  <html>      <head>          <meta charset="utf-8" />          <meta name="viewport" content="width=device-width, initial-scale=1.0" />          <title>@ViewData["Title"] - KnockoutDemo</title>            <link rel="stylesheet" href="~/dist/vendor.css" asp-append-version="true" />          <environment names="Staging,Production">              <link rel="stylesheet" href="~/dist/site.css" asp-append-version="true" />          </environment>      </head>      <body>          @RenderBody()            <script src="~/dist/vendor.js" asp-append-version="true"></script>          @RenderSection("scripts", required: false)      </body>  </html> |

**Bundling files with Webpack 2**

The way [webpack 2](https://webpack.js.org/guides/getting-started/) bundles all files is defined into the webpack.config.js and webpack.config.vendor.js configuration files.

webpack.config.vendor.js uses the [DllPlugin](https://github.com/webpack/docs/wiki/list-of-plugins" \l "dllplugin" \t "_blank) to bundle all needed npm modules as a library that is then referenced by webpack.config.js with the [DllReferencePlugin](https://github.com/webpack/docs/wiki/list-of-plugins" \l "dllreferenceplugin" \t "_blank) plugin.

webpack.config.js specifies how to bundle all js, ts, CSS and images that are specific to the project. It uses the [awesome-typescript-loader](https://github.com/s-panferov/awesome-typescript-loader) plugin to compile and load TypeScript according to the compilation configuration contained in tsconfig.json.

knockout.js components are bundled in separate files and loaded on demand, thanks to the lazy loading feature of the webpack 2 ['bundle-loader!'](https://github.com/webpack-contrib/bundle-loader) plugin.

Lazy loading is activated by prefixing the file name contained in a “require” with the 'bundle-loader?lazy!

When this is done, the call to “require” instead of returning the actual module returns a “load function”, that when called with a callback parameter, triggers the actual module load.

In our case, the actual “load function” will be invoked by the webpack-component-loader.ts custom knockout loader that we will look at in the next section.

The line:

|  |
| --- |
| test: /\.css$/, use: isDevBuild ? ['style-loader', 'css-loader']         : ExtractTextPlugin.extract({ use: 'css-loader?minimize' }) |

..in webpack.config.js specifies that CSS should be bundled in the html as *in-line* style during development, and as a *unique minimized file*, in all other environments.

The line:

|  |
| --- |
| test: /\.(png|jpg|jpeg|gif|svg)$/, use: 'url-loader?limit=25000' |

..causes all images referenced in CSS files and js/ts files to be inserted in-line if their size is less than 25k, and in other modules, if they exceed that size. In the next section, we will see an example of image bundling.

All plugins are referenced in the webpack 2 “plugins” section with further configurations in their constructors.

While all application specific source client files are contained in the ClientApp folder, all files bundled by webpack 2 are deployed in wwwroot/dist.

You don’t need to call webpack 2 from the command line to process all files, since the *UseWebpackDevMiddleware*middleware in Startup.cs automatically performs this job whenever the application is started in the development mode:

|  |
| --- |
| if (env.IsDevelopment())  {      app.UseDeveloperExceptionPage();      app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions      {          HotModuleReplacement = true      });  } |

The above code invokes webpack 2 with the “hot module replacement” feature.

When this feature on webpack 2 detects file changes, it automatically sends the updated modules to the browser, thus enabling the developer to see the effects of any change immediately, with no requirements of refreshing the page or restarting application debugging.

In the remainder of the article, we will analyze in detail the “pure SPA” model and how to add more SPA pages and components. Then we will see how to add other Razor Views that use knockout.js components with or without a router, and finally we will mix Razor and knockout.js code in standard views.

**Implementing Knockout.js Single Page Applications**

In this section, we will dive more deep into SPA specific details.

All components are defined in the ClientApp/components folder. The root component (app-root) registered in boot.ts creates a browser history object and then calls ko.applyBindings to start knockout.js:

|  |
| --- |
| ko.components.register('app-root', AppRootComponent);    ko.applyBindings({ history: createHistory() }); |

In Index.cshtml, the “history” property is passed as parameter to the app-root component that performs the whole job of configuring the SPA engine:

|  |
| --- |
| <app-root params="history: history"></app-root> |

The history object passed as a parameter to the app-root component is actually received by the constructor of its ViewModel defined in ClientApp\components\app-root\app-root.ts. It is used to initialize a CrossRoads based router that is defined in router.ts.

Router behavior is defined by the routes listed at the beginning of the app-root.ts :

|  |
| --- |
| const routes: Route[] = [      { url: '',              params: { page: 'home-page' } },      { url: 'counter',       params: { page: 'counter-example' } },      { url: 'fetch-data',    params: { page: 'fetch-data' } }  ]; |

These are passed to the router constructor, together with the history object:

|  |
| --- |
| this.\_router = new Router(params.history, routes) |

The remainder of app-root.ts registers all components used by the SPA.

The nav-menu component that is used as a main menu is loaded immediately and registered with:

|  |
| --- |
| ko.components.register('nav-menu', navMenu); |

..whereas all other modules use a call to “require” with the lazy loading technique explained in the previous section:

|  |
| --- |
| ko.components.register('home-page',      require('bundle-loader?lazy!../home-page/home-page'));  ko.components.register('counter-example',      require('bundle-loader?lazy!../counter-example/counter-example'));  ko.components.register('fetch-data',      require('bundle-loader?lazy!../fetch-data/fetch-data')); |

When the file is processed, webpack 2 recognizes each lazy loading request and replaces the “require” call with another call. This call instead of returning a ViewModel/template pair, returns a loader function that must be called to download such a pair from the server, the first time the component is invoked.

That is why a [custom knockout.js component loader](http://knockoutjs.com/documentation/component-loaders.html) that may adequately handle the loader function is registered in webpack-component-loader.ts. Since this component loader must be the preferred one, it is added at the beginning of the list of all loaders with an unshift operation:

|  |
| --- |
| ko.components.loaders.unshift({      loadComponent: (name, componentConfig, callback) => {          if (typeof componentConfig === 'function') {              // It's a lazy-loaded webpack bundle              (componentConfig as any)(loadedModule => {                  // Handle TypeScript-style default exports                  if (loadedModule.\_\_esModule && loadedModule.default) {                      loadedModule = loadedModule.default;                  }                    // Pass the loaded module to KO's default loader                  ko.components.defaultLoader                      .loadComponent(name, loadedModule, callback);              });          } else {              // It's something else - let another component loader handle it              callback(null);          }      }  }); |

The code above specifies a [loadComponent](http://knockoutjs.com/documentation/component-loaders.html" \t "_blank) function to be invoked immediately before the component is instantiated. It is passed the component name as first parameter, and a callback, that is called once the component is ready as the last parameter. The second parameter is the load function returned by the “require” call in the component registration.

This function is invoked and passed a lambda callback that in turn, is invoked once the component has been successfully downloaded from the server.

The lambda callback does the following:

* receives the result of the downloaded module invocation in the loadedModule parameter,
* does some processing to conform to the TypeScript module export conventions, and
* finally calls knockout.js default loader passing it loadedModule that now contains the actual ViewModel/template pair that defines the required component.

app-root template prepares the place to load components that act as SPA pages, invoking the component binding with the component name contained in the current route.

|  |
| --- |
| <div class='container-fluid'>      <div class='row'>          <div class='col-sm-3'>              <nav-menu params='route: route'></nav-menu>          </div>          <div class='col-sm-9'               data-bind='component: {name: route().page, params: route }'>            </div>      </div>  </div> |

This way, when the user clicks a link, a new route becomes the current route, and the component name it contains is passed to the component binding, which in turn causes the component be downloaded from the server and instantiated.

**Downloading data from a Controller**

The fetch-data component scaffolded by the SPA template shows how a component may get data from an Mvc controller.

It interacts with the server with the [isomorphic-fetch](https://www.npmjs.com/package/isomorphic-fetch) npm package that is a [window. fetch](https://davidwalsh.name/fetch) polyfill that works both server side with node.js, as well as on client side. More specifically, it uses [whatwg-fetch](https://www.npmjs.com/package/whatwg-fetch" \t "_blank) on client side and [node-fetch](https://www.npmjs.com/package/node-fetch) on server side.

In this simple example, component data is retrieved from the server as soon as the component is loaded, so the whole data retrieval code is enclosed in the ViewModel constructor in fetch-data.ts :

|  |
| --- |
| interface WeatherForecast {      dateFormatted: string;      temperatureC: number;      temperatureF: number;      summary: string;  }    class FetchDataViewModel {      public forecasts = ko.observableArray<WeatherForecast>();        constructor() {          fetch('/api/SampleData/WeatherForecasts')              .then(response => response.json()                  as Promise<WeatherForecast[]>)              .then(data => {                  this.forecasts(data);              });      }  } |

Data is retrieved from the WeatherForecasts action method of the SampleDataController controller, transformed into JavaScript object by calling the “response.json” method, and then inserted in the observable contained in the forecast property.

On the server side, data is generated randomly:

|  |
| --- |
| [Route("api/[controller]")]  public class SampleDataController : Controller  {      private static string[] Summaries = new[]      {          "Freezing", "Bracing", "Chilly", "Cool", "Mild",          "Warm", "Balmy", "Hot", "Sweltering", "Scorching"      };        [HttpGet("[action]")]      public IEnumerable<WeatherForecast> WeatherForecasts()      {          var rng = new Random();          return Enumerable.Range(1, 5).Select(index => new WeatherForecast          {              DateFormatted = DateTime.Now.AddDays(index).ToString("d"),              TemperatureC = rng.Next(-20, 55),              Summary = Summaries[rng.Next(Summaries.Length)]          });      }        public class WeatherForecast      {          public string DateFormatted { get; set; }          public int TemperatureC { get; set; }          public string Summary { get; set; }            public int TemperatureF          {              get              {                  return 32 + (int)(TemperatureC / 0.5556);              }          }      }  } |

The fetch-data template in fetch-data.html contains typical knockout.js bindings to iterate on a collection:

|  |
| --- |
| <p data-bind='ifnot: forecasts'><em>Loading...</em></p>    <table class='table' data-bind='if: forecasts'>      <thead>          <tr>              <th>Date</th>              <th>Temp. (C)</th>              <th>Temp. (F)</th>              <th>Summary</th>          </tr>      </thead>      <tbody data-bind='foreach: forecasts'>          <tr>              <td data-bind='text: dateFormatted'></td>              <td data-bind='text: temperatureC'></td>              <td data-bind='text: temperatureF'></td>              <td data-bind='text: summary'></td>          </tr>      </tbody>  </table> |

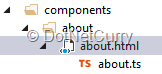
While data is being downloaded from the server, the forecasts property is empty, so the “ifnot” binding shows the paragraph content. Finally, when “forecast” observable is filled with data, the “foreach” binding shows the table rows bound to each item properties through the text binding.

**Creating a new SPA page**

In this subsection, we will see how to add a new SPA page.

As a first step, we need a new component.

Add a new folder called “about” in the “components” folder. Then create an “about.html”, and an “about.ts” files inside that folder:



Type the following in about.html:

|  |
| --- |
| <h1>About</h1>  <p>knockout.js SPA example</p> |

Now write the following code in about.ts:

|  |
| --- |
| import \* as ko from 'knockout';    class AboutPageViewModel {  }    export default {      viewModel: AboutPageViewModel,      template: require('./about.html')  }; |

At the moment, our view model does nothing, since our template contains just static html with no bindings. In the next section, we will add some logic to display an image. So the TypeScript module just exports the view model / template pair.

Before using our component, we must register it.

Registration is obligatory not only for SPA pages but also for components called from within views or other components. Registration can be added in the app-root.ts file next to all other SPA pages registration:

|  |
| --- |
| ko.components.register('about',          require('bundle-loader?lazy!../about/about')); |

The new SPA page is registered with lazy loading like all other pages.

Components that acts as SPA page must have a route associated with them. We may add a new route to the route list contained in the “routes” constant defined in app-root.ts:

|  |
| --- |
| const routes: Route[] = [      { url: '',              params: { page: 'home-page' } },      { url: 'counter',       params: { page: 'counter-example' } },      { url: 'fetch-data', params: { page: 'fetch-data' } },      { url: 'about', params: { page: 'about' } }  ]; |

Now our new page is working, we just need to link it somehow.

We must add a link in the main menu defined with the nav-menu component. Open nav-menu.html and add the new <li> tag below, at the end of its <ul> tag:

|  |
| --- |
| <li>      <a href='/about' data-bind='css: { active: route().page === "about" }'>          <span class='glyphicon glyphicon-tags'></span> About      </a>  </li> |

The css binding adds the active CSS class whenever the about page is the one currently displayed.

Run the project and click the about link. You should see the newly added page.

**Adding an image to the new page**

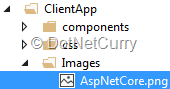
In this subsection, we will add an image to an already existing component, and will see how to bundle it with webpack 2, and how to render it with a knockout binding.

Images may be added directly to the wwwroot distribution folder and then referenced in all html files. However, you may do a *require* from ts files and then attach them to the Dom with the “attr” binding.

The main benefit of the second technique is that images may be preprocessed by various webpack 2 plugins. The knockout SPA template comes with the url-loader plugin that puts small images in-line instead of referencing their URLs.

However, you may use also [plugins for creating responsive images](https://github.com/Levelmoney/resize-image-loader).

Add an “Images” folder to the “ClentApp” folder and add the “AspNetCore.png” you can find in the source code of this article (or any other image you like):



Now *require* this image and insert it in a new property of the ViewModel of the component we defined in the previous subsection:

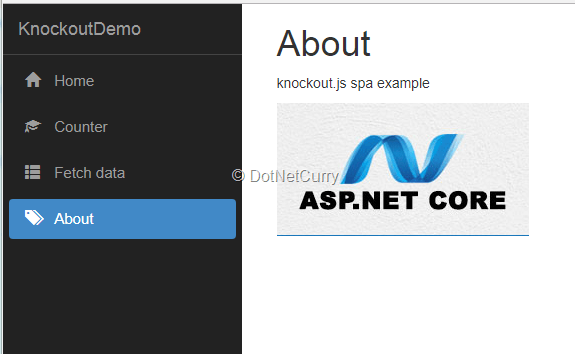
|  |
| --- |
| import \* as ko from 'knockout';  var img = require("../../Images/AspNetCore.png");    class AboutPageViewModel {      image = img;  }    export default {      viewModel: AboutPageViewModel,      template: require('./about.html')  }; |

We used “require” since with an “import” statement, the TypeScript compiler would have signalled an error because TypeScript import statement works only with TypeScript and JavaScript files.

Now we may add an <img> tag to the page template:

|  |
| --- |
| <h1>About</h1>  <p>knockout.js SPA example</p>  <img  data-bind="attr: {src: image}"/> |

Run the project and click the about link. You should see something like this:



**Several Razor Views with Knockout.js components**

A standard SPA application usually contains a single Razor view that hosts the whole SPA application, in our case the Index.cshtml.

Instead, what if you want to enhance several Razor views with knockout.js components?

Some Razor views may act as SPA connected with the browser history and with a router (like the Index.cshtml Razor View), while others may use knockout.js in a different way.

Hybrid SPA/Razor pages that mix both server side and SPA techniques offer great flexibility in practical applications because applications based entirely on SPA techniques offer a better interaction with the user but cost more, are more difficult to maintain and have a shorter life since client side techniques evolve very quickly.

Therefore, only web applications with a limited usage of SPA techniques and where a higher interaction with the user is necessary, appear quite attractive.

In this section, we will show how to build a similar scenario, by adding a new View that accesses the same contents available in the Index.cshtml view, using a bootstrap tab.

As a first step, we will modify our router so that it may properly handle relative links that point to different action methods. Open router.ts and observe the line of code below:

|  |
| --- |
| if (href && href.charAt(0) == '/') { |

This code has the purpose of selecting which urls to process with the SPA router and which ones with usual http requests: all relative urls are handled by the SPA router, while complete urls are handled with http requests.

We may allow relative links to be handled with usual http requests by adding to them a “data-external=true” attribute and by modifying the previous *if* statement as follows:

|  |
| --- |
| if (href && href.charAt(0) == '/'          && !$(target).attr('data-external')) { |

Now we may add a new action method to the home controller:

|  |
| --- |
| public IActionResult TabSelector()  {      return View();  } |

Also add a TabSelector view to Views\Home, and fill it with the following content:

|  |
| --- |
| @{      ViewData["Title"] = "Tab Based Page";  }  <div class='row'>      <div class='col-xs-9 col-xs-offset-1'>          <h1>@ViewData["Title"]</h1>          <a asp-action="Index" asp-controller="Home">              retun to SPA pages          </a>      </div>  </div> |

At the moment, the content is minimal. We will add the remainder of the code after having verified that everything works properly.

First, we need a link to the new page.

We can add it to the nav-menu component. Open nav-menu.html and add the following <li> at the end of its <ul>:

|  |
| --- |
| <a href='/Home/TabSelector' data-external="true">      <span class='glyphicon glyphicon-folder-open'></span> Tab page  </a> |

Run the project and verify that the SPA page and the new page we added, link to each other properly.

Before we may add the bootstrap tab with the knockout.js components in its panes in the TabSelector.cshtml view, we must prepare a TypeScript file for that view.

This TypeScript file must do the following:

1. It must import bootstrap JavaScript components, since bootstrap has been included in the vendor.js bundle that is a webpack 2 library, so it will execute only if it is imported by a non-library bundle.

2. It must register all knockout.js components used in the view

3. It must call ko.applyBindings to start knockout.js:

Add a new TypeScript file in the ClientApp folder and call it tab.ts. Then, fill it with the following content:

|  |
| --- |
| import './css/site.css';  import 'bootstrap';  import \* as ko from 'knockout';    import home from './components/home-page/home-page';  import counterExample from './components/counter-example/counter-example';  import fetchData from './components/fetch-data/fetch-data';  import about from './components/about/about';    ko.components.register('home-page', home);  ko.components.register('counter-example', counterExample);  ko.components.register('fetch-data', fetchData);  ko.components.register('about', about);    ko.applyBindings({}); |

This file will be the root of a new webpack 2 bundle, called “tab”. We must add the new bundle definition in the “entry” section of webpack.config.js that now becomes:

|  |
| --- |
| entry: {      'main': './ClientApp/boot.ts',      'tab': './ClientApp/tab.ts'  } |

Now we have everything we need to run the final version of TabSelector.cshtml:

|  |
| --- |
| @{      ViewData["Title"] = "Tab Based Page";  }  <div class='row'>      <div class='col-xs-9 col-xs-offset-1'>          <h1>@ViewData["Title"]</h1>          <a asp-action="Index" asp-controller="Home">              retun to SPA pages          </a>          <div>              <ul class="nav nav-tabs" role="tablist">                  <li role="presentation" class="active">                      <a href="#home" aria-controls="home" role="tab"                         data-toggle="tab">                          home                      </a>                  </li>                  <li role="presentation">                      <a href="#count" aria-controls="count" role="tab"                         data-toggle="tab">                          counter example                      </a>                  </li>                  <li role="presentation">                      <a href="#fetch" aria-controls="fetch" role="tab"                         data-toggle="tab">                          fetch data                      </a>                  </li>                  <li role="presentation">                      <a href="#about" aria-controls="about" role="tab"                         data-toggle="tab">                          about                      </a>                  </li>              </ul>              <div class="tab-content">                  <div role="tabpanel" class="tab-pane active" id="home">                      <home-page></home-page>                  </div>                  <div role="tabpanel" class="tab-pane" id="count">                      <counter-example></counter-example>                  </div>                  <div role="tabpanel" class="tab-pane" id="fetch">                      <fetch-data></fetch-data>                  </div>                  <div role="tabpanel" class="tab-pane" id="about">                      <about></about>                  </div>              </div>           </div>       </div>  </div>  @section scripts {      <script src="~/dist/tab.js" asp-append-version="true"></script>  } |

Run the project and click the tab link. You should see something like this:

**Adding Knockout.js bindings to Razor Views**

In this section, we will use Knockout.js just to enrich the Html generated with usual Razor views and tag helpers. Please note that this is something quite difficult to achieve with other client frameworks like angular and react.js.

We need a simple server side ViewModel to show how Asp.net Mvc views and knockout.js bindings may play well together. Let us add a new “ViewModels” folder to the project root, and then add a SimpleTextViewModel.cs file with the following content:

|  |
| --- |
| using System.ComponentModel.DataAnnotations;    namespace KnockoutDemo.ViewModels  {      public class SimpleTextViewModel      {          [Required, Display(Name = "simple text")]          public string SimpleText { get; set; }      }  } |

Then add “using KnockoutDemo.ViewModels” to the home controller, and also add two more action methods:

|  |
| --- |
| public IActionResult KnockoutMvc()  {      return View();  }  [HttpPost]  public IActionResult KnockoutMvc(SimpleTextViewModel model)  {      return View(model);  } |

Then add the Views\Home\ KnockoutMvc.cshtml view and fill it with the following:

|  |
| --- |
| @model KnockoutDemo.ViewModels.SimpleTextViewModel  @{      ViewData["Title"] = "Mvc + Knockou.js";  }  <div class='container-fluid'>      <div class='row'>          <div class='col-xs-9 col-xs-offset-1'>              <h1>@ViewData["Title"]</h1>              <a asp-action="Index" asp-controller="Home">                  retun to SPA pages              </a>              <form asp-controller="Home" asp-action="KnockoutMvc"                    method="post" class="form-horizontal">                  <div class="form-group">                      <label asp-for="SimpleText" class="control-label"></label>                      <div class="col-xs-12">                          <input asp-for="SimpleText" type="text" class="form-control"                                 data-bind="textInput: SimpleText" />                          <span asp-validation-for="SimpleText"                                class="text-danger"></span>                      </div>                    </div>                  <div class="form-group">                      <label for="mirror\_input" class="control-label">                          mirror:                      </label>                      <div class="col-xs-12">                          <p class="form-control-static"                             data-bind="text: SimpleText"></p>                      </div>                  </div>                  <div class="form-group">                      <button type="submit" class="btn btn-primary">Submit</button>                  </div>              </form>          </div>      </div>  </div>  @section scripts {      <script src="~/dist/mvc.js" asp-append-version="true"></script>  } |

The above code contains label, input and validation tag helpers. The input tag helper also contains a knockout.js binding to mirror everything the user writes in a paragraph. A form encloses the input field, so that the user may submit its input to the post version of the KnockoutMvc action method. We will see that both Mvc features and Razor/Mvc features works properly and complement each other.

In particular, the label will show the content of the server side ViewModel Display attribute, and the input will be required because of the RequiredAttribute added to the ViewModel SimpleText property.

The view references the mvc.js bundle that starts knockout.js and references the needed vendor.js library modules. We may create it as we did in the previous section.

Let us add an mvc.ts Typescript file to the ClientApp folder and fill it with:

|  |
| --- |
| import './css/site.css';  import 'bootstrap';  import \* as ko from 'knockout';    ko.applyBindings({      SimpleText:ko.observable<string>(          (document.getElementById('SimpleText') as HTMLInputElement).value)  }); |

The code is quite simple.

It starts knockout.js and initializes the client side ViewModel with the value contained in the page unique input field. In order to create the mvc.js bundle, we must add a new pair to the entry field in webpack.config.js that becomes:

|  |
| --- |
| entry: {      'main': './ClientApp/boot.ts',      'tab': './ClientApp/tab.ts',      'mvc': './ClientApp/mvc.ts'  } |

Everything should be properly set up now. We just need a link to the new page that we may add as usual, at the end of the <ul> in nav-menu.html:

|  |
| --- |
| <li>      <a href='/Home/KnockoutMvc' data-external="true">          <span class='glyphicon glyphicon-blackboard'></span> Mvc page      </a>  </li> |

Run the project and verify that both, form submission, validation (submit the empty input), and knockout.js work properly.

Conclusion:

We showed how Knockout offers a great degree of flexibility and cooperation with other frameworks.

It may be used to implement a standard SPA with the help of History.js and CrossRoad.js, but it may also be used in mixed solutions where knockout.js components and bindings, along with other frameworks like bootstrap.js are used in standard Mvc pages that are part of a mixed SPA/MVC application.

[**Download the entire source code**](http://bit.ly/dncm32-knockout-aspcore) of this article (Github).

*This article was technically reviewed by*[*Daniel Jimenez Garcia*](https://www.dotnetcurry.com/author/daniel-jimenez-garcia)*.*